USENIX Association

Proceedings of the
24th USENIX Security Symposium

August 12–14, 2015
Washington, D.C.

Conference Organizers

Program Chair
Jaeyeon Jung, *Microsoft Research*

Deputy Program Chair
Thorsten Holz, *Ruhr-Universität Bochum*

Program Committee
Sadia Afroz, *University of California, Berkeley*
Devdatta Akhawe, *Dropbox*
Davide Balzarotti, *Eurecom*
Igor Bilogrevic, *Google*
Sasha Boldyreva, *Georgia Institute of Technology*
Joseph Bonneau, *Stanford University and Electronic Frontier Foundation*
Nikita Borisov, *University of Illinois at Urbana-Champaign*
David Brumley, *Carnegie Mellon University*
Kevin Butler, *University of Florida*
Juan Caballero, *IMDEA Software Institute*
Srdjan Capkun, *ETH Zürich*
Stephen Checkoway, *Johns Hopkins University*
Nicolas Christin, *Carnegie Mellon University*
Byung-Gon Chun, *Seoul National University*
George Danezis, *University College London*
Tamara Denning, *University of Utah*
Michael Dietz, *Google*
Adam Doupé, *Arizona State University*
Josiah Dykstra, *NSA Research*
Manuel Egele, *Boston University*
Serge Egelman, *University of California, Berkeley, and International Computer Science Institute*
William Enck, *North Carolina State University*
David Evans, *University of Virginia*
Matt Fredrikson, *University of Wisconsin—Madison*
Roxana Geambasu, *Columbia University*
Rachel Greenstadt, *Drexel University*
Chris Grier, *DataBricks*
Guofei Gu, *Texas A&M University*
Alex Halderman, *University of Michigan*
Nadia Heninger, *University of Pennsylvania*
Susan Hohenberger, *Johns Hopkins University*
Jean-Pierre Hubaux, *École Polytechnique Fédérale de Lausanne (EPFL)*
Cynthia Irvine, *Naval Postgraduate School*
Rob Johnson, *Stony Brook University*
Brent Byunghoon Kang, *Korea Advanced Institute of Science and Technology (KAIST)*
Taesoo Kim, *Georgia Institute of Technology*
Engin Kirda, *Northeastern University*
Tadayoshi Kohno, *University of Washington*
Farinaz Koushanfar, *Rice University*
Zhou Li, *RSA Labs*
David Lie, *University of Toronto*
Janne Lindqvist, *Rutgers University*
Long Lu, *Stony Brook University*
Stephen McCamant, *University of Minnesota*
Damon McCoy, *George Mason University*
Jonathan McCune, *Google*
Sarah Meiklejohn, *University College London*
David Molnar, *Microsoft Research*
Tyler Moore, *Southern Methodist University*
Nick Nikiforakis, *Stony Brook University*
Cristina Nita-Rotaru, *Purdue University*
Zachary N. J. Peterson, *California Polytechnic State University*
Michalis Polychronakis, *Stony Brook University*
Adrienne Porter Felt, *Google*
Georgios Portokalidis, *Stevens Institute of Technology*
Niels Provos, *Google*
Benjamin Ransford, *University of Washington*
Thomas Ristenpart, *University of Wisconsin—Madison*
Will Robertson, *Northeastern University*
Franziska Roesner, *University of Washington*
Nitesh Saxena, *University of Alabama at Birmingham*
Prateek Saxena, *National University of Singapore*
R. Sekar, *Stony Brook University*
Hovav Shacham, *University of California, San Diego*
Micah Sherr, *Georgetown University*
Elaine Shi, *University of Maryland, College Park*
Reza Shokri, *The University of Texas at Austin*
Cynthia Sturton, *The University of North Carolina at Chapel Hill*
Patrick Traynor, *University of Florida*
Ingrid Verbauwhede, *Katholieke Universiteit Leuven*
Giovanni Vigna, *University of California, Santa Barbara*
David Wagner, *University of California, Berkeley*
Ralf-Philipp Weinmann, *Comsecuris*
Xiaoyong Zhou, *Samsung Research America*

Invited Talks Chair
Angelos Keromytis, *DARPA*

Invited Talks Committee
Michael Bailey, *University of Illinois at Urbana-Champaign*
Damon McCoy, *George Mason University*
Gary McGraw, *Cigital*

Poster Session Co-Chairs
Adam Doupé, *Arizona State University*
Sarah Meiklejohn, *University College London*

Work-in-Progress Reports (WiPs) Coordinator
Tadayoshi Kohno, University of Washington

Steering Committee
Matt Blaze, *University of Pennsylvania*
Dan Boneh, *Stanford University*
Casey Henderson, *USENIX Association*
Tadayoshi Kohno, *University of Washington*
Niels Provos, *Google*
David Wagner, *University of California, Berkeley*
Dan Wallach, *Rice University*

External Reviewers

Ruba Abu-Salma
Sumayah Alrwais
Abhishek Anand
Ben Andow
Elias Athanasopoulos
Micheal Bailey
Lucas Ballard
Manuel Barbosa
Vincent Bindschaedler
Bruno Blanchet
Bill Bolosky
Aylin Caliskan-Islam
Jan Camenisch
Nicholas Carlini
Henry Carter
Sang Kil Cha
Peter Chapman
Dominic Chen
Shuo Chen
Brian Cho
John Chuang
Mariana D'Angelo
Italo Dacosta
Thurston Dang
Soteris Demetriou
Zakir Durumeric
Antonio Faonio
Daniel Figueiredo
Christopher Fletcher
Afshar Ganjali
Christina Garman
Christina Garman
Behrad Garmany
Robert Gawlik
Martin Georgiev
Kevin Hong
Amir Houmansadr
Wei Huang
Yan Huang
Zhen Huang
Kevin Huguenin
Thomas P. Jakobsen
David Jensen
Seny Kamara
Ehsan Kazemi
Ehsan Kazemi
Erin Kenneally
Beom Heyn Kim
Benjamin Kollenda
Philipp Koppe
Sangmin Lee
Yeonjoon Lee
Jay Lorch
Paul D. Martin
Matthew Maurer
Travis Mayberry
Abner Mendoza
Ian Miers
Ian Miers
Andrew Miller
Dhaval Miyani
Manar Mohamed
Thierry Moreau
Alex Moshchuk
Dibya Mukhopadhyay
Muhammad Naveed
Ajaya Neupane
Giang Nguyen
Rishab Nithyanand
Sukwon Oh
Alexandra Olteanu
Rebekah Overdorf
Xiaorui Pang
Pedram Pedarsani
Riccardo Pelizzi
Anh Pham
Benny Pinkas
Rui Qiao
Moheeb Abu Rajab
Bradley Reaves
Ling Ren
Michael Rushanan
Nolen Scaife
Stuart Schechter
Rohan Sehgal
Maliheh Shirvanian
Babins Shrestha
Prakash Shrestha
Shridatt Sugrim
Edward Suh
Laszlo Szekeres
Henry Tan
George Theodorakopoulos
Kurt Thomas
Rijnard van Tonder
Marie Vasek
Haopei Wang
Fengguo Wei
Michelle Wong
Maverick Woo
Eric Wustrow
Lei Xu
Guangliang Yang
Jun Yuan
Kan Yuan
Jialong Zhang
Kehuan Zhang
Mingwei Zhang
Nan Zhang

24th USENIX Security Symposium
August 12–14, 2015
Washington, D.C.

Wednesday, August 12

Measurement: We Didn't Start the Fire

Now You're Just Something That I Used to Code

Tic-Attack-Toe

Word Crimes

Sock It To Me: TLS No Less

Forget Me Not

Thursday, August 13

Operating System Security: It's All About the Base

(Thursday, August 13, continues on next page)

Ace Ventura: PETS Detective

ORAMorama!

But Maybe All You Need Is Something to Trust

PELCGB

And the Hackers Gonna Hack, Hack, Hack, Hack, Hack

It's a Binary Joke: Either You Get It, or You Don't

Friday, August 14

Pain in the App

Oh, What a Tangled Web We Weave

(Friday, August 14, continues on next page)

The World's Address: An App That's Worn

ADDioS!

Attacks: I Won't Let You Down

How Do You Secure a Cloud and Pin it Down?

Knock Knock. Who’s There? Icy. Icy who? I See You Too

How Do You Solve a Problem Like M-al-ware?

The Supplement to the Proceedings of the 22nd USENIX Security Symposium follows.

Message from the 24th USENIX Security Symposium Program Chair

Welcome to the 24th USENIX Security Symposium in Washington, D.C.!

I hope you enjoy the technical program, hallway track, and fun evening events in the next three days. USENIX Security has been a premier venue for security and privacy research and I look forward to seeing the lasting impact that the papers of this year will make in years to come.

After agreeing to chair USENIX Security '15 Program Committee (PC), I sought feedback on different approaches to reviewing by reaching out to former chairs of USENIX Security and to chairs of the IEEE Symposium on Security and Privacy, ACM CCS, NSDI, ACM SIGCOMM, ACM CHI, and UbiComp. I also read chair reports from ASPLOS and ICSE. While no process will ever be perfect, I hope future conferences will be able to benefit from what we learned at USENIX Security this year.

Selection of Program Committee: I was very lucky to have a fantastic set of rock stars from our field who volunteered to serve on the program committee this year. I analyzed the topics of USENIX Security 2014 submissions and grouped them into seven areas and allocated the number of PC members to invite based on the number of expected submissions per area. To diversify the PC, I had a target of at least 20% PC members from each of four categories: outside the US, not from academia, not male, and new to the USENIX Security PC. To cope with the growth of submissions, I divided the PC into those required to attend the PC meeting ("attending") and those who were not ("remote") and provisioned the PC such that the review load was kept fewer than 20 submissions per member. 36 volunteers served as attending PC members and 39 served as remote PC members.

First round of reviews (Feb. 26–Apr. 2, 2015): We received 426 submissions, a 22% increase over the past year! 19 papers were desk rejected due to a violation of submission requirements and the rest were assigned to at least two reviewers per submission. The program committee spent one week on online discussion once reviews had been collected. As in past years, we decided to finalize decisions in the first round for a subset of papers that had confident reviews and did not appear to have a chance of acceptance. While in prior years we have used a similar process to decide the outcomes of many submissions at the end of the first round, the decision to issue early notifications and provide early access to reviews is new this year. 228 papers (54%) were rejected in the first round of decisions.

Second round of reviews (Apr. 3–May 6, 2015): Most papers received at least two more reviews in the second round. After the reviewing deadline, the program committee spent an additional two weeks discussing these papers using an online forum. Each paper was assigned to a discussion lead whose responsibility was to summarize reviews and drive a consensus among the reviewers between "suggest accept," "suggest reject," and "discuss." 22 papers received a "suggest accept" recommendation; 94 "suggest reject"; and 82 "discuss."

Un-blinding papers (May 6, 2015): Outcomes and discussion points were finalized for each paper and the deputy chair and I decided on the list of 88 papers to discuss at the PC meeting based on the recommendations. At that point the author names were made visible to reviewers. The un-blinding was helpful during the meeting to clarify conflicts and to help prevent authors from being punished for failing to cite their own work or from reviewers who might have a bias based on a false assumption regarding the authors' identity.

PC meeting (May 7–8, 2015, at Microsoft Research in Redmond, WA): 35 PC members attended the PC meeting and several remote PC members called into it. The PC began with a discussion of top five ranked papers and bottom five ranked papers to calibrate. To speed up a discussion, we allocated four minutes for a paper that was suggested to accept by the reviewers and eight minutes for the rest. The PC discussed 76 papers on the first day and 12 papers on the second day. After going through the list of 88 papers, the PC spent two extra hours discussing tabled papers and 14 papers that were voted to be resurrected. After the final decisions were made, we had accepted 67 papers, 16% of the submissions: all 22 papers tagged as "suggest accept," 44 papers tagged as "discuss." and 1 paper tagged as "suggest reject."

(Continued on page xii)

(Continued from page xi)

The program committee members spent countless hours not only reviewing papers but also discussing papers with each other online and in person. For instance, one controversial submission received seven reviews (including those from two external experts) and 44 comments online. On top of that, the PC spent an hour after dinner on the first day of the PC meeting to come to a consensus.

The technical program would not have been possible without contributions from the 75 program committee members and over 100 external reviewers who provided thoughtful reviews and recommendations and had to put up with nagging emails and reminders from me especially around the review deadlines. I would also like to thank Thorsten Holz for serving as the deputy chair; Angelos Keromytis for chairing the invited talks committee; Sarah Meiklejohn and Adam Doupé for serving as the poster session chairs; Tadayoshi Kohno for serving as the WiPs chair and mentoring a new chair like me; student volunteers Anna Simpson, Peter Ney, Adam Lerner, and Philipp Koppe, for scribing at the PC meeting and checking reviews; Eddie Kohler for adding new features into the already awesome HotCRP system that made paper triaging easier; Kevin Fu for creating funny session titles; Microsoft for sponsoring the PC meeting; Stuart Schechter for hosting an ice cream social and a post-PC meeting party; the USENIX staff, especially Casey Henderson and Michele Nelson for all the support throughout the process; and the authors of 426 papers for submitting their research for consideration. Finally, I would like to thank the USENIX steering committee to allow me to have this incredible opportunity to work with so many wonderful people.

Thanks to you all.

Jaeyeon Jung, *Microsoft Research*
USENIX Security '15 Program Chair

Post-Mortem of a Zombie: Conficker Cleanup After Six Years

Hadi Asghari, Michael Ciere and Michel J.G. van Eeten
Delft University of Technology

Abstract

Research on botnet mitigation has focused predominantly on methods to technically disrupt the command-and-control infrastructure. Much less is known about the effectiveness of large-scale efforts to clean up infected machines. We analyze longitudinal data from the sinkhole of Conficker, one the largest botnets ever seen, to assess the impact of what has been emerging as a best practice: national anti-botnet initiatives that support large-scale cleanup of end user machines. It has been six years since the Conficker botnet was sinkholed. The attackers have abandoned it. Still, nearly a million machines remain infected. Conficker provides us with a unique opportunity to estimate cleanup rates, because there are relatively few interfering factors at work. This paper is the first to propose a systematic approach to transform noisy sinkhole data into comparative infection metrics and normalized estimates of cleanup rates. We compare the growth, peak, and decay of Conficker across countries. We find that institutional differences, such as ICT development or unlicensed software use, explain much of the variance, while the national anti-botnet centers have had no visible impact. Cleanup seems even slower than the replacement of machines running Windows XP. In general, the infected users appear outside the reach of current remediation practices. Some ISPs may have judged the neutralized botnet an insufficient threat to merit remediation. These machines can however be magnets for other threats — we find an overlap between GameoverZeus and Conficker infections. We conclude by reflecting on what this means for the future of botnet mitigation.

1 Introduction

For years, researchers have been working on methods to take over or disrupt the command-and-control (C&C) infrastructure of botnets (e.g. [14, 37, 26]). Their successes have been answered by the attackers with ever more sophisticated C&C mechanisms that are increasingly resilient against takeover attempts [30].

In pale contrast to this wealth of work stands the limited research into the other side of botnet mitigation: cleanup of the infected machines of end users. After a botnet is successfully sinkholed, the bots or zombies basically remain waiting for the attackers to find a way to reconnect to them, update their binaries and move the machines out of the sinkhole. This happens with some regularity. The recent sinkholing attempt of GameoverZeus [32], for example, is more a tug of war between attackers and defenders, rather than definitive takedown action. The bots that remain after a takedown of C&C infrastructure may also attract other attackers, as these machines remain vulnerable and hence can be re-compromised.

To some extent, cleanup of bots is an automated process, driven by anti-virus software, software patches and tools like Microsoft's Malicious Software Removal Tool, which is included in Windows' automatic update cycle. These automated actions are deemed insufficient, however. In recent years, wide support has been established for the idea that Internet Service Providers (ISPs) should contact affected customers and help them remediate their compromised machines [39, 22]. This shift has been accompanied by proposals to treat large-scale infections as a public health issue [6, 8].

As part of this public health approach, we have seen the emergence of large-scale cleanup campaigns, most notably in the form of national anti-botnet initiatives. Public and private stakeholders, especially ISPs, collaborate to notify infected end users and help them clean their machines. Examples include Germany's Anti-Botnet Advisory Center (BotFrei), Australia's Internet Industry Code of Practice (iCode), and Japan's Cyber Clean Center (CCC, superseded by ACTIVE) [27].

Setting up large-scale cleanup mechanisms is cumbersome and costly. This underlines the need to measure whether these efforts are effective. The central question of this paper is: What factors drive cleanup rates of infected machines? We explore whether the leading national anti-botnet initiatives have increased the speed of cleanup.

We answer this question via longitudinal data from the sinkhole of Conficker, one the largest botnets ever seen. Conficker provides us with a unique opportunity to study the impact of national initiatives. It has been six years since the vulnerability was patched and the botnet was sinkholed. The attackers have basically abandoned it years ago, which means that infection rates are driven by cleanup rather than the attacker countermeasures. Still, nearly a million machines remain infected (see figure 1). The Conficker Working Group, the collective industry effort against the botnet, concluded in 2010 that remediation has been a failure [7].

Before one can draw lessons from sinkhole data, or from most other data sources on infected machines, several methodological problems have to be overcome. This paper is the first to systematically work through these issues, transforming noisy sinkhole data into comparative infection metrics and normalized estimates of cleanup rates.

For this research, we were generously given access to the Conficker sinkhole logs, which provide a unique long term view into the life of the botnet. The dataset runs from February 2009 until September 2014, and covers all countries — 241 ISO codes — and 34,000 autonomous systems. It records millions of unique IP addresses each year — for instance, 223 million in 2009, and 120 million in 2013. For this paper, we focus on bots located in 62 countries.

In sum, the contributions of this paper are as follows:

1. We develop a systematic approach to transform noisy sinkhole data into comparative infection metrics and normalized estimates of cleanup rates.

2. We present the first long term study on botnet remediation.

3. We provide the first empirical test of the best practice exemplified by the leading national anti-botnet initiatives.

4. We identify several factors that influence cleanup rates across countries.

2 Background

2.1 Conficker timeline and variants

In this section we will provide a brief background on the history of the Conficker worm, its spreading and defense mechanisms, and some milestones in the activities of the Conficker Working Group.

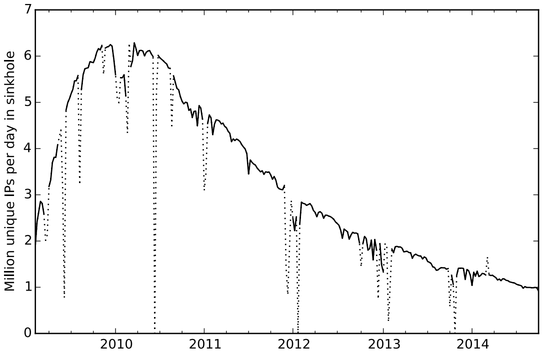


Figure 1: Conficker bots worldwide

The Conficker worm, also known as Downadup, was first detected in November 2008. The worm spread by exploiting vulnerability MS08-067 in Microsoft Windows, which had just been announced and patched. The vulnerability affected all versions of Microsoft Windows at the time, including server versions. A detailed technical analysis is available in [29]. Briefly put, infected machines scanned the IP space for vulnerable machines and infected them in a number steps. To be vulnerable, a machine needed to be unpatched and online with its NetBIOS ports open and not behind a firewall. Remarkably, a third of all machines had still not installed the patch by January 2009, a few months after its availability [11]. Consequently, the worm spread at an explosive rate. The malware authors released an update on December 29, 2008, which was named Conficker-B. The update added new methods of spreading, including via infected USB devices and shared network folders with weak passwords. This made the worm propagate even faster [7].

Infected machines communicated with the attackers via an innovative, centralized system. Every day, the bots attempted to connect to 250 new pseudo-randomly generated domains under eight different top-level domains. The attackers needed to register only one of these domains to reach the bots and update their instructions and binaries. Defenders, on the other hand, needed to block all these domains, every day, to disrupt the C&C. Another aspect of Conficker was the use of intelligent defense mechanisms, that made the worm harder to remove. It disabled Windows updates, popular anti-virus products, and several Windows security services. It also blocked access to popular security websites [29, 7].

Conficker continued to grow, causing alarm in the cybersecurity community about the potential scale of attacks, even though the botnet had not yet been very active at that point. In late January, the community — including Microsoft, ICANN, domain registries, anti-virus vendors, and academic researchers — responded by forming the Conficker Working Group [7, 31]. The most important task of the working group was to coordinate and register or block all the domains the bots would use to communicate, staying ahead of the Conficker authors. The group was mostly successful in neutralizing the botnet and disconnecting it from its owners; however, small errors were made on two occasions in March, allowing the attackers to gain access to part of the botnet population and update them to the C variant.

The Conficker-C variant had two key new features: the number of pseudo-randomly generated domains was increased to 50,000 per day, distributed over a hundred different TLDs, and a P2P update protocol was added. These features complicated the work of the working group. On April 9, 2009, Conficker-C bots upgraded to a new variant that included a scareware program which sold fake anti-virus at prices between $50–$100. The fake anti-virus program, probably a pay-per-install contract, was purchased by close to a million unwitting users, as was later discovered. This use of the botnet prompted law enforcement agencies to increase their efforts to pursue the authors of Conficker.[1] Eventually, in 2011, the U.S. Federal Bureau of Investigation, in collaboration with police in several other countries, arrested several individuals associated with this $72-million scareware ring. [21, 19]

2.2 National anti-botnet centers

Despite the successes of the cybersecurity community in neutralizing Conficker, a large number of infected machines still remained. This painful fact was recognized early on; in its 'Lessons Learned' document from 2010, the Conficker Working Group reported remediation as its top failure [7]. Despite being inactive, Conficker remains one of the largest botnets. As recent as June 2014, it was listed as the #6 botnet in the world by anti-virus vendor ESET [9]. This underlines the idea that neutralizing the C&C infrastructure in combination with automated cleanup tools will not eradicate the infected machines; some organized form of cleanup is necessary.

During the past years, industry and regulatory guidelines have been calling for increased participation of ISPs in cleanup efforts. For instance, the European Network and Information Security Agency [1], the Internet Engineering Task Force [22], the Federal Communications Commission [10], and the Organization for Economic Cooperation and Development [27] have all called upon ISPs to contact infected customers and help them clean up their compromised machines.

The main reason for this shift is that ISPs can identify and contact the owners of the infected machines, and provide direct support to end users. They can also quarantine machines that do not get cleaned up. Earlier work has found evidence that ISP mitigation can significantly impact end user security [40].

Along with this shift of responsibility towards ISPs, some countries have established national anti-botnet initiatives to support the ISPs and end users in cleanup efforts. The setup is different in each country, but typically it involves the collection of data on infected machines (from botnet sinkholes, honeypots, spamtraps, and other sources); notifying ISPs of infections within their networks; and providing support for end users, via a website and sometimes a call-center.

A number of countries have been running such centers, often as part of a public-private partnership. Table 1 lists the countries with active initiatives in late 2011, according to an OECD report [27]. The report also mentions the U.S. & U.K. as developing such initiatives. The Netherlands is listed as having 'ISP-specific' programs, for at that time, KPN and Ziggo — the two largest ISPs — were heading such programs voluntarily [39].[2] Finland, though not listed, has been a leader with consistently low infection rates for years. It has had a notification and cleanup mechanism in place since 2005, as part of a collaboration between the national CERT, the telco regulator and main ISPs [20, 25]. At the time of writing, other countries are starting anti-botnet centers as well. In the EU alone, seven new national centers have been announced [2]. These will obviously not impact the past cleanup rates of Conficker, but they do underwrite the importance of empirically testing the efficacy of this mitigation strategy.

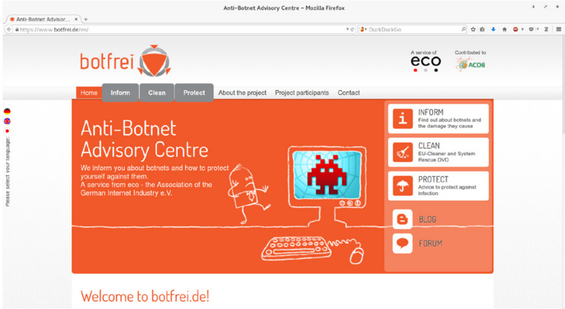
Figure 2 shows the website of the German anti-botnet advisory center, *botfrei*. The center was launched in 2010 by eco, the German Internet industry association, and is partially funded by the German government. The center does three things. First, it identifies users with infected PCs. Second, they inform the infected customers via their ISPs. Third, they offer cleanup support, through a website — with free removal tools and a forum — and

COUNTRY	INITIATIVE
Australia	Internet Industry Code of Practice (iCode)
Germany	German Anti-Botnet Initiative (BotFrei)
Ireland	Irish Anti-Botnet Initiative
Japan	Cyber Clean Center / ACTIVE
Korea	KrCERT/CC Anti-Botnet Initiative
Netherlands	Dutch Anti-Botnet Initiative (Abuse-Hub)

Table 1: List of countries with anti-botnet initiatives [27]

[1] Microsoft also set a $250,000 bounty for information leading to arrests.

[2] It has now been replaced by a wider initiative involving all main providers and covering the bulk of the broadband market.


Anti-Botnet Advisory Centre

We inform you about botnets and how to protect yourself against them.
A service from eco - the Association of the German Internet Industry e.V.

INFORM
Find out about botnets and the damage they cause

CLEAN
EU-Cleaner and System Rescue DVD

PROTECT
Advice to protect against infection

Welcome to botfrei.de!

Figure 2: The German Anti-Botnet Advisory Center website - botfrei.de

a call center [17]. The center covers a wide range of malware, including Conficker. We should mention that eco staff told us that much of the German Conficker response took place before the center was launched. In their own evaluations, the center reports successes in terms of the number of users visiting its website, the number of cleanup actions performed, and overall reductions in malware rates in Germany. Interestingly enough, a large number of users visit botfrei.de directly, without being prompted by their ISP. This highlights the impact of media attention, as well as the demand for proactive steps among part of the user population.

We only highlight Germany's botfrei program as an example. In short, one would expect that countries running similar anti-botnet initiatives to have higher cleanup rates of Conficker bots. This, we shall evaluate.

2.3 Related Work

Similar to other botnets, much of the work on the Conficker worm has focused predominantly on technical analysis, e.g., [29]. Other research has studied the worm's outbreak and modeled its infection patterns, e.g., [42], [16], [33] and [41]. There have also been a few studies looking into the functioning of the Working Group, e.g., [31]. None of this work looks specifically at the issue of remediation. Although [33] uses the same dataset as this paper to model the spread of the worm, their results are skewed by the fact that they ignore DHCP churn, which is known to cause errors in infection rates of up to one order of magnitude for some countries [37].

This paper also connects to the literature on botnet mitigation, specifically to cleanup efforts. This includes the industry guidelines we discussed earlier, e.g., [1], [27], [10] and [22]; as well as academic work that tries to model different mitigation strategies, e.g., [6], [18] and [13]. We contribute to this discussion by bringing longitudinal data to bear on the problem and empirically evaluating one of the key proposals to emanate from this literature. This expands some of our earlier work.

In a broader context, a large body of research focuses on other forms of botnet mitigation, e.g., [14, 37, 26, 30], modeling worm infections, e.g. [35, 44, 43, 28], and challenges in longitudinal cybersecurity studies. For the sake of brevity we will not cite more works in these areas here (— except for works used in other sections).

3 Methodology

Answering the central research question requires a number of steps. First, we set out to derive reliable estimates of the number of Conficker bots in each country over time. This involves processing and cleaning the noisy sinkhole data, as well as handling several measurement issues. Later, we use the estimates to compare infection trends in various countries, identify patterns and specifically see if countries with anti-botnet initiatives have done any better. We do this by by fitting a descriptive model to each country's time-series of infection rates. This provides us with a specific set of parameters, namely the growth rate, the peak infection level, and the decay rate. We explore a few alternative models and opt for a two-piece model that accurately captures these characteristics. Lastly, to answer the central question, we explore the relationship between the estimated parameters and a set of explanatory variables.

3.1 The Conficker Dataset

The Conficker dataset has four characteristics that make it uniquely suited for studying large-scale cleanup efforts. First, it contains the complete record of one sinkholed botnet, making it less convoluted than for example spam data, and with far fewer false positives. Second, it logs most of the population on a daily basis, avoiding limitations from seeing only a sample of the botnet. Third, the dataset is longitudinal and tracks a period of almost six years. Many sinkholes used in scientific research typically cover weeks rather than months, let alone six years. Fourth, most infection data reflects a mix of attacker and defender behavior, as well as different levels (global & local). This makes it hard to determine what drives a trend – is it the result of attacker behavior, defender innovation, or just randomness? Conficker, however, was neutralized early on, with the attackers losing control and abandoning the botnet. Most other global defensive actions (e.g., patching and sinkholing) were also done in early 2009. Hence, the infection levels in our dataset predominantly reflect cleanup efforts. These combined attributes make the Conficker dataset excellent for studying the policy effects we are interested in.

Raw Data

Our raw data comes from the Conficker sinkhole logs. As explained in the background section, Conficker bots used an innovative centralized command and control infrastructure. The bots seek to connect to a number of pseudo-random domains every day, and ask for updated instructions or binaries from their masters. The algorithm that generates this domain list was reverse engineered early on, and various teams, including the Conficker Working Group, seized legal control of these domains. The domains were then 'sinkholed': servers were set up to listen and log every attempt to access the domains. The resulting logs include the IP address of each machine making such an attempt, timestamps, and a few other bits of information.

Processing Sinkhole Logs

The raw logs were originally stored in plain text, before adoption of the *nmsg* binary format in late 2010. The logs are huge; a typical hour of logs in January 2013 is around half a gigabyte, which adds up to tens of terabytes per year. From the raw logs we extract the IP address, which in the majority of cases will be a Conficker A, B, or C bot (the sinkholed domains were not typically used for other purposes). Then, using the MaxMind GeoIP database [23] and an IP-to-ASN database based on Routeviews BGP data [4], we determine the country and Autonomous System that this IP address belonged to at that moment in time. We lastly count the number of unique IP addresses in each region per hour.

With some exceptions, we capture most Conficker bots worldwide. The limitations are due to sinkholes downtime; logs for some sinkholed domains not being handed over to the working group [7]; and bots being behind an egress firewall, blocking their access to the sinkhole. None of these issues however creates a systematic bias, so we may treat them as noise.

After processing the logs we have a dataset spanning from February 2009 to September 2014, covering 241 ISO country codes and 34,000 autonomous systems. The dataset contains approximately 178 million unique IP addresses per year. In this paper we focus on bots located in 62 countries, which were selected as follows. We started with the 34 members of the Organization for Economic Cooperation and Development (OECD), and 7 additional members of the European Union which are not part of the OECD. These countries have a common development baseline, and good data is available on their policies, making comparison easier. We add to this list 23 countries that rank high in terms of Conficker or spam bots — cumulatively covering 80 percent of all such bots worldwide. These countries are interesting from a cybersecurity perspective. Finally, two countries were removed due to severe measurement issues affecting their bot counts, which we will describe later. The full list of countries can be seen in figure 8 or in the appendix.

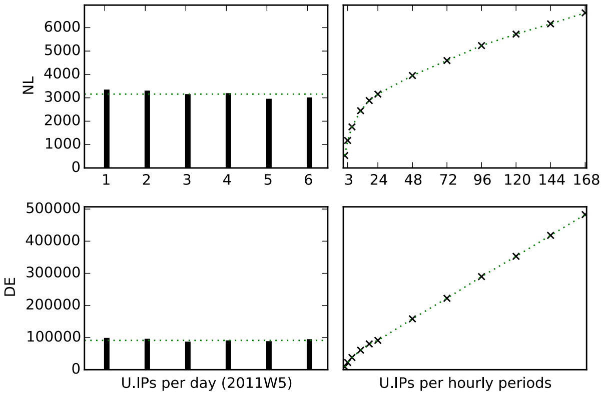


Figure 3: Unique IP counts over various time-periods

3.2 Counting bots from IP addresses

The Conficker dataset suffers from a limitation that is common among most sinkhole data and other data on infected machines, such as spam traps, firewall logs, and passive DNS records: one has to use IP addresses as a proxy for infected machines. Earlier research has established that IP addresses are coarse unique identifiers and they can be off by one order of magnitude in a matter of days [37], because of differences in the dynamic IP address allocation policies of providers (so-called *DHCP churn*). Simply put, because of dynamic addresses, the same infected machine can appear in the logs under multiple IP addresses. The higher the churn rate, the more over-counting.

Figure 3 visualizes this problem. It shows the count of unique Conficker IP addresses in February 2011 over various time periods — 3 hours, 12 hours, one day, up to a week. We see an interesting growth curve, non-linear at the start, then linear. Not all computers are powered on at every point in time, so it makes sense to see more IP addresses in the sinkhole over longer time periods. However, between the 6th and 7th day, we have most likely seen most infected machines already. The new IP addresses are unlikely to be new infections, as the daily count is stable over the period. The difference is thus driven by infected machines reappearing with a new IP address.

The figure shows IP address counts for the Netherlands and Germany. From qualitative reports we know that IP churn is relatively low in the Netherlands — an Internet subscriber can retain the same IP address for months — while in Germany the address typically changes every 24 hours. This is reflected in the figure: the slope for Germany is much steeper. Should one ignore the differences in churn rates among countries, and simply count unique IP addresses over a week, then a severe bias will be introduced against countries such as Germany. Using shorter time periods, though leading to under-counting, decreases this bias.[3] We settle for this simple solution: counting the average number of unique IPs *per hour*, thereby eliminating the churn factor. This hourly count will be a fraction of the total bot count, but that is not a problem when we make comparisons based on scale-invariant measures, such as cleanup rates.

Network Address Translation (NAT) and the use of HTTP proxies can also cause under-counting. This is particularly problematic if it happens at the ISP level, leading to large biases when comparing cleanup policies. After comparing subscriber numbers with IP address space size in our selection of countries, we concluded that ISP-level NAT is widely practiced in India. As we have no clear way of correcting such cases, we chose to exclude India from our analysis.

3.3 Missing measurements

The Conficker dataset has another problem that is also common: missing measurements. Looking back at figure 1, we see several sudden drops in bot counts, which we highlighted with dotted lines. These drops are primarily caused by sinkhole infrastructure downtime — typically for a few hours, but at one point even several weeks. These measurement errors are a serious issue, as they only occur in one direction and may skew our analysis. We considered several approaches to dealing with them. One approach is to model the measurement process explicitly. Another approach is to try and minimize the impact of aberrant observations by using robust curve-fitting methods. This approach adds unnecessary complexity and is not very intuitive. A third option is to pre-process the data using curve smoothing techniques; for instance by taking the exponentially weighted rolling average or applying the Hodrick-Prescott filter. Although not necessarily wrong, this also adds its own new biases as it changes data. The fourth approach, and the one that we use, is to detect and remove the outliers heuristically.

For this purpose, we calculate the distance between each weekly value in the global graph with the rolling median of its surrounding two months, and throw out the top 10%. This works because most bots log in about once a day, so the IP counts of adjacent periods are not independent. The IP count may increase, decrease, or slightly fluctuate, but a sudden decrease in infected machines followed by a sudden return of infections to the previous level is highly unlikely. The interested reader is referred to the appendix to see the individual graphs for all the countries with the outliers removed.[4]

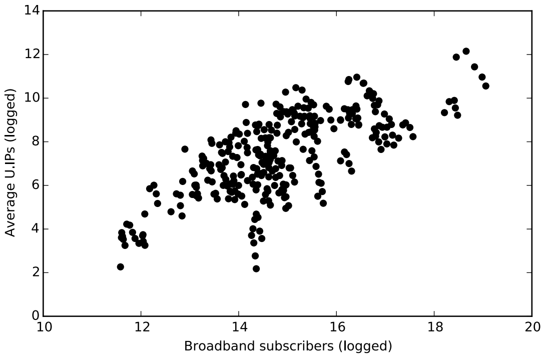


Figure 4: Conficker bots versus broadband subscribers

3.4 Normalizing bot counts by country size

Countries with more Internet users are likely to have more Conficker bots, regardless of remediation efforts. Figure 4 illustrates this. It thus makes sense to normalize the unique IP counts by a measure of country size; in particular if one is to compare peak infection rates. One such measure is the size of a country's IP space, but IP address usage practices vary considerably between countries. A more appropriate denominator and the one we use is the number of Internet broadband subscribers. This is available from a number of sources, including the Worldbank Development Indicators.

4 Modeling Infections

4.1 Descriptive Analysis

Figure 5 shows the Conficker infection trends for Germany, United States, France, and Russia. The x-axis is time; the y-axis is the average number of unique IP addresses seen per day in the sinkhole logs, corrected for churn. We observe a similar pattern: a period of rapid growth; a plateau period, where the number of infected machines peaks and remains somewhat stable for a short or longer amount of time; and finally, a period of gradual decline.

What explains these similar trends among countries, and in particular, the points in time where the changes

[3] Ideally, we would calculate a churn rate — the average number of IPs per bot per day — and use that to generate a good estimate of the actual number of bots. That is not an easy task, and requires making quite a number of assumptions.

[4] An extreme case was Malaysia, where the length of the drops and fluctuations spanned several months. This most likely indicates country-level egress filtering, prompting us to also exclude Malaysia from the analysis.

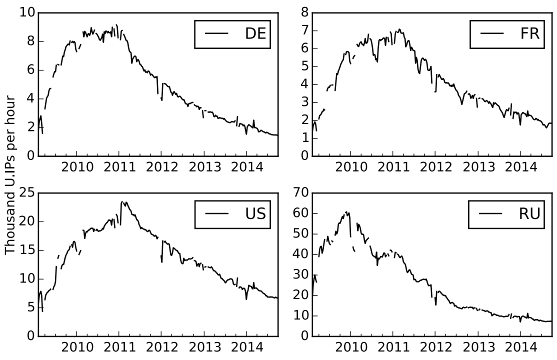


Figure 5: Conficker trends for four countries

occur on the graphs? At first glance, one might think that the decline is set off by some event — for instance, the arrest of the bot-masters, or a release of a patch. But this is not the case. As previously explained, all patches for Conficker were released by early 2009, while the worm continued spreading after that. This is because most computers that get infected with Conficker are "unprotected" — that is, they are either unpatched or without security software, in case the worm spreads via weak passwords on networks shares, USB drives, or domain controllers. The peak in 2010 – 2011 is thus the worm reaching some form of saturation where all vulnerable computers are infected. In the case of business networks, administrators may have finally gotten the worm's reinfection mechanisms under control [24].

Like the growth phase and the peak, the decline can also not be directly explained by external attacker behavior. Arrests related to Conficker occurred mid 2011, while the decline started earlier. In addition, most of the botnet was already out of the control of the attackers. What we are seeing appears to be a 'natural' process of the botnet. Infections may have spread faster in some countries, and cleanups may have been faster in others, but the overall patterns are similar across all countries.

4.2 Epidemic Models

It is often proposed in the security literature to model malware infections similarly as epidemics of infectious diseases, e.g. [28, 44]. The analog is that vulnerable hosts get infected, and start infecting other hosts in their vicinity; at some later point they are recovered or removed (cleaned, patched, upgraded or replaced).

This leads to multiple phases, similar to what we see for Conficker: in the beginning, each new infection increases the pressure on vulnerable hosts, leading to an explosive growth. Over time, fewer and fewer vulnerable hosts remain to be infected. This leads to a phase where the force of new infections and the force of recovery are locked in dynamic equilibrium. The size of the infected population reaches a plateau. In the final phase, the force of recovery takes over, and slowly the number of infections declines towards zero.

Early on in our modeling efforts we experimented with a number of epidemic models, but eventually decided against them. Epidemic models involve a set of latent compartments and a set of differential equations that govern the transitions between them — see [12] for an extensive overview. Most models make a number of assumptions about the underlying structure of the population and the propagation mechanism of the disease.

The basic models for instance assume constant transition rates over time. Such assumptions might hold to an acceptable degree in short time spans, but not over six years. The early works applying these models to the Code Red and Slammer worms [44, 43] used data spanning just a few weeks. One can still use the models even when the assumptions are not met, but the parameters cannot be then easily interpreted. To illustrate: the basic Kermack-McKendrick SIR model fits our data to a reasonable degree. However, we know that this model assumes no reinfections, while Conficker reinfections were a major problem for some companies [24].

More complex models reduce assumptions by adding additional latent variables. This creates a new problem: often when solved numerically, different combinations of the parameters fit the data equally well. We observed this for some countries with even the basic SIR model. Such estimates are not a problem when the aim is to predict an outbreak. But they are showstoppers when the aim is to compare and interpret the parameters and make inferences about policies.

4.3 Our model

For the outlined reasons, we opted for a simple descriptive model. The model follows the characteristic trend of infection rates, provides just enough flexibility to capture the differences between countries, and makes no assumptions about the underlying behavior of Conficker. It merely describes the observed trends in a small set of parameters.

The model consists of two parts: a logistic growth that ends in a plateau; followed by an exponential decay. Logistic growth is a basic model of self-limiting population growth, where first the rate of growth is proportional to the size of the existing population, and then declines as the natural limit is approached (— the seminal work of Staniford, et al. [35] also used logistic growth). In our case, this natural limit is the number of vulnerable hosts.

Exponential decay corresponds to a daily decrease of the number of Conficker bots by a fixed percentage. Figure 6 shows the number of infections per subscriber over

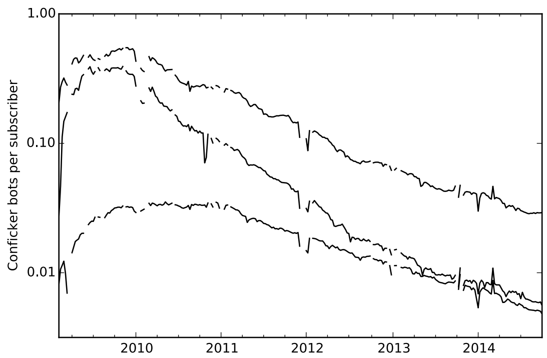


Figure 6: Conficker bots per subscriber on logarithm scale for (from top to bottom) Russia, Belarus, Germany.

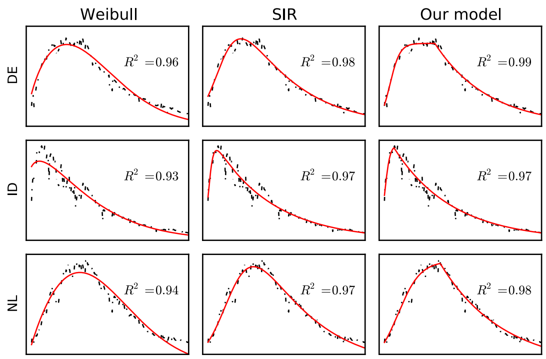


Figure 7: Comparison of alternative models

time for three countries on a logarithm scale. We see a downward-sloping straight line in the last phase that corresponds to an exponential decay: the botnet shrank by a more or less a constant percentage each day. We do not claim that the assumptions underpinning the logistic growth and the exponential decay models are fully satisfied, but in the absence of knowledge of the exact dynamics, their simplicity seems the most reasonable approach.

The model allows us to reduce the time series data for each country to these parameters: (1) the infection rate in the growth phase, (2) the peak number of infections, (3) the time at which this peak occurred, and (4) the exponential decay rate in the declining phase. We will fit our model on the time series for all countries, and then compare the estimates of these parameters.

Mathematically, our model is formulated as follows:

$$\text{bots}(t) = \begin{cases} \dfrac{K}{1+e^{-r(t-t_0)}}, & \text{if } t < t_P \\ \\ He^{-\gamma(t-t_P)}, & \text{if } t \geq t_P \end{cases} \tag{1}$$

where $\text{bots}(t)$ is the number of bots at time t, t_P is the time of the peak (where the logistic growth transitions to exponential decay), and H the height of the peak. The logistic growth phase has growth rate r, asymptote K, and midpoint t_0. The parameter γ is the exponential decay rate. The height of the peak is identified by the other parameters:

$$H = \frac{K}{1+e^{-r(t_P-t_0)}}.$$

4.4 Inspection of Model Fit

We fit the curves using the Levenberg-Marquardt least squares algorithm with the aid of the *lmfit* Python module. The results are point estimates; standard errors were computed by *lmfit* by approximating the Hessian matrix at the point estimates. With these standard errors we computed Wald-type confidence intervals (point estimate ± 2 s.e.) for all parameters. These intervals have no exact interpretation in this case, but provide some idea of the precision of the point estimates.

The reader can find plots of the fitted curves for all 62 countries in the appendix. The fits are good, with R^2 values all between 0.95 and 1. Our model is especially effective for countries with sharp peaks, that is, the abrupt transitions from growth to decay that can be seen in Hungary and South Africa, for example. For some countries, such as Pakistan and Ukraine, we have very little data on the growth phase, as they reached their peak infection rate around the time sinkholing started. For these countries we will ignore the growth estimates in further analysis. By virtue of our two-phase model, the estimates of the decay rates are unaffected by this issue.

We note that our model is deterministic rather than stochastic; that is, it does not account for one-time shocks in cleanup that lead to a lasting drop in infection rates. Nevertheless, we see that the data follows the fitted exponential decay curves quite closely, which indicates that bots get cleaned up at a constant rate and non-simultaneously.[5]

Alternative models: We tried fitting models from epidemiology (e.g. the SIR model) and reliability engineering (e.g. the Weibull curve), but they did not do well in such cases, and adjusted R^2 values were lower for almost all countries. Additionally, for a number of countries, the parameter estimates were unstable. Figure 7 illustrates why: our model's distinct phases captures the height of peak and exponential decay more accurately.

[5] The exception is China: near the end of 2010 we see a massive drop in Conficker infections. After some investigation, we found clues that this drop might be associated by a sudden spur in the adoption of IPv6 addresses, which are not directly observable to the sinkhole.

5 Findings

5.1 Country Parameter Estimates

Figure 8 shows the parameter estimates and their precision for each of the 62 countries: the growth rate, peak height, time of the peak, and the decay rate.

The variance in the peak number of infections is striking: between as little as 0.01% to over 1% of Internet broadband subscribers. The median is .1%. It appears that countries with high peaks tend to also have high growth rates, though we have to keep in mind that the growth rate estimates are less precise, because the data does not fully cover that phase. Looking at the peak height, it seems that this is not associated with low cleanup rates. For example, Belarus (BY) has the highest decay rate, but a peak height well above the median.

The timing of the peaks is distributed around the last weeks of 2010. Countries with earlier peaks are mostly countries with higher growth rates. This suggests that the time of the peak is simply a matter of when Conficker ran out of vulnerable machines to infect; a faster growth means this happens sooner. Hence, it seems unlikely that early peaks indicate successful remediation.

The median decay rate estimate is .009, which corresponds to a 37% decline per year ($100 \cdot (1 - e^{-.009 \cdot 52})$). In countries with low decay rates (around .005), the botnet shrank by 23% per year, versus over 50% per year on the high end.

5.2 National Anti-Botnet Initiatives

We are now in a position to address the paper's central question and to explore the effects of the leading national anti-botnet initiatives (ABIs). In figure 8, we have highlighted the countries with such initiatives as crosses. One would expect that these countries have slower botnet growth, a lower peak height, and especially a faster cleanup rate. There is no clear evidence for any of this; the countries with ABIs are all over the place. We do see some clustering on the lower end of the peak height graphs; however, this position is shared with a number of other countries that are institutionally similar (in terms of wealth for example) but not running such initiatives.

We can formally test if the population median is equal for the two groups using the Wilcoxon ranksum test. The *p*-value of the test when comparing the Conficker decay rate among the two sets of countries is 0.54, which is too large to conclude that the ABIs had a meaningful effect. It is somewhat surprising, and disappointing, to see no evidence for the impact of the leading remediation efforts on bot cleanup.

We briefly look at three possible explanations. The first one is that country trends might be driven by infections in other networks than those of the ISPs, as we know that the ABIs focus mostly on ISPs. This explanation fails, however, as can be seen in figure 2. The majority of the Conficker bots were located in the networks of the retail ISPs in these countries, compared to educational, corporate or governmental networks. This pattern held in 2010, the year of peak infections, and 2013, the decay phase, with one minor deviation: in the Netherlands, cleanup in ISP networks was faster than in other networks.

Country	ISP % 2010	ISP % 2013
AU	77%	74%
DE	89%	82%
FI	73%	69%
IE	72%	74%
JP	64%	67%
KR	83%	87%
NL	72%	**37%**
Others	81%	75%

Table 2: Conficker bots located in retail ISPs

A second explanation might be that the ABIs did not include Conficker in their notification and cleanup efforts. In two countries, Germany and the Netherlands, we were able to contact participants of the ABI. They claimed that Conficker sinkhole feeds were included and sent to the ISPs. Perhaps the ISPs did not act on the data — or at least not at a scale that would impact the decay rate; they might have judged Conficker infections to be of low risk, since the botnet had been neutralized. This explanation might be correct, but it also reinforces our earlier conclusion that the ABIs did not have a significant impact. After all, this explanation implies that the ABIs have failed to get the ISPs and their customers to undertake cleanup at a larger scale.

Given that cleanup incurs cost for the ISP, one could understand that they might decide to ignore sinkholed and neutralized botnets. On closer inspection, this decision seems misguided, however. If a machine is infected with Conficker, it means it is in a vulnerable — and perhaps infected — state for other malware as well. Since we had access to the global logs of the sinkhole for GameoverZeus — a more recent and serious threat — we ran a cross comparison of the two botnet populations. We found that based on common IP addresses, a surprising 15% of all GameoverZeus bots are also infected with Conficker. During six weeks at the end of 2014, the GameoverZeus sinkhole saw close to 1.9 million unique IP addresses; the Conficker sinkhole saw 12 million unique IP addresses; around 284 thousand addresses appear in both lists. Given that both malware types only infected a small percentage of the total pop-

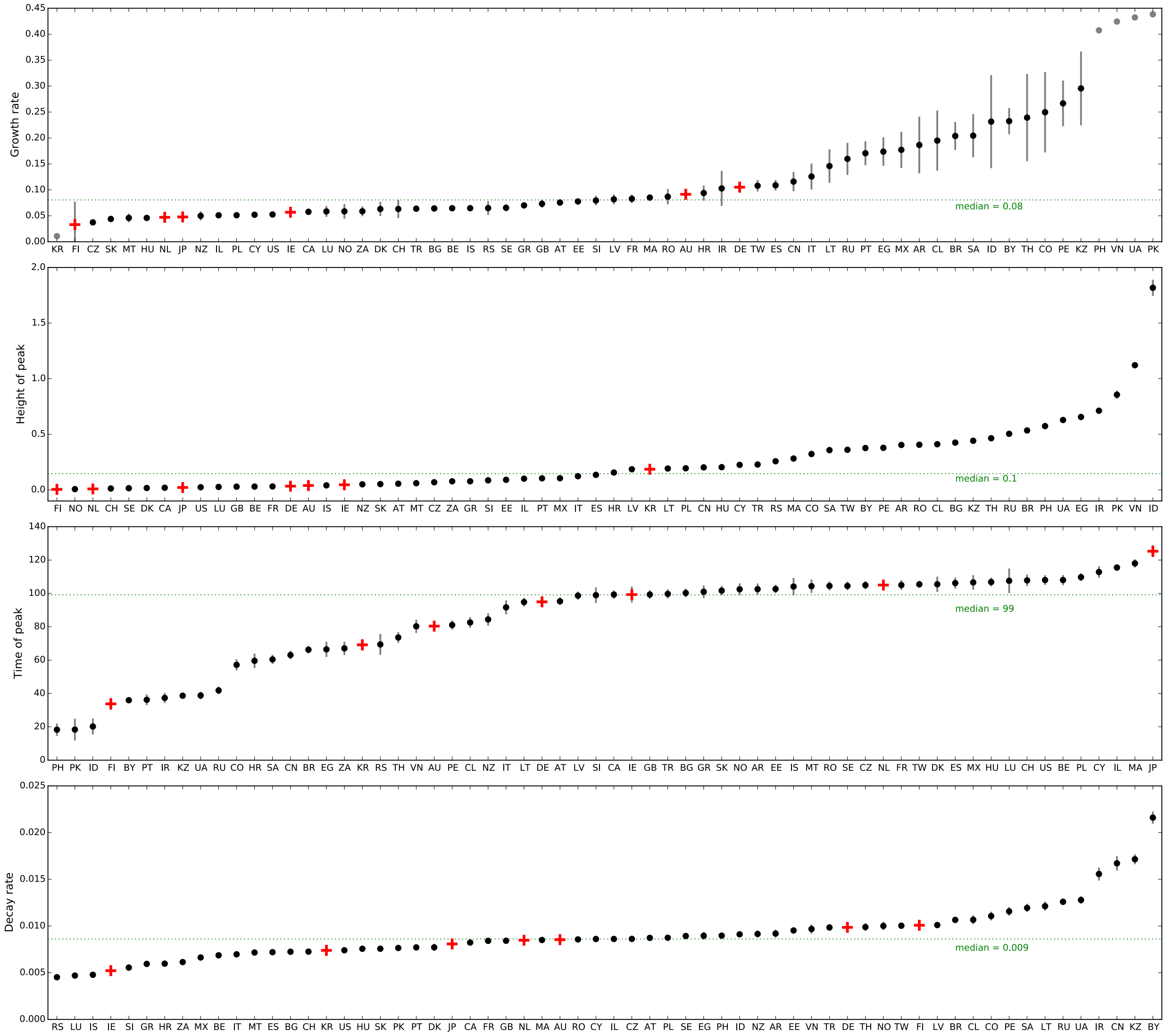


Figure 8: Parameter estimates and confidence intervals

ulation of broadband subscribers, this overlap is surprisingly large.[6] It stands in stark contrast to the findings of a recent study that systematically determined the overlap among 85 blacklists and found that most entries were unique to one list, and that overlap between independent lists was typically less than one percent [34]. In other words, Conficker bots should be considered worthwhile targets for cleanup.

5.3 Institutional Factors

Given that anti-botnet initiatives cannot explain the variation among the country parameters shown in figure 8, we turn our attention to several institutional factors that are often attributed with malware infection rates (e.g., see [40]). These are broadband access, unlicensed software use, and ICT development on a national level. In addition, given the spreading mechanism of Conficker, we also look at Operating System market shares, as well as PC upgrade cycles. We correlate these factors with the relevant parameters.

[6] The calculated overlap in terms of bots might be inflated as a result of both NAT and DHCP churn. Churn can in this case have both an over-counting and under-counting effect. Under-counting will occur if one bot appears in the two sinkholes with different IP addresses, as a result of different connection times to the sinkholes. Doing the IP comparisons at a daily level yields a 6% overlap, which is still considerable.

Correlating Growth Rate

Broadband access is often mentioned as a technological enabler of malware; in particular, since Conficker was a worm that spread initially by scanning for hosts to infect, one could expect its growth in countries with higher broadband speeds to be faster. Holding other factors constant, most epidemiological models would also predict this faster growth with increased network speeds. This turns out not to be the case. The Spearman correlation coefficient between average national broadband speeds, as reported by the International Telecommunication Union [15], and Conficker growth rate is in fact negative: -0.30. This is most probably due to other factors confounding with higher broadband speeds, e.g. national wealth. In any case, the effects of broadband access and speeds are negligible compared to other factors, and we will not pursue this further.

Correlating Height of Peak

As we saw, there is a wide dispersion between countries in the peak number of Conficker bots. What explains the large differences in peak infection rates?

Operating system market shares: Since Conficker only infects machines running Windows 2000, XP, Vista, or Server 2003/2008, some variation in peak height may be explained by differences in use of these operating systems (versus Windows 7 or non-Windows systems). We use data from StatCounter Global Stats [36], which is based on page view analytics of some three million websites. Figure 9 shows the peak height against the combined Windows XP and Vista market shares in January 2010 (other vulnerable OS versions were negligible). We see a strong correlation — with a Pearson correlation coefficient of 0.55. This in itself is perhaps not surprising.

Dividing the peak heights by the XP/Vista market shares gives us estimates of the *peak number of infections per vulnerable user*; we shall call this metric $\widetilde{\text{hp}}$. This metric allows for fairer comparisons between countries, as one would expect countries with higher market shares of vulnerable OS's to harbor more infections regardless of other factors. Interestingly, there is still considerable variation in this metric – the coefficient of variance is 1.2. We investigate two institutional factors that may explain this variation.

ICT development index is an index published by the ITU based on a number of well-established ICT indicators. It allows for benchmarking and measuring the digital divide and ICT development among countries (based on ICT readiness and infrastructure, ICT intensity and use, ICT skills and literacy [15]). This is obviously a broad indicator, and can indicate the ability to manage cybersecurity risks, including botnet cleanups, among both citizens and firms. Figure 10 shows this metric against $\widetilde{\text{hp}}$, and interestingly enough we see a strong correlation.

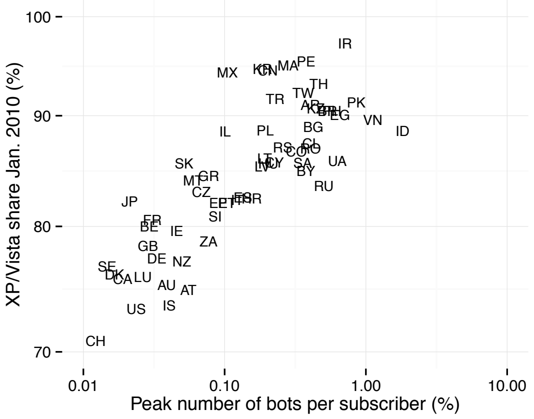


Figure 9: Bots versus XP & Vista use

Unlicensed software use or piracy rates are another oft mentioned factor influencing malware infection rates. In addition to the fact that pirated software might include malware itself, users running pirated OS's often turn off automatic updates, for fear of updates disabling their unlicensed software — even though Microsoft consistently states that it will also ship security updates to unlicensed versions of Windows [38]. Disabling automatic updates leaves a machine open to vulnerabilities, and stops automated cleanups. We use the unlicensed software rates calculated by the Business Software Alliance [5]. This factor also turns out to be strongly correlated with $\widetilde{\text{hp}}$, as shown in figure 10.

Since ICT development and piracy rates are themselves correlated, we use the following simple linear regression to explore thier joint association with peak Conficker infection rates:

$$\log(\widetilde{\text{hp}}) = \alpha + \beta_1 \cdot \text{ict-dev} + \beta_2 \cdot \text{piracy} + \varepsilon$$

where both regressors were standardized by subtracting the mean and dividing by two standard devia-

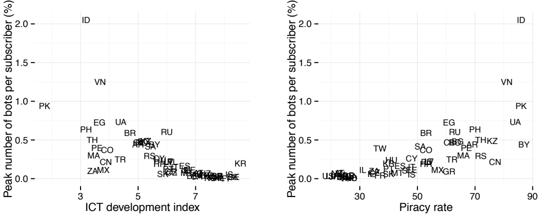


Figure 10: $\widetilde{\text{hp}}$ versus ICT development & piracy

tions. We use the logarithm of $\widetilde{hp}$ as it is a proportion. The least squares estimates (standard errors) are $\hat{\beta}_1 = -0.78(0.27)$, $p < 0.01$, and $\hat{\beta}_2 = 1.7(0.27)$, $p < 0.001$. These coefficients can be interpreted as follows: everything else kept equal, countries with low (one sd below the mean) ICT development have $e^{0.78} = 2.2$ times more Conficker bots per XP/Vista user at the peak than countries with high ICT development (one sd above the mean), and, similarly, countries with high piracy rates (one sd above the mean) have an $e^{1.7} = 5.5$ times higher peak than countries with low piracy rates (one sd below the mean). The R^2 of this regression is 0.78, which indicates that ICT development and piracy rates explain most of the variation in Conficker peak height.

Correlating Decay Rate

Although decay rates are less dispersed than peak heights, there are still noticeable differences among countries. Given the rather slow cleanup rates — the median of 0.009 translates to a 37% decrease in the number of bots after one year — one hypothesis that comes to mind is that perhaps some of the cleanup is being driven by users upgrading their OS's (to say Windows 7), or buying a new computer and disposing of the old fully.

For each country we estimated the **decay rate of the market share of Windows XP and Vista** from January 2011 to June 2013 using the StatCounter GlobalStats data. Figure 11 shows these decay rates versus Conficker decay rates. There is a weak correlation among the two, with a Spearman correlation coefficient of 0.26.

But more interesting and somewhat surprising is that in many countries, the Conficker botnet shrank at a slower pace than the market share of Windows XP / Vista (all countries below and to the right of the dashed line). Basically this means that the users infected with Conficker are less likely to upgrade their computers then the average consumer.[7]

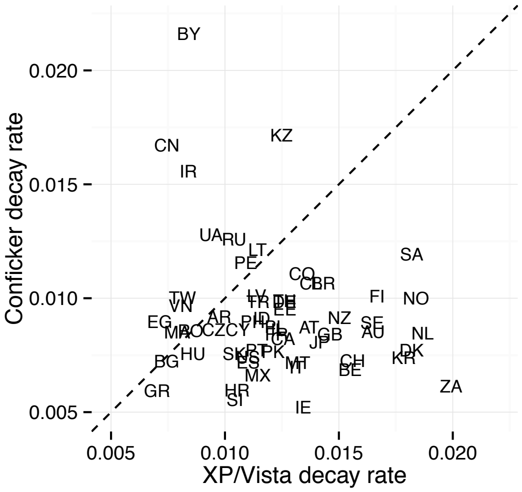


Figure 11: Conficker decay vs. XP/Vista decay

6 Discussion

We found that the large scale national anti-botnet initiatives had no observable impact on the growth, peak height, or decay of the Conficker botnet. This is surprising and unfortunate, as one would expect Conficker bots to be among those targeted for cleanup by such initiatives. We checked that the majority of bots were indeed located among the networks of ISPs, and also observed that some of these machines have multiple infections. Turning away from the initiatives and to institutional factors that could explain the differences among countries, we observed that the ICT development index and piracy rates can explain 78% of the variation in peak height, even after correcting for OS market shares. We also found that the Conficker cleanup rate is less than the average PC upgrade rate.

Perhaps not all security experts are surprised by these findings. They are nevertheless important in forming effective anti-botnet policies. We presented the research to an audience of industry practitioners active in botnet cleanup. Two North American ISPs commented that they informed their customers about Conficker infections — as part of the ISP's own policy, not a country-level initiative. They stated that some customers repeatedly ignored notifications, while others got re-infected soon after cleanup. Another difficulty was licensing issues requiring ISPs to point users to a variety of cleanup tool websites (e.g., on microsoft.com) instead of directly distributing a tool, which complicates the process for some users. Interestingly enough both ISPs ranked well with regards to Conficker peak, showing that their efforts did have a positive impact. Their challenges suggests areas for improvement.

Future work in this area can be taken in several directions. One is to test the various parameters against other independent variables — e.g., the number of CERTs, privacy regulation, and other governance indicators. A second avenue is to explore Conficker infection rates at the ISP level versus the country level. A random effects regression could reveal to what extent ISPs in the same country follow similar patterns. We might see whether particular ISPs differ widely from their country baseline, which would be interesting from an anti-botnet perspective. Third, it might be fruitful to contact a number of

[7] This difference between users who remain infected with Conficker and the average user might be more extreme in countries with a higher level of ICT development. This can be observed in the graph.

users still infected with Conficker in a qualitative survey, to see why they remain unaware or unworried about running infected machines. This can help develop new mitigation strategies for the most vulnerable part of the population. Perhaps some infections are forgotten embedded systems, not end users. Forth and more broadly is to conduct research on the challenges identified by the ISPs: notification mechanisms and simplifying cleanup.

7 Conclusion and Policy Implications

In this paper, we tackled the often ignored side of botnet mitigation: large-scale cleanup efforts. We explored the impact of the emerging best practice of setting up national anti-botnet initiatives with ISPs. Did these initiatives accelerate cleanup?

The longitudinal data from the Conficker botnet provided us with a unique opportunity to explore this question. We proposed a systematic approach to transform noisy sinkhole data into comparative infection metrics and normalized estimates of cleanup rates. After removing outliers, and by using the hourly Conficker IP address count per subscriber to compensate for a variety of known measurement issues, we modeled the infection trends using a two-part model. We thereby condensed the dataset to three key parameters for each country, and compared the growth, peak, and decay of Conficker, which we compared across countries.

The main findings were that institutional factors such as ICT development and unlicensed software use have influenced the spread and cleanup of Conficker more than the leading large scale anti-botnet initiatives. Cleanup seems even slower than the replacement of machines running Windows XP, and thus infected users appear outside the reach of remediation practices. At first glance, these findings seem rather gloomy. The Conficker Working Group, a collective effort against botnets, had noted remediation to be their largest failure [7]. We have now found that the most promising emerging practice to overcome that failure suffers similar problems.

So what can be done? Our findings lead us to identify several implications. First of all, the fact that peak infection levels strongly correlate with ICT development and software piracy, suggests that botnet mitigation can go hand in hand with economic development and capacity building. Helping countries develop their ICT capabilities can lower the global impact of infections over the long run. In addition, the strong correlation with software piracy suggests that automatic updates and unattended cleanups are some of the strongest tools in our arsenal. It support policies to enable security updates to install by default, and delivering them to all machines, including those running unlicensed copies [3]. Some of these points were also echoed by the ISPs mentioned in section 6.

Second, the fact that long-living bots appear in a reliable dataset — that is, one with few false positives — suggests that future anti-botnet initiatives need to commit ISPs to take action on such data sources, even if the sinkholed botnet is no longer a direct threat. These machines are vulnerable and likely to harbor other threats as well. Tracking these infections will be an important way to measure ISP compliance with these commitments, as well as incentivize cleanup for those users outside the reach of automated cleanup tools.

Third, given that cleanup is a long term effort, future anti-botnet initiatives should support, and perhaps fund, the long-term sustainability of sinkholes. This is a necessity if we want ISPs to act on this data.

A long term view is rare in the area of cybersecurity, which tends to focus on the most recent advances and threats. In contrast to C&C takedown, bot remediation needs the mindset of a marathon runner, not a sprinter. To conclude on a more optimistic note, Finland has been in the marathon for a longer time than basically all other countries. It pays off: they have been enjoying consistently low infection rates for years now. In other words, a long term view is not only needed, but possible.

Acknowledgment

The authors would like to explicitly thank Chris Lee, Paul Vixie and Eric Ziegast for providing us with access to the Conficker sinkhole and supporting our research.

We also thank Ning An, Ben Edwards, Dina Hadziosmanovic, Stephanie Forest, Jan Philip Koenders, Rene Mahieu, Hooshang Motarjem, Piet van Mieghem, Julie Ryan, as well as the participants of Microsoft DCC 2015 and USENIX reviewers for contributing ideas and providing feedback at various stages of this paper.

References

[1] Botnets: Measurement, detection, disinfection and defence.

[2] ADVANCED CYBER DEFENCE CENTRE. Support centers - advanced cyber defence centre (ACDC).

[3] ANDERSON, R., BHME, R., CLAYTON, R., AND MOORE, T. Security economics and the internal market. 00068.

[4] ASGHARI, H. Python IP address to autonomous system number lookup module.

[5] BUSINESS SOFTWARE ALLIANCE. BSA global software survey: The compliance gap: Home. 00000.

[6] CLAYTON, R. Might governments clean-up malware? 87–104.

[7] CONFICKER WORKING GROUP. Conficker working group: Lessons learned.

[8] EASTWEST INSTITUTE. The internet health model for cybersecurity. 00000.

[9] ESET. Global threat report - june 2014.

[10] FEDERAL COMMUNICATIONS COMISSION. U.s. anti-bot code of conduct (ABCs) for internet service providers (ISPs).

[11] GOODIN, D. Superworm seizes 9m PCs, 'stunned' researchers say.

[12] HEESTERBEEK, J. Mathematical epidemiology of infectious diseases: model building, analysis and interpretation. 02020.

[13] HOFMEYR, S., MOORE, T., FORREST, S., EDWARDS, B., AND STELLE, G. *Modeling internet-scale policies for cleaning up malware*. Springer, pp. 149–170.

[14] HOLZ, T., STEINER, M., DAHL, F., BIERSACK, E., AND FREILING, F. C. Measurements and mitigation of peer-to-peer-based botnets: A case study on storm worm. 1–9. 00375.

[15] INTERNATIONAL TELECOMMUNICATIONS UNION. Measuring the information society. 00002.

[16] IRWIN, B. A network telescope perspective of the conficker outbreak. In *Information Security for South Africa (ISSA), 2012*, IEEE, pp. 1–8.

[17] KARGE, S. The german anti-botnet initiative.

[18] KHATTAK, S., RAMAY, N. R., KHAN, K. R., SYED, A. A., AND KHAYAM, S. A. A taxonomy of botnet behavior, detection, and defense. 898–924.

[19] KIRK, J. Ukraine helps disrupt $72m conficker hacking ring.

[20] KOIVUNEN, E. *Why Wasnt I Notified?: Information Security Incident Reporting Demystified*, vol. 7127. Springer Berlin Heidelberg, pp. 55–70. 00000.

[21] KREBS, B. 72m USD scareware ring used conficker worm.

[22] LIVINGOOD, J., MODY, N., AND O'REIRDAN, M. Recommendations for the remediation of bots in ISP networks.

[23] MAXMIND. https://www.maxmind.com/en/geoip2-precision-country.

[24] MICROSOFT. Microsoft security intelligence report - how conficker continues to propogate.

[25] MICROSOFT. TelieSonera, european telecom uses microsoft security data to remove botnet devices from network.

[26] NADJI, Y., ANTONAKAKIS, M., PERDISCI, R., DAGON, D., AND LEE, W. Beheading hydras: performing effective botnet takedowns. ACM Press, pp. 121–132.

[27] OECD. Proactive policy measures by internet service providers against botnets.

[28] PASTOR-SATORRAS, R., CASTELLANO, C., VAN MIEGHEM, P., AND VESPIGNANI, A. Epidemic processes in complex networks. 00019.

[29] PORRAS, P., SAIDI, H., AND YEGNESWARAN, V. An analysis of confickers logic and rendezvous points.

[30] ROSSOW, C., ANDRIESSE, D., WERNER, T., STONE-GROSS, B., PLOHMANN, D., DIETRICH, C., AND BOS, H. SoK: P2pwned - modeling and evaluating the resilience of peer-to-peer botnets. In *2013 IEEE Symposium on Security and Privacy (SP)*, pp. 97–111. 00035.

[31] SCHMIDT, A. Secrecy versus openness: Internet security and the limits of open source and peer production.

[32] SHADOWSERVER FOUNDATION. Gameover zeus.

[33] SHIN, S., GU, G., REDDY, N., AND LEE, C. P. A large-scale empirical study of conficker. 676–690.

[34] SPRING, J. Blacklist ecosystem analysis. 00000.

[35] STANIFORD, S., PAXSON, V., WEAVER, N., AND OTHERS. How to own the internet in your spare time. In *USENIX Security Symposium*, pp. 149–167.

[36] STATCOUNTER. Free invisible web tracker, hit counter and web stats. 00000.

[37] STONE-GROSS, B., COVA, M., CAVALLARO, L., GILBERT, B., SZYDLOWSKI, M., KEMMERER, R., KRUEGEL, C., AND VIGNA, G. Your botnet is my botnet: Analysis of a botnet takeover. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, ACM, pp. 635–647.

[38] TOM'S HARDWARE. Microsoft: Pirated windows 7 will still get updates. 00000.

[39] VAN EETEN, M. J., ASGHARI, H., BAUER, J. M., AND TABATABAIE, S. Internet service providers and botnet mitigation: A fact-finding study on the dutch market.

[40] VAN EETEN, M. J., BAUER, J. M., ASGHARI, H., TABATABAIE, S., AND RAND, D. The role of internet service providers in botnet mitigation: An empirical analysis based on spam data.

[41] WEAVER, R. A probabilistic population study of the conficker-c botnet. In *Passive and Active Measurement*, Springer, pp. 181–190.

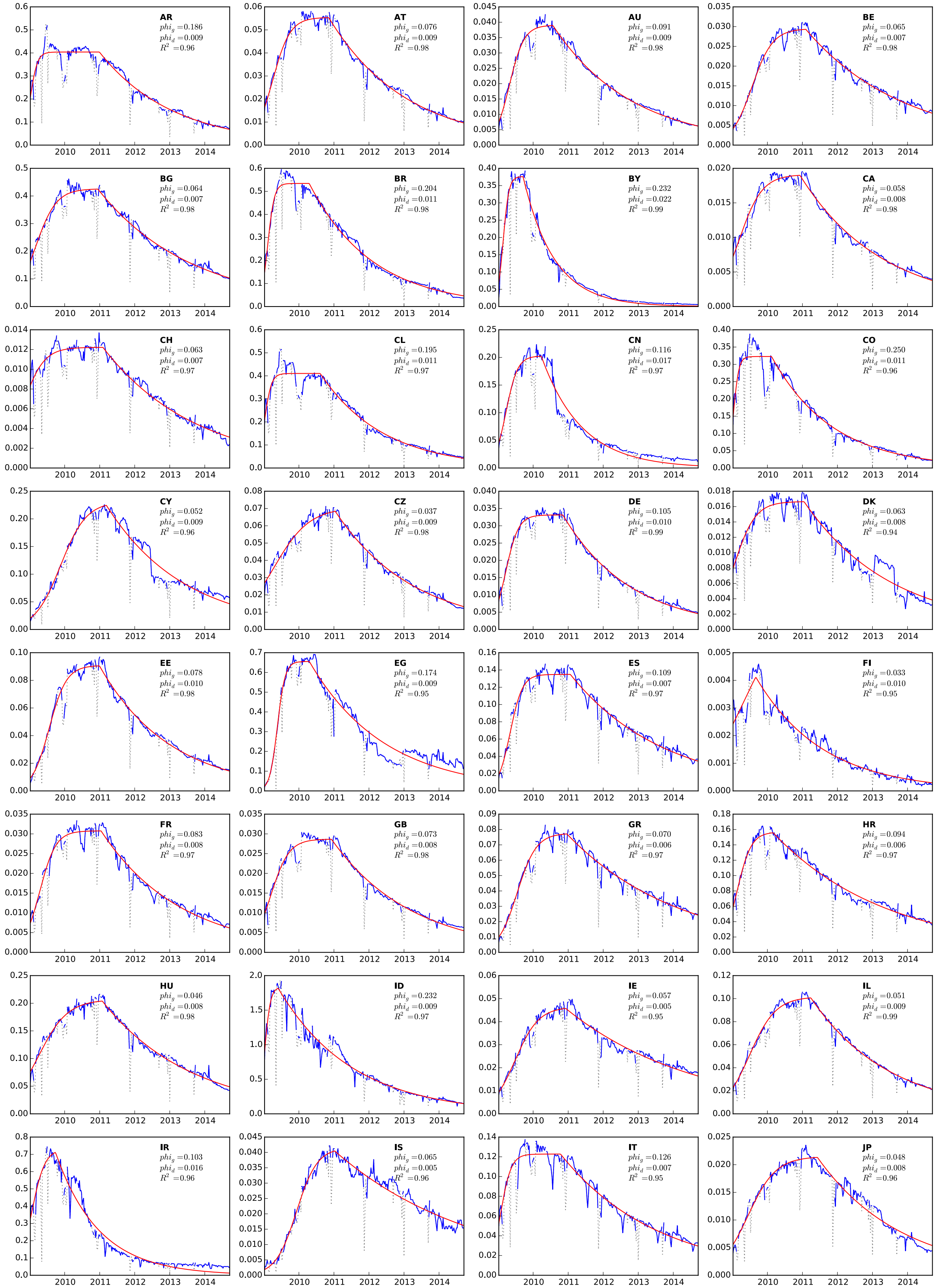
[42] ZHANG, C., ZHOU, S., AND CHAIN, B. M. Hybrid spreading of the internet worm conficker.

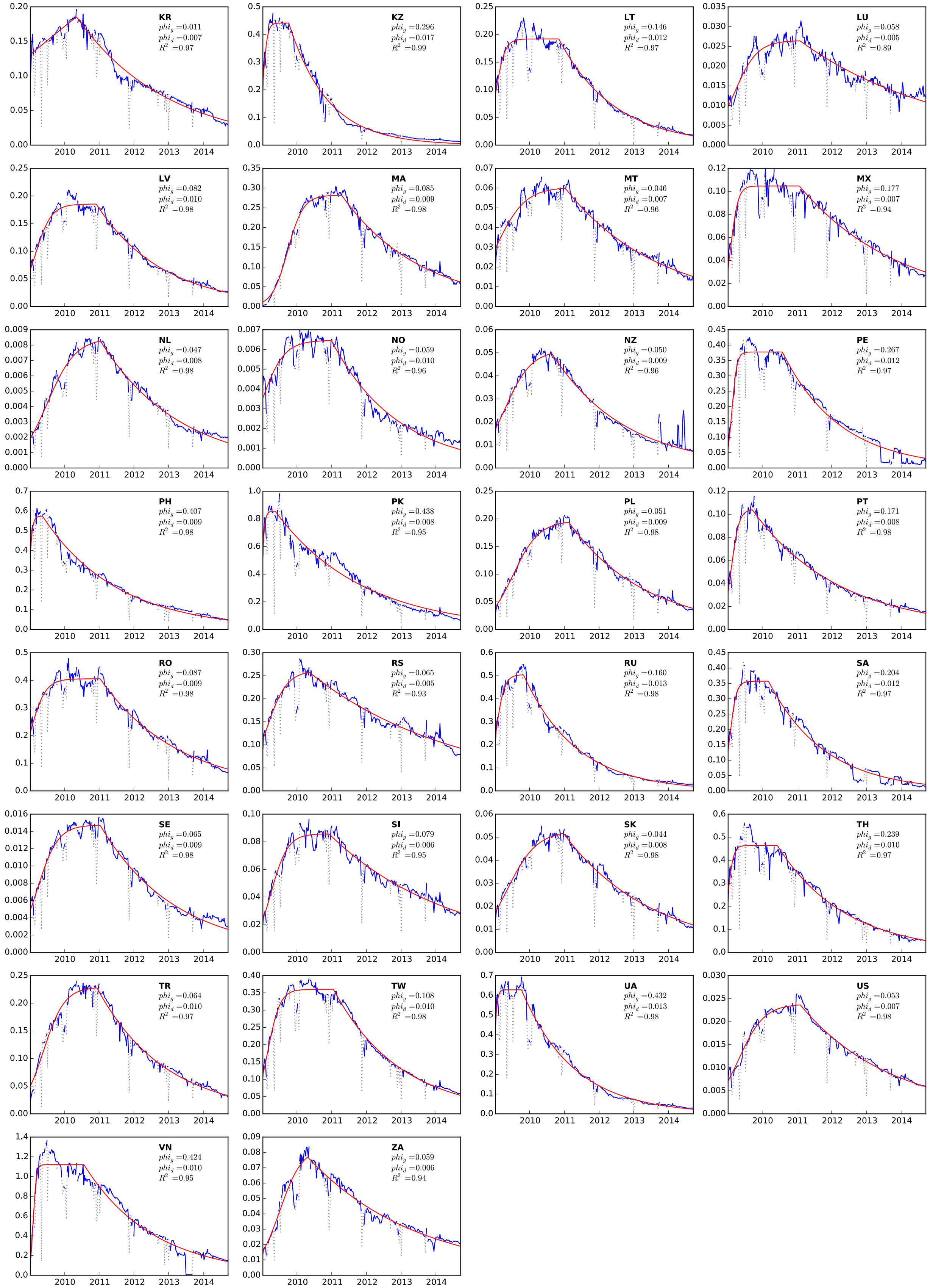
[43] ZOU, C. C., GAO, L., GONG, W., AND TOWSLEY, D. Monitoring and early warning for internet worms. In *Proceedings of the 10th ACM conference on Computer and communications security*, ACM, pp. 190–199.

[44] ZOU, C. C., GONG, W., AND TOWSLEY, D. Code red worm propagation modeling and analysis. In *Proceedings of the 9th ACM conference on Computer and communications security*, ACM, pp. 138–147.

Appendix - Individual Country Graphs

In this appendix we provide the model fit for all the 62 countries used in the paper. The graphs show the relative number of Conficker bots in each country - as measured by average unique Conficker IP addresses per hour in the sinkholes, divided by broadband subscriber counts for each country. (Please refer to the methodology section for the rationale). In each graph, the solid line (in blue) indicates the measurement; the dotted line (in gray) is removed outliers; and the smooth-solid line (in red) indicates the fitted model. The model has four parameters: growth and decay rates — given on the graph — and height and time of peak infections — deducible from the axes. The R^2 is also given for each country.


AR phi_g =0.186 phi_d =0.009 R^2 =0.96
AT phi_g =0.076 phi_d =0.009 R^2 =0.98
AU phi_g =0.091 phi_d =0.009 R^2 =0.98
BE phi_g =0.065 phi_d =0.007 R^2 =0.98
BG phi_g =0.064 phi_d =0.007 R^2 =0.98
BR phi_g =0.204 phi_d =0.011 R^2 =0.98
BY phi_g =0.232 phi_d =0.022 R^2 =0.99
CA phi_g =0.058 phi_d =0.008 R^2 =0.98
CH phi_g =0.063 phi_d =0.007 R^2 =0.97
CL phi_g =0.195 phi_d =0.011 R^2 =0.97
CN phi_g =0.116 phi_d =0.017 R^2 =0.97
CO phi_g =0.250 phi_d =0.011 R^2 =0.96
CY phi_g =0.052 phi_d =0.009 R^2 =0.96
CZ phi_g =0.037 phi_d =0.009 R^2 =0.98
DE phi_g =0.105 phi_d =0.010 R^2 =0.99
DK phi_g =0.063 phi_d =0.008 R^2 =0.94
EE phi_g =0.078 phi_d =0.010 R^2 =0.98
EG phi_g =0.174 phi_d =0.009 R^2 =0.95
ES phi_g =0.109 phi_d =0.007 R^2 =0.97
FI phi_g =0.033 phi_d =0.010 R^2 =0.95
FR phi_g =0.083 phi_d =0.008 R^2 =0.97
GB phi_g =0.073 phi_d =0.008 R^2 =0.98
GR phi_g =0.070 phi_d =0.006 R^2 =0.97
HR phi_g =0.094 phi_d =0.006 R^2 =0.97
HU phi_g =0.046 phi_d =0.008 R^2 =0.98
ID phi_g =0.232 phi_d =0.009 R^2 =0.97
IE phi_g =0.057 phi_d =0.005 R^2 =0.95
IL phi_g =0.051 phi_d =0.009 R^2 =0.99
IR phi_g =0.103 phi_d =0.016 R^2 =0.96
IS phi_g =0.065 phi_d =0.005 R^2 =0.96
IT phi_g =0.126 phi_d =0.007 R^2 =0.95
JP phi_g =0.048 phi_d =0.008 R^2 =0.96
2010 2011 2012 2013 2014


KR
phi_g =0.011
phi_d =0.007
R^2 =0.97
KZ
phi_g =0.296
phi_d =0.017
R^2 =0.99
LT
phi_g =0.146
phi_d =0.012
R^2 =0.97
LU
phi_g =0.058
phi_d =0.005
R^2 =0.89
LV
phi_g =0.082
phi_d =0.010
R^2 =0.98
MA
phi_g =0.085
phi_d =0.009
R^2 =0.98
MT
phi_g =0.046
phi_d =0.007
R^2 =0.96
MX
phi_g =0.177
phi_d =0.007
R^2 =0.94
NL
phi_g =0.047
phi_d =0.008
R^2 =0.98
NO
phi_g =0.059
phi_d =0.010
R^2 =0.96
NZ
phi_g =0.050
phi_d =0.009
R^2 =0.96
PE
phi_g =0.267
phi_d =0.012
R^2 =0.97
PH
phi_g =0.407
phi_d =0.009
R^2 =0.98
PK
phi_g =0.438
phi_d =0.008
R^2 =0.95
PL
phi_g =0.051
phi_d =0.009
R^2 =0.98
PT
phi_g =0.171
phi_d =0.008
R^2 =0.98
RO
phi_g =0.087
phi_d =0.009
R^2 =0.98
RS
phi_g =0.065
phi_d =0.005
R^2 =0.93
RU
phi_g =0.160
phi_d =0.013
R^2 =0.98
SA
phi_g =0.204
phi_d =0.012
R^2 =0.97
SE
phi_g =0.065
phi_d =0.009
R^2 =0.98
SI
phi_g =0.079
phi_d =0.006
R^2 =0.95
SK
phi_g =0.044
phi_d =0.008
R^2 =0.98
TH
phi_g =0.239
phi_d =0.010
R^2 =0.97
TR
phi_g =0.064
phi_d =0.010
R^2 =0.97
TW
phi_g =0.108
phi_d =0.010
R^2 =0.98
UA
phi_g =0.432
phi_d =0.013
R^2 =0.98
US
phi_g =0.053
phi_d =0.007
R^2 =0.98
VN
phi_g =0.424
phi_d =0.010
R^2 =0.95
ZA
phi_g =0.059
phi_d =0.006
R^2 =0.94
2010
2011
2012
2013
2014

Mo(bile) Money, Mo(bile) Problems: Analysis of Branchless Banking Applications in the Developing World

Bradley Reaves
University of Florida
reaves@ufl.edu

Nolen Scaife
University of Florida
scaife@ufl.edu

Adam Bates
University of Florida
adammbates@ufl.edu

Patrick Traynor
University of Florida
traynor@cise.ufl.edu

Kevin R.B. Butler
University of Florida
butler@ufl.edu

Abstract

Mobile money, also known as branchless banking, brings much-needed financial services to the unbanked in the developing world. Leveraging ubiquitous cellular networks, these services are now being deployed as smart phone apps, providing an electronic payment infrastructure where alternatives such as credit cards generally do not exist. Although widely marketed as a more secure option to cash, these applications are often not subject to the traditional regulations applied in the financial sector, leaving doubt as to the veracity of such claims. In this paper, we evaluate these claims and perform the first in-depth measurement analysis of branchless banking applications. We first perform an automated analysis of all 46 known Android mobile money apps across the 246 known mobile money providers and demonstrate that automated analysis fails to provide reliable insights. We subsequently perform comprehensive manual teardown of the registration, login, and transaction procedures of a diverse 15% of these apps. We uncover pervasive and systemic vulnerabilities spanning botched certification validation, do-it-yourself cryptography, and myriad other forms of information leakage that allow an attacker to impersonate legitimate users, modify transactions in flight, and steal financial records. These findings confirm that the majority of these apps fail to provide the protections needed by financial services. Finally, through inspection of providers' terms of service, we also discover that liability for these problems unfairly rests on the shoulders of the customer, threatening to erode trust in branchless banking and hinder efforts for global financial inclusion.

1 Introduction

The majority of modern commerce relies on cashless payment systems. Developed economies depend on the near instantaneous movement of money, often across great distances, in order to fuel the engines of industry. These rapid, regular, and massive exchanges have created significant opportunities for employment and progress, propelling forward growth and prosperity in participating countries. Unfortunately, not all economies have access to the benefits of such systems and throughout much of the developing world, physical currency remains the de facto means of exchange.

Mobile money, also known as branchless banking, applications attempt to fill this void. Generally deployed by companies outside of the traditional financial services sector (e.g., telecommunications providers), branchless banking systems rely on the near ubiquitous deployment of cellular networks and mobile devices around the world. Customers can not only deposit their physical currency through a range of independent vendors, but can also perform direct peer-to-peer payments and convert credits from such transactions back into cash. Over the past decade, these systems have helped to raise the standard of living and have revolutionized the way in which money is used in developing economies. Over 30% of the GDP in many such nations can now be attributed to branchless banking applications [39], many of which now perform more transactions per month than traditional payment processors, including PayPal [36].

One of the biggest perceived advantages of these applications is security. Whereas carrying large amounts of currency long distances can be dangerous to physical security, branchless banking applications can allow for commercial transactions to occur without the risk of theft. Accordingly, these systems are marketed as a secure new means of enabling commerce. Unfortunately, the strength of such claims from a *technical* perspective has not been publicly investigated or verified. Such an analysis is therefore critical to the continued growth of branchless banking systems.

In this paper, we perform the first comprehensive analysis of branchless banking applications. Through these efforts, we make the following contributions:

- **Analysis of Branchless Banking Applications:** We perform the first comprehensive security analysis of branchless banking applications. First, we use a well-known automated analysis tool on all 46 known Android mobile money apps across all 246 known mobile money systems. We then methodically select seven Android-based branchless banking applications from Brazil, India, Indonesia, Thailand, and the Phillipines with a combined user base of millions. We then develop and execute a comprehensive, reproducible methodology for analyzing the entire application communication flow. In so doing, we create the first snapshot of the global state of security for such applications.

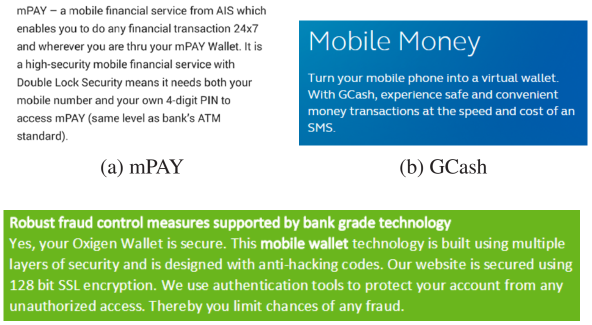
- **Identifications of Systemic Vulnerabilities:** Our analysis discovers pervasive weaknesses and shows that six of the seven applications broadly fail to preserve the integrity of their transactions. We then compare our results to those provided through automated analysis and show that current tools do not reliably indicate severe, systemic security faults. Accordingly, neither users nor providers can reason about the veracity of requests by the majority of these systems.

- **Analysis of Liability:** We combine our technical findings with the assignment of liability described within every application's terms of service, and determine that users of these applications have no recourse for fraudulent activity. Therefore, it is our belief that these applications create significant financial dangers for their users.

The remainder of this paper is organized as follows: Section 2 provides background information on branchless banking and describes how these applications compare to other mobile payment systems; Section 3 details our methodology and analysis architecture; Section 4 presents our findings and categorizes them in terms of the CWE classification system; Section 5 delivers discussion and recommendations for technical remediation; Section 6 offers an analysis of the Terms of Service and the assignment of liability; Section 7 discusses relevant related work; and Section 8 provides concluding remarks.

2 Mobile Money in the Developing World

The lack of access to basic financial services is a contributing factor to poverty throughout the world [17]. Millions live without access to basic banking services, such as value storage and electronic payments, simply because they live hours or days away from the nearest bank branch. Lacking this access makes it more difficult for the poor to save for future goals or prepare for future setbacks, conduct commerce remotely, or protect money against loss or theft. Accordingly, providing banking through mobile phone networks offers the promise of dramatically improving the lives of the world's poor.


mPAY – a mobile financial service from AIS which enables you to do any financial transaction 24x7 and wherever you are thru your mPAY Wallet. It is a high-security mobile financial service with Double Lock Security means it needs both your mobile number and your own 4-digit PIN to access mPAY (same level as bank's ATM standard).

(a) mPAY

Mobile Money

Turn your mobile phone into a virtual wallet. With GCash, experience safe and convenient money transactions at the speed and cost of an SMS.

(b) GCash

Robust fraud control measures supported by bank grade technology

Yes, your Oxigen Wallet is secure. This **mobile wallet** technology is built using multiple layers of security and is designed with anti-hacking codes. Our website is secured using 128 bit SSL encryption. We use authentication tools to protect your account from any unauthorized access. Thereby you limit chances of any fraud.

(c) Oxigen Wallet

Figure 1: Mobile money apps are heavily marketed as being safe to use. These screenshots from providers' marketing materials show the extent of these claims.

The M-Pesa system in Kenya [21] pioneered the *mobile money* service model, in which agents (typically local shopkeepers) act as intermediaries for deposits, withdrawals, and sometimes registration. Both agents and users interact with the mobile money system using SMS or a special application menu enabled by code on a SIM card, enabling users to send money, make bill payments, top up airtime, and check account balances. The key feature of M-Pesa and other systems is that their use does not require having a previously established relationship with a bank. In effect, mobile money systems are bootstrapping an alternative banking infrastructure in areas where traditional banking is impractical, uneconomic due to minimum balances, or simply non-existent. M-Pesa has not yet released a mobile app, but is arguably the most impactful mobile money system and highlights the promise of branchless banking for developing economies.

Mobile money has become ubiquitous and essential. M-Pesa boasts more than 18.2 million registrations in a country of 43.2 million [37]. In Kenya and eight other countries, there are more mobile money accounts than bank accounts. As of August 2014, there were a total of 246 mobile money services in 88 countries serving a total of over 203 million registered accounts, continuing a trend [49] up from 219 services in December 2013. Note that these numbers explicitly exclude services that are simply a mobile interface for existing banking systems. Financial security, and trust in branchless banking systems, depends on the assurances that these systems are secure against fraud and attack. Several of the apps that we study offer strong assurances of security in their promotional materials. Figure 1 provides examples

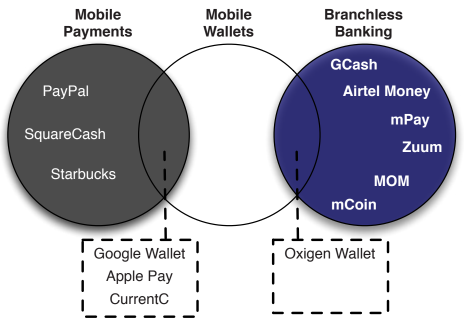


Figure 2: While Mobile Money (Branchless Banking) and Mobile Payments share significant overlapping functionality, the key differences are the communications methods the systems use and that mobile money systems do not rely on existing banking infrastructure.

of these promises. This promise of financial security is even reflected in the M-Pesa advertising slogan "Relax, you have got M-Pesa." [52]. However, the veracity of these claims is unknown.

2.1 Comparison to Other Services

Mobile money is closely related to other technologies. Figure 2 presents a Venn diagram indicating how representative mobile apps fall into the categories of mobile payments, mobile wallets, and branchless banking systems. Most mobile finance systems share the ability to make payments to other individuals or merchants. In our study, the mobile apps for these finance systems are distinguished as follows:

- **Mobile Payment** describes systems that allow a mobile device to make a payment to an individual or merchant using *traditional banking infrastructure*. Example systems include PayPal, Google Wallet, Apple Pay, Softpay (formerly ISIS), CurrentC, and Square Cash. These systems acting as an intermediary for an existing credit card or bank account.

- **Mobile Wallets** store multiple payment credentials for either mobile money or mobile payment systems and/or facilitate promotional offers, discounts, or loyalty programs. Many mobile money systems (like Oxigen Wallet, analyzed in this paper) and mobile payment systems (like Google Wallet and Apple Pay) are also mobile wallets.

- **Branchless Banking** is designed around policies that facilitate easy inclusion. Enrollment often simply requires just a phone number or national ID number be entered into the mobile money system. These systems have no minimum balances and low transaction fees, and feature reduced "Know Your Customer"[1] regulations [51]. Another key feature of branchless banking systems is that in many cases they do not rely on Internet (IP) connectivity exclusively, but also use SMS, Unstructured Supplementary Service Data (USSD), or cellular voice (via Interactive Voice Response, or IVR) to conduct transactions. While methods for protecting data confidentiality and integrity over IP are well established, the channel cryptography used for USSD and SMS has been known to be vulnerable for quite some time [56].

3 App Selection and Analysis

In this section, we discuss how apps were chosen for analysis and how the analysis was conducted.

3.1 Mallodroid Analysis

Using data from the GSMA Mobile Money Tracker [6], we identified a total of 47 Android mobile money apps across 28 countries. We first ran an automated analysis on all 47 of these apps using Mallodroid [28], a static analysis tool for detecting SSL/TLS vulnerabilities, in order to establish a baseline. Table 3 in the appendix provides a comprehensive list of the known Android mobile money applications and their static analysis results. Mallodroid detects vulnerabilities in 24 apps, but its analysis only provides a basic indicator of problematic code; it does not, as we show, exclude dead code or detect major flaws in design. For example, it cannot guarantee that sensitive flows *actually use* SSL/TLS. It similarly cannot detect ecosystem vulnerabilities, including the use of deprecated, vulnerable, or incorrect SSL/TLS configurations on remote servers. Finally, the absence of SSL/TLS does not necessarily condemn an application, as applications can still operate securely using other protocols. Accordingly, such automated analysis provides an incomplete picture at best, and at worst an incorrect one. This is a limitation of all automatic approaches, not just Mallodroid.

In the original Mallodroid paper, its authors performed a manual analysis on 100 of the 1,074 (9.3%) apps their tool detected to verify its findings; however, only 41% of those apps were vulnerable to SSL/TLS man-in-the-middle attacks. It is therefore imperative to further verify the findings of this tool to remove false positives and false negatives.

[1] "Know Your Customer" (KYC), "Anti-Money Laundering" (AML), and "Combating Financing of Terrorism" policies are regulations used throughout the world to frustrate financial crime activity.

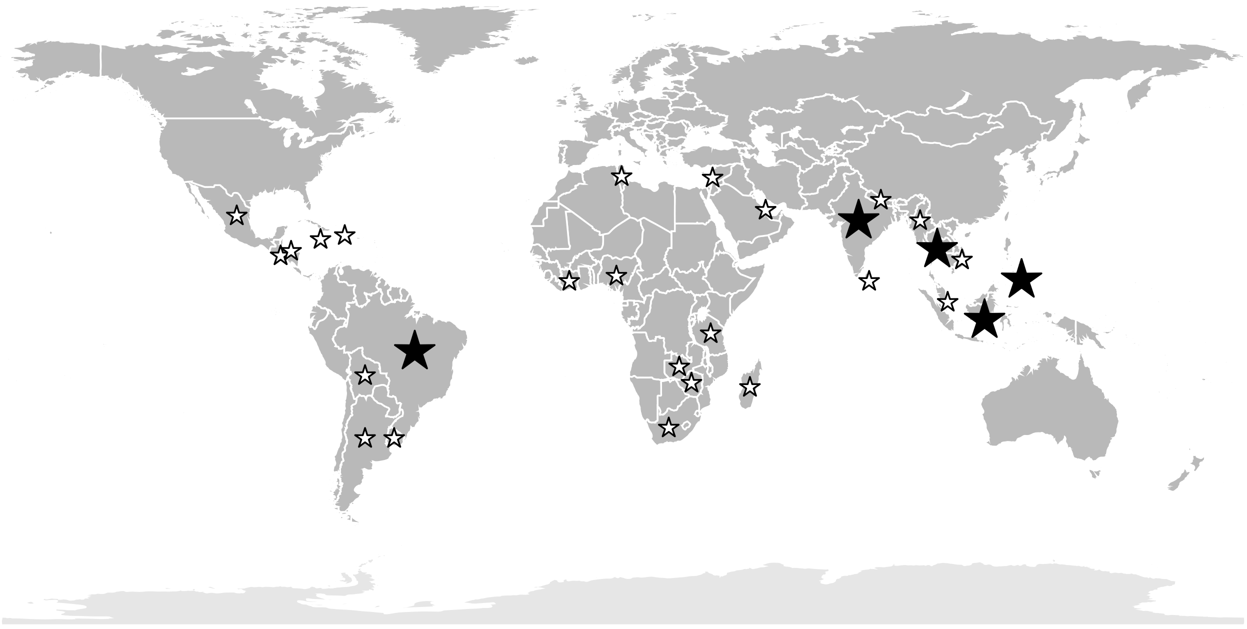


Figure 3: The mobile money applications we analyzed were developed for a diverse range of countries. In total, we performed an initial analysis on applications from 28 countries representing up to approximately 1.2 million users based on market download counts. From this, we selected 7 applications to fully analyze from 5 countries. Each black star represents these countries, and the white stars represent the remainder of the mobile money systems.

3.2 App Selection

Given the above observations, we selected seven mobile money apps for more extensive analysis. These seven apps represent 15% of the total applications and were selected to reflect diversity across markets, providers, features, download counts, and static analysis results. Collectively, these apps serve millions of users. Figure 3 shows the geographic diversity across all of the mobile money apps we observed and those we selected for manual analysis.

We focus on Android applications in this paper because Android has the largest market share worldwide [43], and far more mobile money applications are available for Android than iOS. However, while we cannot make claims about iOS apps that we did not analyze, we do note that all errors disclosed in Section 4 are possible in iOS and are not specific to Android.

3.3 Manual Analysis Process

Our analysis is the first of its kind to perform an in-depth analysis of the protocols used by these applications as well as inspect *both ends* of the SSL/TLS sessions they may use. Each layer of the communication builds on the last; any error in implementation potentially affects the security guarantees of all others. This holistic view of the entire app communication protocol at multiple layers offers a deep view of the fragility of these systems.

In order to accomplish this, our analysis consisted of two phases. The first phase provided an overview of the functionality provided by the app; this included analyzing the application's code and manifest and testing the quality of any SSL/TLS implementations on remote servers. Where possible, we obtained an in-country phone number and created an account for the mobile money system. The overview phase was followed by a reverse engineering phase involving manual analysis of the code. For those apps that we possessed accounts, we also executed the app and verified any findings we found in the code.

Our main interest is in verifying the integrity of these sensitive financial apps. While privacy issues like IMEI or location leakage are concerning [26], we focus on communications between the app and the IP or SMS backend systems, where an adversary can observe, modify, and/or generate transactions.

Phase 1: Inspection

In the inspection phase, we determined the basic functionality and structure of the app in order to guide later analysis. Figure 4 shows the overall toolchain for analyzing the apps along with each respective output.

The first step of the analysis was to obtain the application manifest using `apktool` [2]. We then used an simple script to generate a report identifying each app component (i.e., activities, services, content providers, and

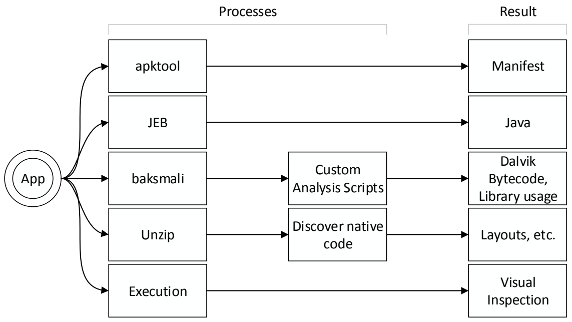


Figure 4: A visualization of the tools used for analyzing the mobile money apps.

broadcast receivers) as well as the permissions declared and defined by the app. This acted as a high-level proxy for understanding the capabilities of the app. With this report, we could note interesting or dangerous permissions (e.g., WRITE_EXTERNAL_STORAGE can leak sensitive information) and which activities are exported to other apps (which can be used to bypass authentication).

The second step of this phase was an automated review of the Dalvik bytecode. We used the `Baksmali` [10] tool to disassemble the application `dex` file. While disassembly provides the Dalvik bytecode, this alone does not assist in reasoning about the protocols, data flows, and behavior of an application. Further inspection is still required to understand the semantic context and interactions of classes and functions.

After obtaining the Dalvik bytecode, we used a script to identify classes that use interesting libraries; these included networking libraries, cryptography libraries (including `java.security` and `javax.crypto` and Bouncy Castle [11]), well-known advertising libraries (as identified by Chen et al. [18]), and libraries that can be used to evade security analysis (like Java ClassLoaders). References to these libraries are found directly in the Dalvik assembly with regular expressions. The third step of the overview was to manually take note of all packages included in the app (external libraries like social media libraries, user interface code, HTTP libraries, etc.).

While analyzing the application's code can provide deep insight into the application's behavior and client/server protocols, it does not provide any indication of the security of the connection as it is negotiated by the server. For example, SSL/TLS servers can offer vulnerable versions of the protocol, weak signature algorithms, and/or expired or invalid certificates. Therefore, the final step of the analysis was to check each application's SSL endpoints using the Qualys SSL Server Test [50].[2] This test provides a comprehensive, non-invasive view of the configuration and capabilities of a server's SSL/TLS implementation.

Phase 2: Reverse Engineering

In order to complete our holistic view of both the application protocols and the client/server SSL/TLS negotiation, we reverse engineered each app in the second phase. For this step, we used the commercial interactive JEB Decompiler [4] to provide Java syntax for most classes. While we primarily used the decompiled output for analysis, we also reviewed the Dalvik assembly to find vulnerabilities. Where we were able to obtain accounts for mobile money accounts, we also confirmed each vulnerability with our accounts when doing so would not negatively impact the service or other users.

Rather than start by identifying interesting methods and classes, we began analysis by following the application lifecycle as the Android framework does, starting with the `Application.onCreate()` method and moving on to the first Activity to execute. From there, we constructed the possible control paths a user can take from the beginning through account registration, login, and money transfer. This approach ensures that our findings are actually present in live code, and accordingly leads to conservative claims about vulnerabilities.[3] After tracing control paths through the `Activity` user interface code, we also analyze other components that appear to have sensitive functionality.

As stated previously, our main interest is in verifying the integrity of these financial applications. In the course of analysis, we look for security errors in the following actions:

- Registration and login
- User authentication after login
- Money transfers

These errors can be classified as:

- Improper authentication procedures
- Message confidentiality and integrity failures (including misuse of cryptography)
- Highly sensitive information leakage (including financial information or authentication credentials)
- Practices that discourage good security hygiene, such as permitting insecure passwords

We discuss our specific findings in Section 4.

3.3.1 Vulnerability Disclosure

As of the publication deadline of this paper we have notified all services of the vulnerabilities. We also included basic details of accepted mitigating practices for each

[2] For security reasons, Qualys does not test application endpoints on non-standard ports or without registered domain names.

[3] In the course of analysis, we found several vulnerabilities in what is apparently dead code. While we disclosed those findings to developers for completeness, we omit them from this paper.

ID	Common Weakness Enumeration	Airtel Money	mPAY	Oxigen Wallet	GCash	Zuum	MOM	mCoin
SSL/TLS & Certificate Verification								
CWE-295	Improper Certificate Validation	✗	✗		✗			✗
Non-standard Cryptography								
CWE-330	Use of Insufficiently Random Values	✗		✗	✗			
CWE-322	Key Exchange without Entity Authentication			✗			✗	
Access Control								
CWE-88	Argument Injection or Modification		✗					
CWE-302	Authentication Bypass by Assumed-Immutable Data			✗	✗			
CWE-521	Weak Password Requirements				✗			
CWE-522	Insufficiently Protected Credentials						✗	
CWE-603	Use of Client-Side Authentication						✗	
CWE-640	Weak Password Recovery Mechanism for Forgotten Password			✗				
Information Leakage								
CWE-200	Information Exposure			✗			✗	✗
CWE-532	Information Exposure Through Log Files		✗		✗		✗	
CWE-312	Cleartext Storage of Sensitive Information		✗		✗			✗
CWE-319	Cleartext Transmission of Sensitive Information			✗	✗		✗	

Table 1: Weaknesses in Mobile Money Applications, indexed to corresponding Common Weakness Enumeration (CWE) records. The CWE database is a comprehensive taxonomy of software vulnerabilities developed by MITRE [55] and provide a common language for software errors.

finding. Most have not sent any response to our disclosures. We have chosen to publicly disclose these vulnerabilities in this paper out of an obligation to inform users of the risks they face in using these insecure services.

4 Results

This section details the results of analyzing the mobile money applications. Overall, we find 28 significant vulnerabilities across seven applications. Table 1 shows these vulnerabilities indexed by CWE and broad categories (apps are ordered by download count). All but one application (Zuum) presents at least one major vulnerability that harmed the confidentiality of user financial information or the integrity of transactions, and most applications have difficulty with the proper use of cryptography in some form.

4.1 Automated Analysis

Our results for SSL/TLS vulnerabilities should mirror the output of an SSL/TLS vulnerability scanner such as Mallodroid. Though two applications were unable to be analyzed by Mallodroid, it detects at least one critical vulnerability in over 50% of the applications it successfully completed.

Mallodroid produces a false positive when it detects an SSL/TLS vulnerability in Zuum, an application that, through manual analysis, we verified was correctly performing certificate validation. The Zuum application *does contain* disabled certificate validation routines, but these are correctly enclosed in logic that checks for development modes.

Conversely, in the case of MoneyOnMobile, Mallodroid produces a false negative. MoneyOnMobile contains no SSL/TLS vulnerability because it does not employ SSL/TLS. While this can be considered correct operation of Mallodroid, it also does not capture the severe information exposure vulnerability in the app.

Overall, we find that Mallodroid, an extremely popular analysis tool for Android apps, does not detect the *correct* use of SSL/TLS in an application. It produces an alert for the most secure app we analyzed and did not for the least. In both cases, manual analysis reveals stark differences between the Mallodroid results and the real security of an app. A comprehensive, correct analysis must include a review of the application's validation and actual use of SSL/TLS sessions as well as *where* these are used in the application (e.g., used for all sensitive communications). Additionally, it is critical to understand whether the remote server enforces secure protocol versions, ciphers, and hashing algorithms. Only a manual analysis provides this holistic view of the communication between application and server so that a complete security evaluation can be made.

4.2 SSL/TLS

As we discussed above, problems with SSL/TLS certificate validation represented the most common vulnerability we found among apps we analyzed. Certificate validation methods inspect a received certificate to ensure that it matches the host in the URL, that it has a trust chain that terminates in a trusted certificate authority, and that it has not been revoked or expired. However, developers are able to disable this validation by creating a new class that implements the `X509TrustManager` interface using arbitrary validation methods, replacing the validation implemented in the parent library. In the applications that override the default code, the routines were empty; that is, they *do nothing* and do not throw excep-

Product	Qualys Score	Most Noteworthy Vulnerabilities
Airtel Money	A-	Weak signature algorithm (SHA1withRSA)
mPAY 1	F	SSL2 support, Insecure Client-Initiated Renegot.
mPAY 2	F	Vulnerable to POODLE attack
Oxigen Wallet	F	SSL2 support, MD5 cipher suite
Zuum	A-	Weak signature algorithm (SHA1withRSA)
GCash	C	Vulnerable to POODLE attack
mCoin	N/A	Uses expired, `localhost` self-signed certificate
MoneyOnMobile	N/A	App does not use SSL/TLS

Table 2: Qualys reports for domains associated with branchless banking apps. “Most Noteworthy Vulnerabilities” lists what Qualys considers to be the most dangerous elements of the server’s configuration. mPAY contacts two domains over SSL, both of which are separately tabulated below. Qualys would not scan mCoin because it connects to a specific IP address, not a domain.

tions on invalid certificates. This insecure practice was previously identified by Georgiev et al. [31] and is specifically targeted by Mallodroid.

Analyzing only the app does not provide complete visibility to the overall security state of an SSL/TLS session. Server misconfiguration can introduce additional vulnerabilities, even when the client application uses correctly implemented SSL/TLS. To account for this, we also ran the Qualys SSL Server Test [50] on each of the HTTPS endpoints we discovered while analyzing the apps. This service tests a number of properties of each server to identify configuration and implementation errors and provide a “grade” for the configuration. These results are presented in Table 2. Three of the endpoints we tested received failing scores due to insecure implementations of SSL/TLS. To underscore the severity of these misconfigurations, we have included the “Most Noteworthy Vulnerabilities” identified by Qualys.

mCoin. Coupling the manual analysis with the Qualys results, we found that in one case, the disabled validation routines were required for the application to function correctly. The mCoin API server provides a certificate that is issued to “localhost” (an invalid hostname for an external service), is expired, and is self-signed (has no trust chain). *No correct certificate validation routine would accept this certificate*. Therefore, without this routine, the mCoin application would be unable to establish a connection to its server. Although Mallodroid detected the disabled validation routines, only our full analysis can detect the relationship between the app’s behavior and the server’s configuration.

The implications of poor validation practices are severe, especially in these critical financial applications. Adversaries can intercept this traffic and sniff cleartext personal or financial information. Furthermore, without additional message integrity checking inside these weak SSL/TLS sessions, a man-in-the-middle adversary is free to manipulate the inside messages.

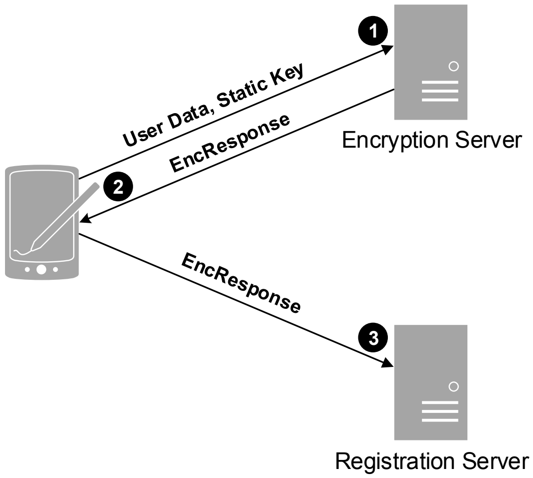


Figure 5: The user registration flow of MoneyOnMobile. All communication is over HTTP.

4.3 Non-Standard Cryptography

Despite the pervasive insecure implementations of SSL/TLS, the client/server protocols that these apps implement are similarly critical to their overall security. We found that four applications used their own custom cryptographic systems or had poor implementations of well-known systems in their protocols. Unfortunately, these practices are easily compromised and severely limit the integrity and privacy guarantees of the software, giving rise to the threat of forged transactions and loss of transaction privacy.

MoneyOnMobile. MoneyOnMobile does not use SSL/TLS. All API calls from the app use HTTP. In fact, we found only one use of cryptography in the application’s network calls. During the user registration process, the app first calls an encryption proxy web service, then sends the service’s response to a registration web service. The call to the encryption server includes both the user data and a fixed static key. A visualization of this protocol is shown in Figure 5.

The encryption server is accessed over the Internet via HTTP, exposing both the user and key data. Because this data is exposed during the initial call, its subsequent encryption and delivery to the registration service provides no security. We found no other uses of this or any other encryption in the MoneyOnMobile app; all other API calls are provided unobfuscated user data as input.

Oxigen Wallet. Like MoneyOnMobile, Oxigen Wallet does not use SSL/TLS. Oxigen Wallet’s registration messages are instead encrypted using the Blowfish algorithm, a strong block cipher. However, a long, random key is not generated for input into Blowfish. Instead, only 17 bits of the key are random. The remaining bits are filled by the mobile phone number, the date, and padding with 0s. The random bits are generated by the Random [34] random number generator. The standard Java documentation [44] *explicitly warns* in its documentation that Random is not sufficiently random for cryptographic key generation.[4] As a result, any attacker can read, modify, or spoof messages. These messages contain demographic information including first and last name, email address, date of birth, and mobile phone number, which constitutes a privacy concern for Oxigen Wallet's users.

After key generation, Oxigen Wallet transmits the key in plaintext along with the message to the server. In other words, every encrypted registration message *includes the key in plaintext*. Naturally, this voids every guarantee of the block cipher. In fact, any attacker who can listen to messages can decrypt and modify them with only a few lines of code.

The remainder of client-server interactions use an RSA public key to send messages to the server. To establish an RSA key for the server, Oxigen Wallet sends a simple HTTP request to receive an RSA key from the Oxigen Wallet server. This message is unauthenticated, which prevents the application from knowing that the received key is from Oxigen Wallet and not from an attacker. Thus, an attacker can pretend to be Oxigen Wallet and send an alternate key to the app. This would allow the attacker to read all messages sent by the client (including those containing passwords) and forward the messages to Oxigen Wallet (with or without modifications) if desired. This RSA man-in-the-middle attack is severe and puts all transactions by a user at risk. At the very least, this will allow an attacker to steal the password from messages. The password can later be used to conduct illicit transactions from the victim's account.

Finally, responses from the Oxigen Wallet servers are not encrypted. This means that any sensitive information that might be contained in a response (e.g., the name of a transaction recipient) can be read by any eavesdropper. This is both a privacy and integrity concern because an attacker could read and modify responses.

GCash. Unlike Oxigen Wallet, GCash uses a static key for encrypting communcations with the remote server. The GCash application package includes a file "enc.key," which contains a symmetric key. During the GCash login process, the user's PIN and session ID are encrypted using this key before being sent to the GCash servers. This key is posted publicly because it is included with every download of GCash. An attacker with this key can decrypt the user's PIN and session ID if the encrypted data is captured. This can subsequently give the attacker the ability to impersonate the user.

The session ID described above is generated during the login process and passed to the server to provide session authentication in subsequent messages. We did not find any other authenticator passed in the message body to the GCash servers after login. The session ID is created using a combination of the device ID, e.g., International Mobile Station Equipment Identity (IMEI), and the device's current date and time. Android will provide this device ID to any application with the READ_PHONE_STATE permission, and device IDs can be spoofed on rooted phones. Additionally, IMEI is frequently abused by mobile apps for persistent tracking of users [25], and is thus also stored in the databases of hundreds of products.

Although the session ID is not a cryptographic construct, the randomness properties required by a strong session ID match those needed by a strong cryptographic key. This lack of randomness results in predictable session IDs can then be used to perform any task as the session's associated user.

Airtel Money. Airtel Money performs a similar mistake while authenticating the user. When launching the application, the client first sends the device's phone number to check if there is an existing Airtel Money account. If so, the server sends back the user's account number in its response. Although this response is transmitted via HTTPS, the app does not validate certificates, creating a compound vulnerability where this information can be discovered by an attacker.

Sensitive operations are secured by the user's 4-digit PIN. The PIN is encrypted in transit using a weakly-constructed key that concatenates the device's phone number and account number in the following format:

$$Key_{enc} = \texttt{j7zgy1yv} \,\|\, phone\# \,\|\, account\# \qquad (1)$$

The prefixed text in the key is immutable and included with the application. Due to the weak SSL/TLS implementation during the initial messages, an adversary can obtain the user's account number and decrypt the PIN. The lack of randomness in this key again produces a vulnerability that can lead to user impersonation.

4.4 Access Control

A number of the applications that we analyzed used access control mechanisms that were poorly implemented or relied on incorrect or unverifiable assumptions that the user's device and its cellular communications channels are uncompromised. Multiple applications relied on SMS communications, but this channel is subject to a number of points of interception [56]. For example, another application on the device with the RECEIVE_SMS

[4] Although the Android offers a SecureRandom class for cryptographically-secure generation, it does not mention its necessity in the documentation.

permission could read the incoming SMS messages of the mobile money application. This functionality is outside the control of the mobile money application. Additionally, an attacker could have physical access to an unlocked phone, where messages can be inspected directly by a person. This channel does not, therefore, provide strong confidentiality or integrity guarantees.

MoneyOnMobile. The MoneyOnMobile app presents the most severe lack of access control we found among the apps we analyzed. The service uses two different PINs, the MPIN and TPIN, to authenticate the user for general functionality and transactions. However, we found that these PINs only prevent the user from moving between Android activities. In fact, the user's PINs are not required to execute any sensitive functionality via the backend APIs. All sensitive API calls (e.g., balance inquiry, mobile recharge, bill pay, etc.) except PIN changes can be executed with *only knowledge of the user's mobile phone number and two API calls.* MoneyOnMobile deploys no session identifiers, cookies, or other stateful tracking mechanisms during the app's execution; therefore, none of these are required to exploit the service.

The first required API call takes the mobile number as input and outputs various parameters of the account (e.g., Customer ID). These parameters identify the account as input in the subsequent API call. Due to the lack of any authentication on these sensitive functions, an adversary with no knowledge of the user's account can execute transactions on the user's behalf. Since the initial call provides information about a user account, this call allows an adversary to brute force phone numbers in order to find MoneyOnMobile users. This call also provides the remainder of the information needed to perform transactions on the account, severely compromising the security of the service.

mPAY. While the MoneyOnMobile servers do not require authentication before performing server tasks, we found the opposite is true with mPAY. The mPAY app accepts and performs unauthenticated commands from its server. The mPAY app uses a web/native app hybrid that allows the server to send commands to the app through the use of a URL parameter "method." These methods instruct the app to perform many actions, including starting the camera, opening the browser to an arbitrary URL, or starting an arbitrary app. If the control flow of the web application from the server side is secure, and the HTTP channel between client and server is free from injection or tampering, it is unlikely that these methods could be harmful. However, if an attacker can modify server code or redirect the URL, this functionality could be used to attack mobile users. Potential attacks include tricking users into downloading malware, providing information to a phishing website, or falling victim to a cross-site request forgery (CSRF) attack. As we discussed in the previous results, mPAY does not correctly validate the certificates used for its SSL/TLS sessions, and so these scenarios are unsettlingly plausible.

GCash. Although GCash implements authentication, it relies on easily-spoofable identity information to secure its accounts. During GCash's user registration process, the user selects a PIN for future authentication. The selected PIN is sent in plaintext over SMS along with the user's name and address. GCash then identifies the user with the phone number used to send the SMS message. This ties the user's account to their phone's SIM card. Unfortunately, SMS spoofing services are common, and these services provide the ability for an unskilled adversary to send messages appearing to be from an arbitrary number [27]. SIM cards can be damaged, lost, or stolen, and since the wallet balance is tied to this SIM, it may be difficult for a user to reclaim their funds.

Additionally, GCash requires the user to select a 4-digit PIN to register an account. As previously mentioned, this PIN is used to authenticate the user to the service. This allows only 10,000 possible combinations of PINs, which is quickly brute-forceable, though more intelligent guessing can be performed using data on the frequency of PIN selection [16]. We were not able to create an account with GCash to determine if the service locks accounts after a number of incorrect login attempts, which is a partial mitigation for this problem.

Oxigen Wallet. Like GCash, Oxigen Wallet also allows users to perform several sensitive actions via SMS. The most severe of these is requesting a new password. As a result, any attacker or application with access to a mobile phone's SMS subsystem can reset the password. That password can be used to login to the app or to send SMS messages to Oxigen Wallet for illicit transactions.

4.5 Information Leakage

Several of the analyzed applications exposed personally-identifying user information and/or data critical to the transactional integrity through various methods, including logging and preference storage.

4.5.1 Logging

The Android logging facility provides developers the ability to write messages to understand the state of their application at various points of its execution. These messages are written to the device's internal storage so they can be viewed at a future time. If the log messages were visible only to developers, this would not present the opportunity for a vulnerability. However, prior to Android 4.1, any application can declare the READ_LOGS permission and read the log files of any other application. That is, any arbitrary application (including malicious ones) may read the logs. According to statistics from Google [32], 20.7% of devices run a version of Android that allows other apps to read logs.

mPAY. mPAY logs include user credentials, personal identifiers, and card numbers.

GCash. GCash writes the plaintext PIN using the verbose logging facility. The Android developer documentation states that verbose logging should not be compiled into production applications [33]. Although GCash has a specific `devLog` function that only writes this data when a debug flag is enabled, there are still statements without this check. Additionally, the session ID is also logged using the native Android logging facility without checking for a developer debug flag. An attacker with GCash log access can identify the user's PIN and the device ID, which could be used to impersonate the user.

MoneyOnMobile. These logs include server responses and account balances.

4.5.2 Preference Storage

Android provides a separate mechanism for storing preferences. This system has the capability of writing the stored preferences to the device's local storage, where they can be recovered by inspecting the contents of the preferences file. Often, developers store preferences data in order to access it across application launches or from different sections of the code without needing to explicitly pass it. While the shared preferences are normally protected from the user and other apps, if the device is rooted (either by the user or a malicious application) the shared preferences file can be read.

GCash. GCash stores the user's PIN in this system. The application clears these preferences in several locations in the code (e.g., logout, expired sessions), however if the application terminates unexpectedly, these routines may not be called, leaving this sensitive information on the device.

mPAY. Similarly, mPAY stores the mobile phone number and customer ID in its preferences.

mCoin. Additionally, mCoin stores the user's name, birthday, and certain financial information such as the user's balance. We also found that mCoin exposes this data in transmission. Debugging code in the mCoin application is also configured to forward the user's mCoin shared preferences to the server with a debug report. As noted above, this may contain the user's personal information. This communication is performed over HTTP and sent in plaintext, providing no confidentiality for the user's data in transmission.

4.5.3 Other Leakage

Oxigen Wallet. We discussed in Section 4.3 that requests from the Oxigen Wallet client are encrypted (insecurely) with either RSA or Blowfish. Oxigen Wallet also discloses mobile numbers of account holders. On sign up, Oxigen Wallet sends a `GetProfile` request to a server to determine if the mobile number requesting a new account is already associated with an email address. The client sends an email address, and the server sends a full mobile number back to the client. The application does appear to understand the security need for this data as only the last few digits of the mobile number are shown on the screen (the remaining digits are replaced by `X`s). However, it appears that the full mobile number is provided in the network message. This means that if an attacker could somehow read the full message, he could learn the mobile number associated with the email address.

Unfortunately, the `GetProfile` request can be sent using the Blowfish encryption method previously described, meaning that an attacker could write his own code to poll the Oxigen Wallet servers to get mobile numbers associated with known email addresses. This enumeration could be used against a few targets or it may be done in bulk as a precursor to SMS spam, SMS phishing, or voice phishing. This bulk enumeration may also tax the Oxigen Wallet servers and degrade service for legitimate users. This attack would not be difficult for an attacker with even rudimentary programming ability.

4.6 Zuum

Zuum is a Brazilian mobile money application built by Mobile Financial Services, a partnership between Telefonica and MasterCard. While many of the other apps we analyzed were developed solely by cellular network providers or third-party development companies, MasterCard is an established company with experience building these types of applications.

This app is particularly notable because we did not find in Zuum the major vulnerabilities present in the other apps. In particular, the application uses SSL/TLS sessions with certificate validation enabled and includes a public key and performs standard cryptographic operations to protect transactions inside the session. Mallodroid detects Zuum's disabled certificate validation routines, but our manual analysis determines that these routines would not run. We discuss MasterCard's involvment in the Payment Card Industry standards, the app's terms of service, and the ramifications of compromise in Section 5.

4.7 Verification

We obtained accounts for MoneyOnMobile, Oxigen Wallet, and Airtel Money in India. For each app, we configured an Android emulator instance to forward its traffic through a man-in-the-middle proxy. In order to remain as passive as possible, we did not attempt to verify any transaction functionality (e.g., adding money to the account, sending or receiving money, paying bills, etc.). We were able to successfully verify every vulnerability that we identified for these apps.

5 Discussion

In this discussion section, we make observations about authentication practices and our SSL/TLS findings, regulations governing these apps, and whether smartphone applications are in fact safer than the legacy apps they replace.

Why do these apps use weak authentication? Numeric PINs were the authentication method of choice for the majority of the apps studied — only three apps allow use of a traditional password. This reliance on PINs is likely a holdover from earlier mobile money systems developed for feature phones. While such PINs are known to be weak against brute force attacks, they are chosen for SMS or USSD systems for two usability reasons. First, they are easily input on limited phone interfaces. Second, short numeric PINs remain usable for users who may have limited literacy (especially in Latin alphabets). Such users are far more common in developing countries, and prior research on secure passwords has assumed user literacy [54]. Creating a distinct strong password for the app may be confusing and limit user acceptability of new apps, despite the clear security benefits.

Beyond static PINs, Airtel Money and Oxigen Wallet (both based in India) use SMS-provided one-time passwords to authenticate users. While effective at preventing remote brute-force attacks, this step provides no defense against the other attacks we describe in the previous section.

Why do these apps fail to validate certificates? While this work and prior works have shown that many Android apps fail to properly validate SSL/TLS certificates [28], the high number of branchless banking apps that fail to validate certificates is still surprising, especially given the mission of these apps. Georgiev et al. found that many applications improperly validate certificates, yet identify the root cause as poorly designed APIs that make it easy to make a validation mistake [31]. One possible explanation is that certificate validation was disabled for a test environment which had no valid certificate. When the app was deployed, developers did not test for improper validation and did not remove the test code that disabled host name validation. Fahl et al. found this explanation to be common in developer interviews [29], and they also further explore other reasons for SSL/TLS vulnerabilities, including developer misunderstandings about the purpose of certificate validation.

In the absence of improved certificate management practices at the application layer, one possible defense is to enforce sane SSL/TLS configurations at the operating system layer. This capability is demonstrated by Fahl et al. for Android [29], while Bates et al. present a mechanism for Linux that simultaneously facilitates the use of SSL trust enhancements [15]. In the event that the system trusts compromised root certificates, a solution like DVCert [23] could be used to protect against man in the middle attacks.

Are legacy systems more secure? In Section 7, we noted that prior work had found that legacy systems are fundamentally insecure as they rely principally on insecure GSM bearer channels. Those systems rely on bearer channel security because of the practical difficulties of developing and deploying secure applications to a plethora of feature phone platforms with widely varying designs and computational capabilities. In contrast, we look at apps developed for relatively homogenous, well-resourced smartphones. One would expect that the advanced capabilities available on the Android platform would increase the security of branchless banking apps. However, given the vulnerabilities we disclose, the branchless banking apps we studied for Android put users at a *greater* risk than legacy systems. Attacking cellular network protocols, while shown to be practical [56], still has a significant barrier to entry in terms of equipment and expertise. In contrast, the attacks we disclose in this paper require only a laptop, common attack tools, and some basic security experience to discover and exploit. Effectively, these attacks are easier to exploit than the previously disclosed attacks against SMS and USSD interfaces.

Does regulation help? In the United States, the PCI Security Standards Council releases a Data Security Standard (PCI DSS) [48], which govern the security requirements for entities that handle cardholder data (e.g., card numbers and expiration dates). The council is a consortium of card issuers including Visa, MasterCard, and others that cooperatively develop this standard. Merchants that accept credit card payments from these issuers are generally required to adhere to the PCI DSS and are subject to auditing.

The DSS document includes requirements, testing procedures, and guidance for securing devices and networks that handle cardholder data. These are not, however, specific enough to include detailed implementation instructions. The effectiveness of these standards is not our main focus; we note that the PCI DSS can be used as a checklist-style document for ensuring well-rounded security implementations.

In 2008, the Reserve Bank of India (RBI) issued guidelines for mobile payment systems [13]. (By their definition, the apps we study would be included in these guidelines). In 12 short pages, they touch on aspects as broad as currencies allowed, KYC/AML policies, interbank settlement policies, corporate governance approval, legal jurisdiction, consumer protection, and technology and security standards for a myriad of delivery channels. The security standards give implementers wide leeway to use their best judgement about specific security practices. MoneyOnMobile, which had the most severe security issues among all of the apps we manually analyzed, prominently displays its RBI authorization on its web site.

Some prescriptions stand out from the rest: an objective to have "digital certificate based inquiry/transaction capabilities," a recommendation to have a mobile PIN that is encrypted on the wire and never stored in cleartext, and use of the mobile phone number as the chief identifier. These recommendations may be responsible for certain design decisions of Airtel Money and Oxigen Wallet (both based in India). For example, the digital certificate recommendation may have driven Oxigen Wallet developers to develop their (very flawed) public key encryption architecture. These recommendations also explain why Airtel Money elected to further encrypt the PIN (and only the PIN) in messages that are encapsulated by TLS. Further, the lack of guidance on what "strong encryption" entails may be partially responsible for the security failures of Airtel Money and Oxigen Wallet. Finally, we note that we believe that Airtel Money, while still vulnerable, was within the letter of the recommendations.

To our knowledge, other mobile money systems studied in this paper are not subject to such industry or government regulation. While a high-quality, auditable industry standard may lead to improved branchless banking security, it is not clear that guidelines like RBI's currently make much of a difference.

6 Terms of Service & Consumer Liability

After uncovering technical vulnerabilities for branchless banking, we investigated their potential implications for fraud liability. In the United States, the consumer is not held liable for fraudulent transactions beyond a small amount. This model recognizes that users are vulnerable to fraud that they are powerless to prevent, combat, or detect prior to incurring losses.

To determine the model used for the branchless banking apps we studied, we surveyed the Terms of Service (ToS) for each of the seven analyzed apps analyzed. The Airtel Money [1], GCash [3], mCoin [5], Oxigen Wallet [9], MoneyOnMobile [7], and Zuum [12] terms all hold the customer solely responsible for most forms of fraudulent activity. Each of these services hold the customer responsible for the safety and security of their password. GCash, mCoin, and Oxigen Wallet also hold the customer responsible for protecting their SIM (i.e., mobile phone). GCash provides a complaint system, provided that the customer notifies GCash in writing within 15 days of the disputed transaction. However, they also make it clear that erroneous transactions are not grounds for dispute. mPAY's terms [8] are less clear on the subject of liability; they provide a dispute resolution system, but do not detail the circumstances for which the customer is responsible. Across the body of these terms of service, it is overwhelmingly clear that the customer is responsible for all transactions conducted with their PIN/password on their mobile device.

The presumption of customer fault for transactions is at odds with the findings of this work. The basis for these arguments appear to be that, if a customer protects their PIN and protects their physical device, there is no way for a third party to initiate a fraudulent transaction. We have demonstrated that this is not the case. Passwords can be easily recovered by an attacker. Six of the seven apps we manually analyzed transmits authentication data over insecure connections, allowing them to be recovered in transit. Additionally, with only brief access to a customer's phone, an attacker could read GCash PINs out of the phone logs or trigger the Oxigen Wallet password recovery mechanism. Even when the mobile device and SIM card are fully under customer control, unauthorized transactions can still occur, due to the pervasive vulnerabilities found in these six apps. By launching a man-in-the-middle attack, an adversary would be able to tamper with transactions while in transit, misleading the provider into believing that a fraudulent transaction originated from a legitimate user. *These attacks are all highly plausible.* Exploits of the identified vulnerabilities are not probabilistic, but would be 100% effective. With only minimal technical capability, an adversary could launch these attacks given the ability to control a local wireless access point. This litany of vulnerabilities comes only from an analysis of client-side code. Table 2 hints that there may be further server side configuration issues, to say nothing of the security of custom server software, system software, or the operating systems used.

Similar to past findings for the "Chip & Pin" credit card system [40], it is possible that these apps are already being exploited in the wild, leaving consumers with no recourse to dispute fraudulent transactions. Based on the discovery of rampant vulnerabilities in these applications, we feel that the liability model for branchless banking applications must be revisited. Providers must not marry such vulnerable systems with a liability model that refuses to take responsibility for the technical flaws, and these realities could prevent sustained growth of branchless banking systems due to the high likelihood of fraud.

7 Related Work

Banking has been a motivation for computer security since the origins of the field. The original Data Encryption Standard was designed to meet the needs of banking and commerce, and Anderson's classic paper "Why Cryptosystems Fail" looked specifically at banking security [14]. Accordingly, mobile money systems have been scrutinized by computer security practitioners. Current research on mobile money systems to-date has focused on the challenges of authentication, channel security, and transaction verification in legacy systems designed for feature phones. Some prior work has provided threat modeling and discussion of broader system-wide security issues. To our knowledge, we are the first to examine the security of smartphone applications used by mobile money systems.

Mobile money systems rely on the network to provide identity services; in essence, identity is the telephone number (MS-ISDN) of the subscriber. To address physical access granting attackers access to accounts, researchers have investigated the use of a small one-time pads as authenticators in place of PINs. Panjwani et al. [47] present a new scheme that avoids vulnerabilities with using one-time passwords with PINs and SMS. Sharma et al. propose using scratch-off one-time authenticators for access with supplemental recorded voice confirmations [53]. These schemes add complexity to the system while only masking the PIN from an adversary who can see a message. These schemes do not provide any guarantees against an adversary who can modify messages or who recovers a message and a pad.

SMS-based systems, in particular, are vulnerable to eavesdropping or message tampering [42], and so have seen several projects to bring additional cryptographic mechanisms to mobile money systems [20, 41, 22]. Systems that use USSD, rather than SMS, as their bearer channel can also use code executing on the SIM card to cryptographically protect messages. However, it is unknown how these protocols are implemented or what guarantees they provide [45].

Finally, several authors have written papers investigating the holistic security of mobile money systems designed exclusively for "dumbphones." Paik et al. [45] note concerns about reliance on GSM traffic channel cryptographic guarantees, including the ability to intercept, replay, and spoof the source of SMS messages. Panjwani fulfills the goals laid out by Paik et al. by providing a brief threat model and a design to protect against the threats they identify [46]. While those papers focus on technical analysis, de Almeida [38] and Harris et al. [35] note the policy implications of the insecurity of mobile money.

While focused strictly on mobile money platforms, this paper also contributes to the literature of Android application security measurement. The pioneering work in this space was TaintDroid [25, 25], a dynamic analysis system that detected private information leakages. Shortly after, Felt et al. found that one-third of apps studied held privileges they did not need [30], while Chin et al. found that 60% of apps manually examined were vulnerable to attacks involving Android Intents [19]. More recently, Fahl et al. [28] and Egele et al. [24] use automated static analysis investigated cryptographic API use in Android, finding respectively that 8% of apps studied were vulnerable to man-in-the-middle attacks and that 88% of apps make some mistake with regards to cryptographic libraries [24]. Our work confirms these results apply to mobile money applications. This project is most similar to the work of Enck et al. [26], who automatically and manually analyzed 1,100 applications for a broad range of security concerns.

However, prior work does not investigate the security guarantees and the severe consequences of smart phone application compromise in branchless banking systems. Our work specifically investigates this open area of research and provides the world's first detailed security analysis of mobile money apps. In doing so, we demonstrate the risk to users who rely on these systems for financial security.

8 Conclusions

Branchless banking applications have and continue to hold the promise to improve the standard of living for many in the developing world. By enabling access to a cashless payment infrastructure, these systems allow residents of such countries to reap the benefits afforded to modern economies and decrease the physical security risks associated with cash transactions. However, the security of the applications providing these services has not previously been vetted in a comprehensive or public fashion. In this paper, we perform precisely such an analysis on seven branchless banking applications, balancing both popularity with geographic representation. Our analysis targets the registration, login, and transaction portions of the representative applications, and codifies discovered vulnerabilities using the CWE classification system. We find significant vulnerabilities in six of the seven applications, which prevent both users and providers from reasoning about the integrity of transactions. We then pair these technical findings with the discovery of fraud liability models that explicitly hold the end user culpable for all fraud. Given the systemic problems we identify, we argue that dramatic improvements to the security of branchless banking applications are imperative to protect the mission of these systems.

Acknowledgments

The authors are grateful to Saili Sahasrabudde for her assistance with this work. We would also like to thank the members of the SENSEI Center at the University of Florida for their help in preparing this work, as well as our anonymous reviewers for their helpful comments.

This work was supported in part by the US National Science Foundation under grant numbers CNS-1526718, CNS-1540217, and CNS-1464087. 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] Airtel Money: Terms and Conditions of Usage. `http://www.airtel.in/personal/money/terms-of-use`.

[2] android-apktool: A Tool for Reverse Engineering Android APK Files. `https://code.google.com/p/android-apktool/`.

[3] GCash Terms and Conditions. `http://www.globe.com.ph/gcash-terms-and-conditions`.

[4] JEB Decompiler. `http://www.android-decompiler.com/`.

[5] mCoin: Terms and Conditions. `http://www.mcoin.co.id/syarat-dan-ketentuan`.

[6] MMU Deployment Tracker. `http://www.gsma.com/mobilefordevelopment/programmes/mobile-money-for-the-unbanked/insights/tracker`.

[7] Money on Mobile Sign-Up: Terms and Conditions. `http://www.money-on-mobile.com`.

[8] mPAY: Terms and Conditions. `http://www.ais.co.th/mpay/en/about-term-condition.aspx`.

[9] Oxigen Wallet: Terms and Conditions. `https://www.oxigenwallet.com/terms-conditions`.

[10] smali: An assembler/disassembler for Android's dex format. `https://code.google.com/p/smali/`.

[11] The Legion of the Bouncy Castle. `https://www.bouncycastle.org/`.

[12] Zuum: Termos e Condições. `http://www.zuum.com.br/institucional/termos`.

[13] Mobile Payment in India — Operative Guidelines for Banks. Technical report, Reserve Bank of India, 2008.

[14] R. Anderson. Why Cryptosystems Fail. In *Proc. of the 1st ACM Conf. on Comp. and Comm. Security*, pages 215–227. ACM Press, 1993.

[15] A. Bates, J. Pletcher, T. Nichols, B. Hollembaek, D. Tian, A. Alkhelaifi, and K. Butler. Securing SSL Certificate Verification through Dynamic Linking. In *Proc. of the 21st ACM Conf. on Comp. and Comm. Security (CCS'14)*, Scottsdale, AZ, USA, Nov. 2014.

[16] N. Berry. PIN analysis. `http://www.datagenetics.com/blog/september32012/`, Sept. 2012.

[17] Bill & Melinda Gates Foundation. Financial Services for the Poor: Strategy Overview. `http://www.gatesfoundation.org/What-We-Do/Global-Development/Financial-Services-for-the-Poor`.

[18] K. Chen, P. Liu, and Y. Zhang. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets. In *Proc. 36th Intl. Conf. Software Engineering*, ICSE 2014, pages 175–186, New York, NY, USA, 2014. ACM.

[19] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing Inter-application Communication in Android. In *Proc. 9th Intl. Conf. Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[20] M. K. Chong. *Usable Authentication for Mobile Banking*. PhD thesis, Univ. of Cape Town, Jan. 2009.

[21] P. Chuhan-Pole and M. Angwafo. Mobile Payments Go Viral: M-PESA in Kenya. In *Yes, Africa Can: Success Stories from a Dynamic Continent*. World Bank Publications, June 2011.

[22] S. Cobourne, K. Mayes, and K. Markantonakis. Using the Smart Card Web Server in Secure Branchless Banking. In *Network and System Security*, number 7873 in Lecture Notes in Computer Science, pages 250–263. Springer Berlin Heidelberg, Jan. 2013.

[23] I. Dacosta, M. Ahamad, and P. Traynor. Trust no one else: Detecting MITM attacks against SSL/TLS without third-parties. In *Proceedings of the European Symposium on Research in Computer Security*, pages 199–216. Springer, 2012.

[24] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proc. 20th ACM Conf. Comp. and Comm. Security*, CCS '13, pages 73–84, New York, NY, USA, 2013. ACM.

[25] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Trans. Comput. Syst.*, 32(2):5:1–5:29, June 2014.

[26] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *Proc. 20th USENIX Security Sym.*, San Francisco, CA, USA, 2011.

[27] W. Enck, P. Traynor, P. McDaniel, and T. La Porta. Exploiting open functionality in SMS-capable cellular networks. In *Proc. of the 12th ACM conference on Comp. and communications security*, pages 393–404. ACM, 2005.

[28] S. Fahl, M. Harbach, T. Muders, L. Baumgartner, B. Freisleben, and M. Smith. Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. In *Proc. 2012 ACM Conf. Comp. and Comm. Security*, CCS '12, pages 50–61, New York, NY, USA, 2012. ACM.

[29] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith. Rethinking SSL Development in an Appified World. In *Proc. 20th ACM Conf. Comp. and Comm. Security*, CCS '13, pages 49–60, New York, NY, USA, 2013. ACM.

[30] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *Proc. 18th ACM Conf. Comp. and Comm. Security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[31] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The Most Dangerous Code in the World: Validating SSL Certificates in Non-browser Software. In *Proc. 2012 ACM Conf. Comp. and Comm. Security*, CCS '12, pages 38–49, New York, NY, USA, 2012. ACM.

[32] Google. Dashboards — Android Developers. `https://developer.android.com/about/dashboards/index.html`.

[33] Google. Log — Android Developers. `https://developer.android.com/reference/android/util/Log.html`.

[34] Google. Random — Android Developers. `https://developer.android.com/reference/java/util/Random.html`.

[35] A. Harris, S. Goodman, and P. Traynor. Privacy and Security Concerns Assocaited with Mobile Money Applications in Africa. *Washington Journal of Law, Technology & Arts*, 8(3), 2013.

[36] V. Highfield. More than 60 Per Cent of Kenyan GDP Came From Mobile Money in June 2012, a New Survey Shows. `http://www.totalpayments.org/2013/03/01/60-cent-kenyan-gdp-mobile-money-june-2012-survey-shows/`, 2012.

[37] J. Kamana. M-PESA: How Kenya Took the Lead in Mobile Money. `http://www.mobiletransaction.org/m-pesa-kenya-the-lead-in-mobile-money/`, Apr. 2014.

[38] G. Martins de Almeida. M-Payments in Brazil: Notes on How a Countrys Background May Determine Timing and Design of a Regulatory Model. *Washington Journal of Law, Technology & Arts*, 8(3), 2013.

[39] C. Mims. 31% of Kenya's GDP is Spent Through Mobile Phones. `http://qz.com/57504/31-of-kenyas-gdp-is-spent-through-mobile-phones/`, Feb. 2013.

[40] S. Murdoch, S. Drimer, R. Anderson, and M. Bond. Chip and PIN is Broken. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 433–446, May 2010.

[41] B. W. Nyamtiga, A. Sam, and L. S. Laizer. Enhanced security model for mobile banking systems in Tanzania. *Intl. Jour. Tech. Enhancements and Emerging Engineering Research*, 1(4):4–20, 2013.

[42] B. W. Nyamtiga and L. S. Sam, Anael Laizer. Security perspectives for USSD versus SMS in conducting mobile transactions: A case study of Tanzania. *Intl. Jour. Tech. Enhancements and Emerging Engineering Research*, 1(3):38–43, 2013.

[43] J. Ong. Android Achieved 85% Smartphone Market Share in Q2. `http://thenextweb.com/google/2014/07/31/android-reached-record-85-smartphone-market-share-q2-2014-report/`, July 2014.

[44] Oracle. Random - Java Platform SE 7. `https://docs.oracle.com/javase/7/docs/api/java/util/Random.html`.

[45] M. Paik. Stragglers of the Herd Get Eaten: Security Concerns for GSM Mobile Banking Applications. In *Proc. 11th Workshop on Mobile Comp. Syst. and Appl.*, HotMobile '10, pages 54–59, New York, NY, USA, 2010. ACM.

[46] S. Panjwani. Towards End-to-End Security in Branchless Banking. In *Proc. 12th Workshop on Mobile Comp. Syst. and Appl.*, HotMobile '11, pages 28–33, New York, NY, USA, 2011. ACM.

[47] S. Panjwani and E. Cutrell. Usably Secure, Low-Cost Authentication for Mobile Banking. In *Proc. 6th Symp. Usable Privacy and Security*, SOUPS '10, pages 4:1–4:12, New York, NY, USA, 2010. ACM.

[48] PCI Security Standards Council, LLC. Data Security Standard — Requirements and Security Assessment Procedures. `https://www.pcisecuritystandards.org/documents/PCI_DSS_v3.pdf`.

[49] C. Penicaud and A. Katakam. Mobile Financial Services for the Unbanked: State of the Industry 2013. Technical report, GSMA, Feb. 2014.

[50] Qualys. SSL Server Test. `https://www.ssllabs.com/ssltest/`.

[51] Reserve Bank of India. Master Circular - KYC norms, AML standards, CFT, Obligation of banks under PMLA, 2002. `http://rbidocs.rbi.org.in/rdocs/notification/PDFs/94CF010713FL.pdf`, 2013.

[52] Safaricom. Relax, you have got M-PESA. `http://www.safaricom.co.ke/personal/m-pesa/m-pesa-services-tariffs/relax-you-have-got-m-pesa`.

[53] A. Sharma, L. Subramanian, and D. Shasha. Secure Branchless Banking. In *3rd ACM Workshop on Networked Syst. for Developing Regions*, Big Sky, Montana, Oct. 2009.

[54] R. Shay, S. Komanduri, A. L. Durity, P. S. Huh, M. L. Mazurek, S. M. Segreti, B. Ur, L. Bauer, N. Christin, and L. F. Cranor. Can Long Passwords Be Secure and Usable? In *Proc. Conf. on Human Factors in Comp. Syst.*, CHI '14, pages 2927–2936, New York, NY, USA, 2014. ACM.

[55] The MITRE Corporation. CWE - Common Weakness Enumeration. `http://cwe.mitre.org/`.

[56] P. Traynor, P. McDaniel, and T. La Porta. *Security for Telecommunications Networks*. Springer, 2008.

Appendix

Package Name	Country	Downloads	Mallodroid Alert
bo.com.tigo.tigoapp	Bolivia	1000-5000	
br.com.mobicare.minhaoi	Brazil	500000-1000000	✗
com.cellulant.wallet	Nigeria	100-500	✗
com.directoriotigo.hwm	Honduras	10000-50000	
com.econet.ecocash	Zimbabwe	10000-50000	
com.ezuza.mobile.agent	Mexico	10-50	
com.f1soft.esewa	Nepal	50000-100000	
com.fetswallet_App	Nigeria	100-500	
com.globe.gcash.android	Philippines	10000-50000	✗
com.indosatapps.dompetku	Indonesia	5000-10000	✗
com.japps.firstmonie	Nigeria	50000-100000	
com.m4u.vivozuum	Brazil	10000-50000	✗
com.mcoin.android	Indonesia	1000-5000	✗
com.mdinar	Tunisia	500-1000	✗
com.mfino.fortismobile	Nigeria	100-500	✗
com.mibilleteramovil	Argentina	500-1000	
com.mobilis.teasy.production	Nigeria	100-500	
com.mom.app	India	10000-50000	
com.moremagic.myanmarmobilemoney	Myanmar	191	
com.mservice.momotransfer	Vietnam	100000-500000	✗
com.myairtelapp	India	1000000-5000000	✗
com.oxigen.oxigenwallet	India	100000-500000	✗
com.pagatech.customer.android	Nigeria	1000-5000	
com.palomar.mpay	Thailand	100000-500000	✗
com.paycom.app	Nigeria	10000-50000	✗
com.pocketmoni.ui	Nigeria	5000-10000	
com.ptdam.emoney	Indonesia	100000-500000	Market Restriction
com.qulix.mozido.jccul.android	Jamaica	1000-5000	✗
com.sbg.mobile.phone	South Africa	100000-500000	N/A
com.simba	Lebanon	1000-5000	✗
com.SingTel.mWallet	Singapore	100000-500000	✗
com.suvidhaa.android	India	10000-50000	Market Restriction
com.tpago.movil	Dominican Republic	5000-10000	✗
com.useboom.android	Mexico	5000-10000	✗
com.vanso.gtbankapp	Nigeria	100000-500000	
com.wizzitint.banking	South Africa	100-500	✗
com.zenithBank.eazymoney	Nigeria	50000-100000	✗
mg.telma.mvola.app	Madagascar	1000-5000	N/A
net.omobio.dialogsc	Sri Lanka	50000-100000	✗
org.readycash.android	Nigeria	1000-5000	
qa.ooredoo.omm	Qatar	5000-10000	
sv.tigo.mfsapp	El Salvador	10000-50000	✗
Tag.Andro	Côte d'Ivoire	500-1000	
th.co.truemoney.wallet	Thailand	100000-500000	✗
tz.tigo.mfsapp	Tanzania	50000-100000	✗
uy.com.antel.bits	Uruguay	10000-50000	
com.vtn.vtnmobilepro	Nigeria	Unknown	
za.co.fnb.connect.itt	South Africa	500000-1000000	

Table 3: We found 48 mobile money Android applications across 28 countries. Highlighted rows represent those applications manually analyzed in this paper. We were unable to obtain two apps due to Android market restrictions. Mallodroid was unable to analyze the apps marked N/A.

Measuring the Longitudinal Evolution of the Online Anonymous Marketplace Ecosystem

Kyle Soska and Nicolas Christin
Carnegie Mellon University
{ksoska, nicolasc}@cmu.edu

Abstract

February 2011 saw the emergence of Silk Road, the first successful online anonymous marketplace, in which buyers and sellers could transact with anonymity properties far superior to those available in alternative online or offline means of commerce. Business on Silk Road, primarily involving narcotics trafficking, rapidly boomed, and competitors emerged. At the same time, law enforcement did not sit idle, and eventually managed to shut down Silk Road in October 2013 and arrest its operator. Far from causing the demise of this novel form of commerce, the Silk Road take-down spawned an entire, dynamic, online anonymous marketplace ecosystem, which has continued to evolve to this day. This paper presents a long-term measurement analysis of a large portion of this online anonymous marketplace ecosystem, including 16 different marketplaces, over more than two years (2013–2015). By using long-term measurements, and combining our own data collection with publicly available previous efforts, we offer a detailed understanding of the growth of the online anonymous marketplace ecosystem. We are able to document the evolution of the types of goods being sold, and assess the effect (or lack thereof) of adversarial events, such as law enforcement operations or large-scale frauds, on the overall size of the economy. We also provide insights into how vendors are diversifying and replicating across marketplaces, and how vendor security practices (e.g., PGP adoption) are evolving. These different aspects help us understand how traditional, physical-world criminal activities are developing an online presence, in the same manner traditional commerce diversified online in the 1990s.

1 Introduction

In February 2011, a new Tor hidden service [16], called "Silk Road," opened its doors. Silk Road portrayed itself as an online anonymous marketplace, where buyers and sellers could meet and conduct electronic commerce transactions in a manner similar to the Amazon Marketplace, or the fixed price listings of eBay. The key innovation in Silk Road was to guarantee stronger anonymity properties to its participants than any other online marketplace. The anonymity properties were achieved by combining the network anonymity properties of Tor hidden services—which make the IP addresses of both the client and the server unknown to each other and to outside observers—with the use of the pseudonymous, decentralized Bitcoin electronic payment system [33]. Silk Road itself did not sell any product, but provided a feedback system to rate vendors and buyers, as well as escrow services (to ensure that transactions were completed to everybody's satisfaction) and optional hedging services (to buffer fluctuations in the value of the bitcoin).

Embolden by the anonymity properties Silk Road provided, sellers and buyers on Silk Road mostly traded in contraband and narcotics. While Silk Road was not the first venue to allow people to purchase such goods online—older forums such at the Open Vendor Database, or smaller web stores such as the Farmer's Market predated it—it was by far the most successful one to date at the time due to its (perceived) superior anonymity guarantees [13]. The Silk Road operator famously declared in August 2013 in an interview with Forbes, that the "War on Drugs" had been won by Silk Road and its patrons [18]. While this was an overstatement, the business model of Silk Road had proven viable enough that competitors, such as Black Market Reloaded, Atlantis, or the Sheep Marketplace had emerged.

Then, in early October 2013, Silk Road was shut down, its operator arrested, and all the money held in escrow on the site confiscated by law enforcement. Within the next couple of weeks, reports of Silk Road sellers and buyers moving to Silk Road's ex-competitors (chiefly, Sheep Marketplace and Black Market Reloaded) or starting their own anonymous marketplaces started to surface. By early November 2013, a novel incarnation of Silk Road, dubbed "Silk Road 2.0" was online—set up by former administrators and vendors of the original Silk Road.[1] Within a few months, numerous marketplaces following the same model of offering an online anonymous rendez-vous point for sellers and buyers appeared. These different marketplaces offered various levels of sophistication, durability and specialization (drugs, weapons, counterfeits, financial accounts, ...). At the same time, marketplaces would often disappear, sometimes due to arrests (e.g., as was the case with Utopia [19]), sometimes voluntarily (e.g., Sheep Marketplace [34]). In other words, the anonymous online marketplace ecosystem had evolved significantly compared to the early days when Silk Road was nearly a monopoly.

In this paper, we present our measurements and analysis of the anonymous marketplace ecosystem over a period of two and a half years between 2013 and 2015. Previous studies either focused on a specific marketplace (e.g., Silk Road [13]), or on simply describing high-level characteristics of certain marketplaces, such as the number of posted listings at a given point in time [15].

By using long-term measurements, combining our own data collection with publicly available previous efforts, and validating the completeness of our dataset using capture and recapture estimation, we offer a much more detailed understanding of the evolution of the online anonymous marketplace ecosystem. In particular, we are able to measure the effect of the Silk Road takedown on the overall sales volume; how reported "scams" in some marketplaces dented consumer confidence; how vendors are diversifying and replicating across marketplaces; and how security practices (e.g., PGP adoption) are evolving. These different aspects paint what we believe is an accurate picture of how traditional, physical-world criminal activities are developing an online presence, in the same manner traditional commerce diversified online in the 1990s.

We discover several interesting properties. Our analysis of the sales volumes demonstrates that as a whole the online anonymous marketplace ecosystem appears to be resilient, on the long term, to adverse events such as law enforcement take-downs or "exit scams" in which the operators abscond with the money. We also evidence stability over time in the types of products being sold and purchased: cannabis-, ecstasy- and cocaine-related products consistently account for about 70% of all sales. Analyzing vendor characteristics shows a mix of highly specialized vendors, who focus on a single product, and sellers who sell a large number of different products. We also discover that vendor population has long-tail characteristics: while a few vendors are (or were) highly successful, the vast majority of vendors grossed less than $10,000 over our entire study interval. This further substantiates the notion that online anonymous marketplaces are primarily competing with street dealers, in the retail space, rather than with established criminal organizations which focus on bulk sales.

The rest of this paper is structured as follows. Section 2 provides a brief overview of how the various online marketplaces we study operate. Section 3 describes our measurement methodology and infrastructure. Section 4 presents our measurement analysis. We discuss limitations of our approach and resulting open questions in Section 5, before introducing the related work in Section 6 and finally concluding in Section 7.

2 Online Anonymous Marketplaces

The sale of contraband and illicit products on the Internet can probably be traced back to the origins of the Internet itself, with a number of forums and bulletin board systems where buyers and sellers could interact.

However, online markets have met with considerable developments in sophistication and scale, over the past six years or so, going from relatively confidential "classifieds"-type of listings such as on the Open Vendor Database, to large online anonymous marketplaces. Following the Silk Road blueprint, modern online anonymous markets run as Tor hidden services, which gives participants (marketplace operators and participants such as buyers and sellers) communication anonymity properties far superior to those available from alternative solutions (e.g., anonymous hosting); and use pseudonymous online currencies as payment systems (e.g., Bitcoin [33]) to make it possible to exchange money electronically without the immediate traceability that conventional payment systems (wire transfers, or credit card payments) provide.

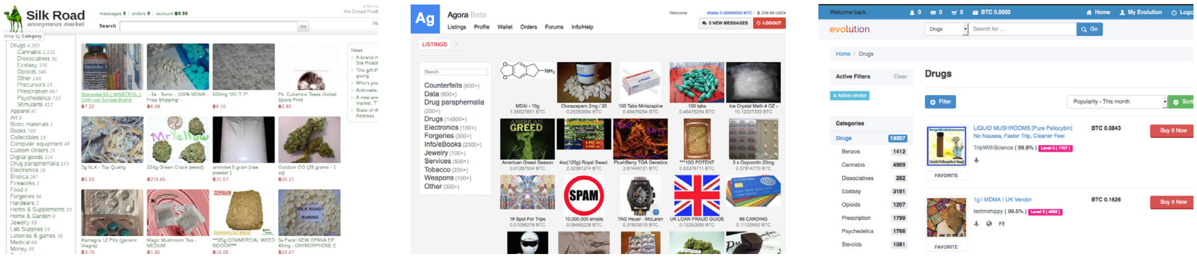
The common point between all these marketplaces is that they actually are not themselves selling contraband. Instead, they are risk management platforms for participants in (mostly illegal) transactions. Risk is mitigated on several levels. First, by abolishing physical interactions between transacting parties, these marketplaces claim to reduce (or indeed, eliminate) the potential for physical violence during the transaction.

Second, by providing superior anonymity guarantees compared to the alternatives, online anonymous marketplaces shield – to some degree[2] – transaction participants from law enforcement intervention.

Third, online anonymous marketplaces provide an escrow system to prevent financial risk. These systems are very similar in spirit to those developed by electronic

[1] Including, ironically, undercover law enforcement agents [7].

[2] Physical items still need to be delivered, which is a potential intervention point for law enforcement as shown in documented arrests [4].



(a) Silk Road (b) Agora (c) Evolution

Figure 1: **Example of marketplaces.** Most marketplaces use very similar interfaces, following the original Silk Road design.

commerce platforms such as eBay or the Amazon Marketplace. Suppose Alice wants to purchase an item from Bob. Instead of directly paying Bob, she pays the marketplace operator, Oscar. Oscar then instructs Bob that he has received the payment, and that the item should be shipped. After Alice confirms receipt of the item, Oscar releases the money held in escrow to Bob. This allows the marketplace to adjudicate any dispute that could arise if Bob claims the item has been shipped, but Alice claims not to have received it. Some marketplaces claim to support Bitcoin's recently standardized "multisig" feature which allows a transaction to be redeemed if, e.g., two out of three parties agree on its validity. For instance, Alice and Bob could agree the funds be transferred without Oscar's explicit blessing, which prevents the escrow funds from being lost if the marketplace is seized or Oscar is incapacitated.[3]

Fourth, and most importantly for our measurements, online anonymous marketplaces provide a feedback system to enforce quality control of the goods being sold. In marketplaces where feedback is mandatory, feedback is a good proxy to derive sales volumes [13]. We will adopt a similar technique to estimate sales volumes.

At the time of this writing the Darknet Stats service [1] lists 28 active marketplaces. As illustrated in Fig. 1 for the Evolution and Agora marketplaces, marketplaces tend to have very similar interfaces, often loosely based on the original Silk Road user interface. Product categories (on the right in each screen capture) are typically self-selected by vendors. We discovered that categories are sometimes incorrectly chosen, which led us to build our own tools to properly categorize items. Feedback data (not shown in the figure) comes in various flavors. Some marketplaces provide individual feedback per product and per transaction. This makes computation of sales volumes relatively easy as long as one can determine with good precision the time at which each piece of feedback was issued. Others provide feedback per vendor; if we can then link vendor feedback to specific items, we can again obtain a good estimate for sales volumes, but if not, we may not be able to derive any meaningful numbers. Last, in some marketplaces, feedback is either not mandatory, or only given as aggregates (e.g., "top 5% vendor"), which does not allow for detailed volume analysis.

3 Measurement methodology

Our measurement methodology consists of 1) crawling online anonymous marketplaces, and 2) parsing them. Table 1 lists all the anonymous marketplaces for which we have data. We scraped 35 different marketplaces a total of 1,908 times yielding a dataset of 3.2 TB in size. The total number of pages obtained from each scrape ranged from 27 to 331,691 pages and performing each scrape took anywhere from minutes up to five days.

The sheer size of the data corpus we are considering, as well as other challenging factors (e.g., hidden service latency and poor marketplace availability) led us to devise a custom web scraping framework built on top of Scrapy [3] and Tor [16], which we discuss first. We then highlight how we decide to parse (or ignore) marketplaces, before touching on validation techniques we use to ensure soundness of our analysis.

3.1 Scraping marketplaces

We designed and implemented the scraping framework with a few simple goals in mind. First, we want our scraping to be carried out in a *stealthy* manner. We do not want to alert a potential marketplace administrator to our presence lest our page requests be censored, by either modifying the content in an attempt to deceive us or denying the request altogether.

[3] The Evolution marketplace claimed to support multisig. However, Evolution's operators absconded with escrow money on March 17th, 2015 [9]; it turns out that their multisig implementation did not function as intended, and was rarely used. Almost none of the stolen funds have been recovered so far.

[4] The November 2011–July 2012 Silk Road data comes from a previously reported collection effort, with publicly available data [13].

Marketplace	Parsed?	Measurement dates	# snap.
Agora	Y	12/28/13–06/12/15	161
Atlantis‡	Y	02/07/13–09/21/13	52
Black Flag‡	Y	10/19/13–10/28/13	9
Black Market Reloaded†	Y	10/11/13–11/29/13	25
Tor Bazaar*	Y	07/02/14–10/15/14	27
Cloud 9*	Y	07/02/14–10/28/14	27
Deep Bay‡	Y	10/19/13–11/29/13	24
Evolution‡	Y	07/02/14–02/16/15	43
Flo Market‡	Y	12/02/13–01/05/14	23
Hydra*	Y	07/01/14–10/28/14	29
The Marketplace†	Y	07/08/14–11/08/14	90
Pandora‡	Y	12/01/13–10/28/14	140
Sheep Marketplace‡	Y	10/19/13–11/29/13	25
Silk Road*[4]	Y	11/22/11–07/24/12	133
	Y	06/18/13–08/18/13	31
Silk Road 2.0*	Y	11/24/13–10/26/14	195
Utopia*	Y	02/06/14–02/10/14	10
AlphaBay	N	03/18/15–06/02/15	17
Andromeda‡	N	07/01/14–11/10/14	30
Behind Blood Shot Eyes‡	N	01/31/14–08/27/14	56
BlackBank	N	07/02/14–05/16/15	56
Blue Sky*	N	12/25/13–06/10/14	126
Budster‡	N	12/01/13–03/11/14	56
Deep Shop‡	N	01/31/14–03/09/14	20
Deep Zone†	N	07/01/14–07/08/14	10
Dutchy‡	N	01/31/14–08/07/14	86
Area 51‡	N	11/20/14–01/20/15	14
Freebay†	N	12/31/13–03/11/14	36
Middle Earth	N	11/21/14–06/02/15	15
Nucleus	N	11/21/14–05/26/15	22
Outlaw	N	01/31/14–04/20/15	99
White Rabbit†	N	01/14/14–05/26/14	61
The Pirate Shop‡	N	01/14/14–09/17/14	102
The Majestic Garden	N	11/21/14–06/02/15	23
Tom Cat†	N	11/18/14–12/08/14	11
Tor Market	N	12/01/13–12/23/13	24

Table 1: **Markets crawled.** The table describes which markets were crawled, the time the measurements spanned, and the number of snapshots that were taken. * denote market sites seized by the police, † voluntary shutdowns, and ‡ (suspected) fraudulent closures (owners absconding with escrow money).

Second, we want the scrapes to be *complete*, *instantaneous*, and *frequent*. Scrapes that are instantaneous and complete convey a coherent picture about what is taking place on the marketplace without doubts about possible unobserved actions or the inconsistency that may be introduced by time delay. Scraping very often ensures that we have high precision in dating when actions occurred, and reduces the chances of missing vendor actions, such as listing and rapidly de-listing a given item.

Third we want our scraper to be *reliable* even when the marketplace that we are measuring is not. Even when a marketplace is unavailable for hours, the scraper should hold state and retry to avoid an incomplete capture.

Fourth, the scraper should be capable of handling *client-side state* normally kept by the users browser such as cookies, and be *robust* enough to avoid any detection schemes that might be devised to thwart the scraper. We attempt to address these design objectives as follows.

Avoiding censorship Before we add a site to the scraping regimen, we first manually inspect it and identify its layout. We build and use as input to the scraper a configuration including regular expressions on the URLs for that particular marketplace. This allows us to avoid following links that may cause undesirable actions to be performed such as adding items to a cart, sending messages or logging out. We also provide as input to the scraper a session cookie that we obtain by manually logging into the marketplace and solving a CAPTCHA; and parameters such as the maximum desired scraping rate.

In addition to being careful about what to request from a marketplace, we obfuscate how we request content. For each page request, the scraper randomly selects a Tor circuit out of 20 pre-built circuits. This strategy ensures that the requests are being distributed over several rendezvous points in the Tor network. This helps prevent triggering anti-DDoS heuristics certain marketplaces use.[5] This strategy also provides redundancy in the event that one of the circuits being used becomes unreliable and speeds up the time it takes to observe the entire site.

Completeness, soundness, and instantaneousness The goal of the data collection is to make an observation of the entire marketplace at an instantaneous point in time, which yields information such as item listings, pricing information, feedback, and user pages. Instantaneous observations are of course impossible, and can only be approximated by scraping the marketplace as quickly as possible. Scraping a site aggressively however limits the stealth of the scraper; We manually identified sites that prohibit aggressive scraping (e.g., Agora) and imposed appropriate rate limits.

Scrape completeness is also crucial. A partial scrape of a site may lead to underestimating the activities taking place. Fortunately, since marketplaces leverage feedback to build vendor reputation, old feedback is rarely deleted. This means that it is sufficient for an item listing and its feedback to be eventually observed in order to know that the transaction took place. Over time, the price of an item may fluctuate however, and information about when the transaction occurred often becomes less precise, so it is much more desirable to observe feedback as soon as possible after it is left. We generally attempted a scrape for each marketplace once every two to three days unless the marketplace was either unavailable or the previous scrape had not yet completed; having collected most of the data we were interested in by that time, we scraped considerably less often toward the end of our data collection interval (February through May 2015).

Many marketplaces that we observed have quite poor reliability, with 70% uptime or lower. It is very difficult

[5] However some marketplaces, e.g., Agora, use session cookies to bind requests coming from different circuits, and require additional attention.

to extract entire scrapes from marketplaces suffering frequent outages. This is particularly true for large sites, where a complete scrape can take several days. As a workaround, we designed the scraping infrastructure to keep state and retry pages using an increasing back-off interval for up to 24 hours. Using such a system allowed the scraper to function despite brief outages in marketplace availability. Retrying the site after 24 hours would be futile as in most cases, the session cookie would have expired and the scrape would require a manual login, and thus a manual restart.

Most marketplaces require the user to log in before they are able to view item listings and other sensitive information. Fortunately, creating an account on these marketplaces is free. However, one typically needs to solve a CAPTCHA when logging in; this was done manually. The process of performing a scrape begins with manually logging into the marketplace, extracting the session cookie, and using it as input to the scrape to continue scraping under that session. In many cases the site will fail to respond to requests properly unless multiple cookies are managed or unless the user agent of the scraper matches the user agent of the browser that generated the cookie. We managed to emulate typical browser behavior in all but one case (BlueSky). We were unable to collect meaningful data on BlueSky, as an anti-scraping measure on the server side was to annihilate any session after approximately 100 page requests, and get the user to log in again.

3.2 Parsing marketplaces

The raw page data collected by the scraper needs to be parsed to extract information useful for analysis. The parser first identifies which marketplace a particular page was scraped from; it then determines which type of page is being analyzed (item listing, user page, feedback page, or any combination of those).

Each page is then parsed using a set of heuristics we manually devised for each marketplace. We treat the information extracted as a single *observation* and record it into a database. Information that does not exist or cannot be parsed is assigned default values.

The heuristics for parsing can often become quite complicated as many marketplaces observed over long periods of time went through several iterations of page formats. This justified our conscious decision to decouple scraping from parsing so that we could minimize data loss. Because of the high manual effort associated with creating and debugging new parsers for marketplaces, we only generated parsers for marketplaces that we perceived to be of significance. While observing the scrapes of several marketplaces, it became apparent that their volume was either extremely small (<\$1,000) or was not measurable by observing the website (e.g., because feedback is not mandatory). These marketplaces were omitted without greatly affecting the overall picture; their analysis is left for future work.

3.3 Internally validating data analysis

To ensure that the analysis we performed was not biased, and as a safety against egregious errors, both authors of this paper concurrently and independently developed multiple implementations of the analysis we present in the next section. During that stage of the work, the two authors relied on the same data sources, but used *different* analysis code and tools and did not communicate with each other until all results were produced.

We then internally confirmed that the independent estimations of total market volumes varied by less than 10% at any single point in time, and less than 5% on average, well within expected margin of errors for data indirectly estimated from potentially noisy sources (user feedback).[6] The independent reproducibility of the analysis is important since, as we will show, estimating market volumes presents many pitfalls, such as the risk of double-counting observations or using a holding price as the true value of an item.

3.4 Validating data completeness

The poor availability of certain marketplaces (e.g., Agora), combined with the large amount of time needed to fully scrape very large marketplaces raises concerns about data completeness. We attempt to estimate the amount of data that might be missing through a process known as marking and recapturing.

The basic idea is as follows. Consider that a given site scrape at time t contains a number M of feedback. Since we do not know whether the scrape is complete, we can only assert that M is a lower bound on the total number of feedback F actually present on the site at time t. Now, consider a second scrape (presumably taken after time t), which contains n pieces of feedback left at or before time t. The number n is another lower bound of F. We then estimate F as $\hat{F} = nM/m$, where m is the number of feedback captured in the first scrape that we *also* observe in the second scrape ($m \leq M$).

The Schnabel estimator [36] extends the above technique to estimate the size of a population to multiple samples, and is thus well-suited to our measurements. For n samples, if we denote by C_t the number of feedback in sample t, by M_t the total number of unique previously observed feedback in sample $(t-1)$, and by R_t the

[6] These minor discrepancies can be attributed to slightly different filtering heuristics, which we discuss later.

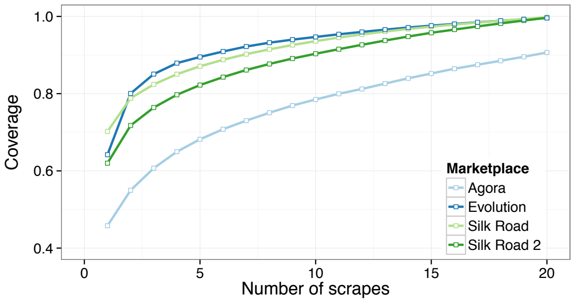


Figure 2: **Coverage of Agora, Silk Road 1, Silk Road 2, and Evolution.** This plot estimates the fraction of all feedback we obtain for a given time, as a function of the number of scrapes we collect.

number of previously observed feedback during sample t, we estimate the total number of feedback at time t as:

$$\hat{F} = \frac{\sum_{t=1}^{n} C_t M_t}{\sum_{t=1}^{n} R_t}\ .$$

The Schnabel estimator implicitly assumes that the distribution is time-invariant and that samples are drawn uniformly. To help ensure time invariance, the estimator begins with a sample at time t. Pieces of feedback with timestamps greater than t are omitted from all samples taken in the future ($t+\tau$). It is also important not to consider samples from too far into the future since items are occasionally de-listed and the corresponding feedback destroyed. To help minimize the impact of feedback deleted in the future, we only use samples within 60 days of t in our estimate.

We illustrate this estimate in Figure 2 for Agora, Silk Road 1, Silk Road 2, and Evolution after multiple observations have been made. Agora has relatively poor reliability and, on average, a single scrape will not manage to capture even half of the feedback present at that time on the site. On other marketplaces it is typical on the first visit to see as much as 60% of the entire population, or higher. After ten or more independent scrapes, we can expect to obtain a dataset that approaches 90% coverage or higher.

Figure 3 further illustrates our point, by comparing the number of pieces of feedback observed on Agora to its estimate. For most of the observed lifetime of Agora, the data that we have is very close to what we estimate the total to be. This is because information about a marketplace at a particular (past) point in time benefits from subsequent observations. Most recent observations do not have this benefit and therefore suffer from poor coverage, leading to significant divergence from their estimate. This results in potentially large underestimations towards the very end of our dataset, which will require us to censor some of this data when estimating volumes.

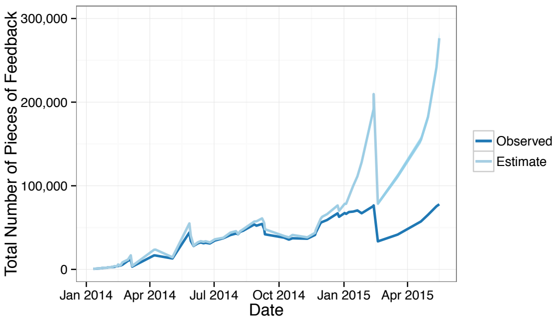


Figure 3: **Observed and estimated number of feedback present on Agora over time.** The lower and upper bounds for the estimate are nearly indistinguishable from the estimate itself.

4 Analysis

We next turn to data analysis. We first estimate the overall evolution of the sales volumes in the entire ecosystem over the past couple of years. We then move to an assessment of the types of products being sold over time. Last, we discuss findings about vendor activity and techniques.

4.1 Sales volumes

The first important question that our analysis answers is how much product in terms of money is being bought and sold on online anonymous marketplaces. While we cannot directly measure the money being transacted from buyers to sellers, or packages being shipped from vendors to customers, we do make frequent observations of product feedback left for particular item listings on the marketplaces. Similar to prior work [13], we use these observations of feedback as a proxy to estimate a lower bound for sales.

Caveats In many marketplaces (e.g., Silk Road, Silk Road 2.0, Agora, Evolution among others) customers are required to leave feedback for a vendor whenever they receive their order of one of the vendor's items. An order for an item may be of varying quantity, so a customer that purchases a single quantity of a product, and a customer that purchases multiple quantities of a product will both leave a single feedback. In an effort to be conservative, we make the assumption that for every feedback observed, only a single quantity was purchased.

Our prudent strategy of estimating sales volume from confirmed observations of feedback diverges from other, simpler approaches, such as counting the number of item listings offered (see, e.g., [15]). For instance, over the observed lifetime of Evolution, a few of the most successful item listings had feedback entries that indicated over 1 million dollars had been spent on each of them. The presence of these highly influential item listings suggests that simply counting the total number of listings on a site is a very poor indicator of sales volume. This claim is compounded by the observation that the average sales per item listing per day on Evolution in early July of 2014 was \$85.14; but by September 2014, after new vendors and item listings had entered, the sales per item listing had declined to \$19.42. Such volatile behavior is particularly common in marketplaces that are small or are going through periods of rapid growth.

Estimation We derived the estimates for the total amount of money transacted in three steps. We first took the set of all feedback observations that had been collected and removed any duplicates. For example, on two consecutive scrapes of a particular marketplace, the same item listing and its entire feedback history were observed and recorded twice. It would be incorrect to count two different observations of the same feedback twice. We thus developed a criterion for uniqueness for each marketplace—typically enforcing uniqueness of fields such as feedback message body, the vendor for which the feedback was left, the title of the item listing and the approximate date the feedback was left. Two pieces of feedback are considered different if and only if they differ in at least one of these categories.

The second step was to identify the the point in time at which the feedback was left. This time is an upper bound on when the transaction occurred. We obtained this estimate by noting the time of the observation and utilizing any information available about the age of the feedback. Different marketplaces have varying precision information about feedback timestamps. In the most precise instances, the time that the feedback was left is specified within the hour; in the most ambiguous cases, we can only infer the month in which feedback was deposited. Fortunately, due to our rather high sampling rate of the marketplaces, in most instances we have roughly a 24-hour accuracy on feedback time.

The third and final step is to identify the value of the transaction that each feedback represents. This involves pairing each feedback observation with a single observation of an item listing and its advertised price. Careful attention must be paid here as a few caveats exist, namely that the advertised price of an item listing varies with time, and that, in some rare cases, the corresponding item is never observed, leaving us unable to identify the value of the transaction.

Item prices change for two different reasons. The first and most common reason is that the vendors responsible for selling items are subject to standard free market pressures and may raise or lower their prices in response to competition, supply, demand, or other factors. The second reason is that when a vendor temporarily wishes to halt sales of an item with the expectation of selling it again in the future, instead of de-listing the item and losing all of the reviews and ratings that have accumulated over time, the vendor instead raises the price to something prohibitively high in order to discourage any sales. This is what we call a *holding price*. Holding prices are particularly dangerous for our analysis, because they can be in excess of millions of dollars. So, mistaking a holding price for an actual price just once could have dramatic consequences on the overall analysis.

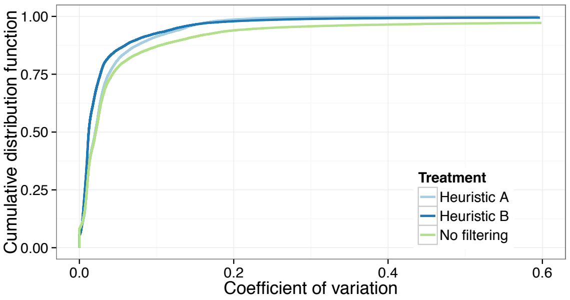


Figure 4: **C.d.f. of Coefficient of Variation for sets of observations of item listings** Both heuristics perform very similarly.

Dealing with holding prices Given a particular feedback and a set of observations of the corresponding product listing, the objective becomes to determine which observation yields the most accurate price for that feedback. Independent analysis (see Section 3.3) yielded two different heuristics for solving this problem. In the first heuristic (Heuristic A), we dismissed observations of the listing where the price was greater than \$10,000 USD as well as observations that showed prices of zero (free). We then dismissed observations that were greater than 5 times the median of the remaining samples as well as observations that were less than 25% the value of the median. We manually observed thousands of product listings and identified that only in some very rare cases were the assumptions violated.

The second heuristic (Heuristic B) proceeded by removing observations with a price >\$10,000 USD, as well as the upper quartile and any observations that were more than 100 times greater than the observation corresponding to the cheapest, non-zero price. To understand the effect that these heuristics had on observations, we calculated the coefficient of variation defined as $c_v = \sigma/\mu$ (standard deviation over mean) for the set

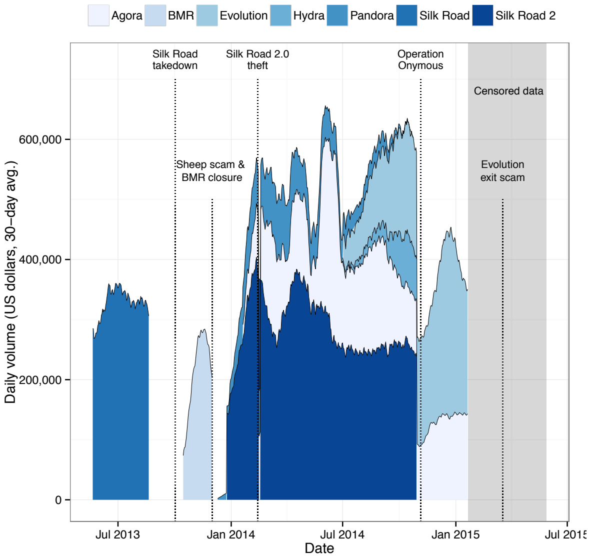


Figure 5: **Sales volumes in the entire ecosystem.** This stacked plot shows how sales volume vary over time for the marketplaces we study.

of observations for each item listing and plotted its cumulative distribution function.

Figure 4 shows that without any filtering, about 5% of all item listings were at some point sampled with highly variable prices, which suggests that a holding price was observed for this listing. Both heuristics produce relatively similar filtering; we ended up using Heuristic A in the rest of the analysis.

After applying the filter, there is still some smaller variation in the pricing of many listings which is consistent with the fluctuation in prices due to typical market pressures but it is clear that no listings with extremely high variations remain. 79,512 total unique item listings were identified, 1,003 (1.26%) of which had no valid observations remaining after filtering, meaning that the output of the heuristic was the empty set, the remaining 78,509 item listings returned at least one acceptable observation.

After filtering the listing observations, we pair each feedback with one of the remaining listing samples. To minimize the difference in estimated price of the feedback from the true price, we select the listing observation that is closest to the feedback in time. At this point we have a set of unique pieces of feedback, each mapped to a price at some point in time; from there, we can construct an estimate for the sales volumes.

Results We present our results in Figure 5 where we show the total volume, per marketplace we study, over time. The plot is stacked, which means that the top line corresponds to the total volume cleared by all marketplaces under study. In early 2013, we only have results for Silk Road, which at that point grossed around $300,000/day, far more than previously estimated for 2012 [13]. This number would project to over $100M in a year; combined by the previous $15M estimate [13] for early 2012, and "filling in" gaps for data we do not have in late 2012, appears consistent with the (revised) US Government calculations of $189M of total grossed income by Silk Road over its lifetime, based on Bitcoin transaction logs.

We then have a data collection gap, roughly corresponding to the time Silk Road was taken down. (We do not show volumes for Atlantis, which are negligible, in the order of $2,000–3,000/day.) Shortly after the Silk Road take-down we started measuring Black Market Reloaded, and realized that it has already made up for a vast portion of the volumes previously seen on Silk Road. We do not have sales data for Sheep Marketplace due to incomplete parses, but we do believe that the combination of both markets made up for the loss of Silk Road. Then, both Sheep and Black Market Reloaded closed – in the case of Sheep, apparently fraudulently. There was then quite a bit of turmoil with various markets starting and failing quickly. Only around late November 2013 did the ecosystem find a bit more stability, as Silk Road 2.0 had been launched and was rapidly growing. In parallel Pandora, Agora, and Evolution were also launched. By late January 2014, volumes far exceeded what was seen prior to the Silk Road take-down. At that point, though, a massive scam on Silk Road 2.0 caused dramatic loss of user confidence, which is evidenced by the rapid decrease until April 2014, before it starts recovering. Competitors however were not affected. (Agora does show spikes due to very imprecise feedback timing at a couple of points.) Eventually, in the Fall of 2014, the anonymous online marketplace ecosystem reached unprecedented highs. We started collecting data from Evolution in July, so it is possible that we miss quite a bit in the early part of 2014, but the overall take-away is unchanged. Finally, in November 2014, Operation Onymous [38] resulted in the take-down of Silk Road 2 and a number of less marketplaces. This did significantly affect total sales, but we immediately see a rebound by people going to Evolution and Agora. We censor the data we obtained from February 2015: at that point we only have results for Agora and Evolution, but coverage is poor, and as explained in Section 3, is likely to underestimate volumes significantly. We did note a short volume decrease prior to the Evolution "exit scam" of March 2015. We have not analyzed data for other smaller marketplaces (e.g., Black Bank, Middle Earth, or Nucleus) but suspect the volumes are much smaller. Finally, more recent marketplaces such as AlphaBay seem to have grown rapidly after the Evolution exit scam, but feedback on AlphaBay is not mandatory, and thus cannot be used to reliably estimate sales volumes.

In short, the entire ecosystem shows resilience to scams – Sheep, but also Pandora, which, as we can see started off very well before losing ground due to a loss in customer confidence, before shutting down. The effect of law enforcement take-downs (Silk Road 1&2, Operation Onymous) is mixed at best: the ecosystem relatively quickly recovered from the Silk Road shutdown, and appears to have withstood Operation Onymous quite well, since aggregate volumes were back within weeks to more than half what they were prior to Operation Onymous. We however caution that one would need longer term data to fully assess the impact of Operation Onymous.

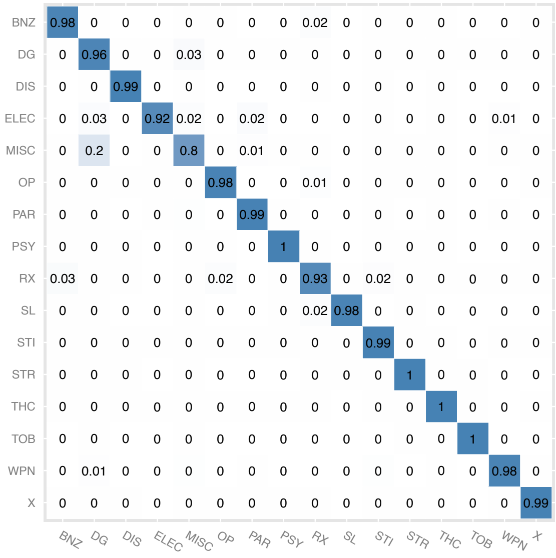
4.2 Product categories

In addition to estimating the value of the products that are being sold, we strived to develop an understanding of what is being sold. Several marketplaces such as Agora and Evolution include information on item listing pages that describe the nature of the listing as provided by the vendor that posted it. Unfortunately these descriptions are often too specific, conflict across marketplaces, and in the case of some sites, are not even available at all.

For our analysis, we need a consistent and coherent labeling for all items, so that we could categorize them into broad mutually exclusive categories. We thus implemented a machine learning classifier that was trained and tested on samples from Agora and Evolution, where ground truth was available via labeling. We then took this classifier and applied it to item listings on all marketplaces to answer the question of what is being sold.

We took 1,941,538 unique samples from Evolution and Agora, where a sample is the concatenation of an item listing's title and all descriptive information about it that was parsed from the page. We tokenized each sample under the assumption that the sample is written in English, resulting in a total of 162,198 unique words observed. We then computed a *tf-idf* value for each of the 162,198 words in the support for each sample, and used these values as inputs to an L2-Penalized SVM under L2-Loss implemented using Python and *scikit-learn*.

We evaluated our classifier using 10-fold cross validation. The overall precision and recall were both (roughly) 0.98. We also evaluated the classifier on Agora data when trained with samples from Evolution and vice-versa to ensure that the classifier was not biased to only perform well on the distributions it was trained on. The confusion matrix in Figure 6 shows that classification performance is very strong for all categories. Only "Misc" is occasionally confused with Digital Goods and Prescriptions are occasionally confused with Benzos (which in fact is not necessarily surprising). We believe that these errors are most likely caused by mislabeled test samples. Although we drew our samples from Evolution and Agora which provide a specific label for each listing, the label is selected by the vendor and may be erroneous, particularly for listings that are hard to place. Manual inspection revealed that several of the errors came from item listings that offered US $100 Bills in exchange for Bitcoin.



	BNZ	DG	DIS	ELEC	MISC	OP	PAR	PSY	RX	SL	STI	STR	THC	TOB	WPN	X
BNZ	0.98	0	0	0	0	0	0	0	0.02	0	0	0	0	0	0	0
DG	0	0.96	0	0	0.03	0	0	0	0	0	0	0	0	0	0	0
DIS	0	0	0.99	0	0	0	0	0	0	0	0	0	0	0	0	0
ELEC	0	0.03	0	0.92	0.02	0	0.02	0	0	0	0	0	0	0	0.01	0
MISC	0	0.2	0	0	0.8	0	0.01	0	0	0	0	0	0	0	0	0
OP	0	0	0	0	0	0.98	0	0	0.01	0	0	0	0	0	0	0
PAR	0	0	0	0	0	0	0.99	0	0	0	0	0	0	0	0	0
PSY	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
RX	0.03	0	0	0	0	0.02	0	0	0.93	0	0.02	0	0	0	0	0
SL	0	0	0	0	0	0	0	0	0.02	0.98	0	0	0	0	0	0
STI	0	0	0	0	0	0	0	0	0	0	0.99	0	0	0	0	0
STR	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
THC	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
TOB	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
WPN	0	0.01	0	0	0	0	0	0	0	0	0	0	0	0	0.98	0
X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.99

Figure 6: **Classifier confusion matrix.** BNZ: Benzos, DG: Digital Goods, DIS: Dissociatives, ELEC: Electronics, MISC: Miscellaneous, OP: Opioids, PAR: Drug Paraphernalia, PSY: Psychedelics, RX: Prescription drugs, SL: Sildenafil, STI: Stimulants, STR: Steroids, THC: Cannabis, TOB: Tobacco, WPN: Weapons, X: Ecstasy.

We then applied the classifier to the aggregate analysis performed earlier. In addition to placing a particular feedback in time, and pairing it with an item listing observation to derive the price, we predicted the class label of that listing and aggregated the price by class label. Figure 7 shows the normalized market aggregate by category. Drug paraphernalia, weapons, electronics, tobacco, sildenafil, and steroids were collapsed into a category called 'Other' for clarity.

Over time the fraction of market share that belongs to each category is relatively stable. However, around October of 2013, December 2013, March 2014, and January 2015, cannabis spikes up to as much as half of the market share. These spikes correspond to the earlier mentioned 1) take-down of Silk Road, 2) closure of Black Market Reloaded and Sheep scam, 3) Silk Road 2.0 theft [5],

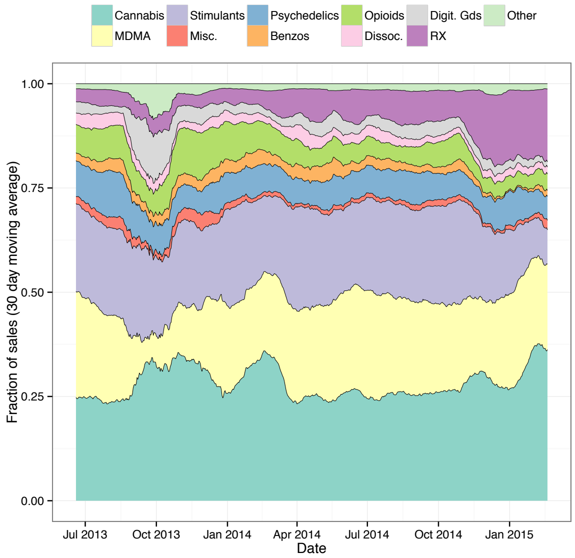


Figure 7: **Fractions of sales per item category.**

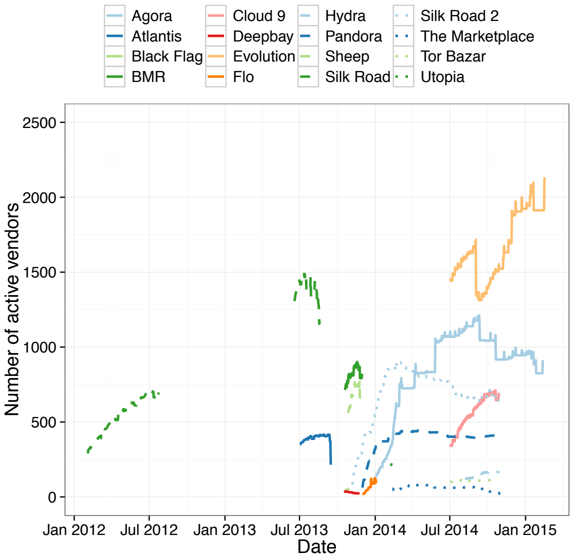


Figure 8: **Evolution of the number of active sellers over time.** Each "seller" here corresponds to a unique marketplace-vendor name pair. Certain sellers participate in several marketplaces and are thus counted multiple times here.

and 4) Operation Onymous respectively. These are all events that generated substantial doubts in both vendors and consumers regarding the safety and security of operating on these marketplaces. At these times the perceived risk of operation was higher, which may have exerted pressure towards buying and selling cannabis as opposed to other products for which the punishment if caught is much more severe. We can also see that digital goods take an unusually high market share in times of uncertainty, which is most obvious around October 2013: this is not surprising as digital goods are often a good way to quickly accumulate large numbers of listings on a new marketplace.

Figure 7 shows that after an event such as a take-down or large scale scam occurs, it takes about 2–3 months before consumer and vendor confidence is restored and the markets converge back to equilibrium. At equilibrium, cannabis and MDMA (ecstasy) are about 25% of market demand each with stimulants closely behind at about 20%. Psychedelics, opioids, and prescription drugs are a little less than 10% of market demand each, although starting in November 2014, prescription drugs have gained significant traction—perhaps making anonymous marketplaces a viable alternative to unlicensed online pharmacies.

4.3 Vendors

Online anonymous marketplaces are only successful when they manage to attract a large enough vendor population to provide a critical mass of offerings. At the same time, vendors are not bound to a specific marketplace. Anecdotal evidence shows that certain sellers list products on several marketplaces at once; likewise, certain sellers "move" from marketplace to marketplace in response to law enforcement take-down or other marketplace failures. Here, we try to provide a good picture of the vendor dynamics across the entire ecosystem.

Number of sellers Figure 8 shows, over time, the evolution of the number of active sellers on all the marketplaces we considered. For each marketplace, a seller is defined as active at time T is we observed her having at least one active listing at time $t \leq T$, and at least one active listing (potentially the same) at a time $t \geq T$. This is a slightly different definition from that used in Christin [13] which required an active listing at time t to count a seller as active. For us, active sellers include sellers that may be on vacation but will come back, whereas Christin did not include such sellers. As a result, our results for Silk Road are very slightly higher than his.

The main takeaway from Figure 8 is that the number of sellers overall has considerably increased since the days of Silk Road. By the time Silk Road stopped activities in 2013, it featured around 1,400 sellers; its leading competitors, Atlantis and Black Market Reloaded (BMR) were much smaller. After the Silk Road take-down (October 2013) and Atlantis closure, we observe that both BMR and the Sheep marketplace rapidly pick up a large influx of sellers. In parallel, Silk Road 2.0 also grows at

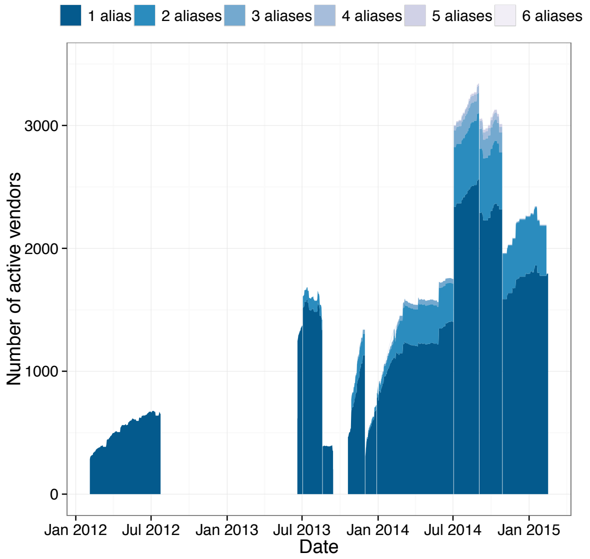


Figure 9: **Number of aliases per seller.** This plot shows the evolution of the number of aliases per seller across all marketplaces, over time. The contour of the curve denotes the total number of sellers overall.

a very rapid pace. Successful newcomers like Pandora, Agora, and Evolution also see quick rises in the number of sellers. After a certain amount of time, however, per-marketplace population tends to stabilize, even in the most popular marketplaces. On the other hand, we also observe that some marketplaces never took off: The Marketplace, Hydra, Deepbay, and Tor Bazaar, for instance, consistently have a small number of vendors. In other words, we see very strong network effects: Either marketplaces manage to get initial traction and then rapidly flourish, or they never manage to take off.

Sellers and aliases After Silk Road was taken down, a number of sellers reportedly moved to Black Market Reloaded or the Sheep Marketplace. More generally, nothing prevents a vendor from opening shop on multiple marketplaces; in fact, it is probably a desirable strategy to hedge against marketplace take-downs or failures. As a result, a given seller, Sally, may have multiple vendor accounts on several marketplaces: Sally may sell on Silk Road 2 as "Sally," on Agora as "sally" and on Evolution as "Easy Sally;" she may even have a second Evolution account ("The Real Easy Sally").

We formally define an *alias* as a unique (*vendor nickname*, *marketplace*) pair, and link different aliases to the same vendor using the combination of the following three heuristics. We first consider vendor nicknames on different marketplaces with only case differences as belonging to the same person (e.g., "Sally" and "sally"). We then use the InfoDesk feature of the Grams "DarkNet Markets" search engine [2] to further link various vendor nicknames.[7] We filter out vendor nicknames consisting only of a common substring (e.g., "weed," "dealer," "Amsterdam," ...) used by many vendors prior to conducting the search. Finally, we link all vendor accounts that claim to be using the same PGP key. Clearly, our linking strategy is very conservative – in the sense that minor variations like "Sally" and "Sally!" will not be linked absent a common PGP key.

Using this set of heuristics, from a total of 29,258 unique aliases observed across our entire measurement interval, we obtain a list of 9,386 sellers. In Figure 9, we show, over time, the number of vendors that have one, two or up to six aliases active at any given time T (where we use the same definition of "active" as earlier, i.e., the alias has at least one listing available before and after T). The plot is by definition incomplete since we can only take into account, for each time t, the marketplaces that we have crawled (and parsed) at time t.

For instance, the earlier part of the data show a complete monopoly: this is not surprising since we only have data for Silk Road at that time, even though Black Market Reloaded was also active at the same time. We observe in the summer of 2013 that a few vendors sell simultaneously on Silk Road and Atlantis, but the practice of having multiple vendor accounts on several sites seems to only really take hold in 2014, after many marketplaces failed in the Fall of 2013 (including Silk Road, and many of its short-lived successors). The second jump in July 2014 corresponds to our starting to collect data for the very large Evolution marketplace. Finally, the decrease observed in late 2014 is due to Operation Onymous [38], which – besides Silk Road 2.0 – took down a relatively large number of secondary marketplaces, such as Cloud 9.

Besides the relatively robust rise is the number of sellers to take-downs and scams, the main takeaway from this plot is that the majority of sellers appear to only use one alias – but this may be a bit misleading, as (as we will see later) a large number of vendors sell extremely limited quantities of products. An interesting extension would be to check whether "top" vendors diversify across marketplaces or not.

We complement this analysis by looking into the "survivability" functions of aliases and sellers, which we report in Figure 10. Here the survival function is defined as the probability $p(\tau)$ that a given seller (resp. alias) observed at time t be still active at time $t+\tau$. The figure shows the survival function, derived from a Kaplan-Meier estimator [24] to account for the fact that we have finite measurement intervals, along with 95% confidence

[7] It is not clear how the Grams search engine is implemented; we suspect the vendor directory is primarily based on manual curation.

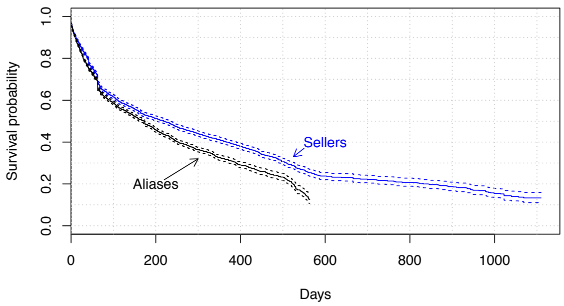


Figure 10: **Seller survivability analysis.** The plot describes the probability a given alias is still active after a certain number of days; and the probability a given seller (regardless of which alias it is using) is still active after a certain number of days. On average, sellers are active for 220 days, while aliases remain active for 172 days.

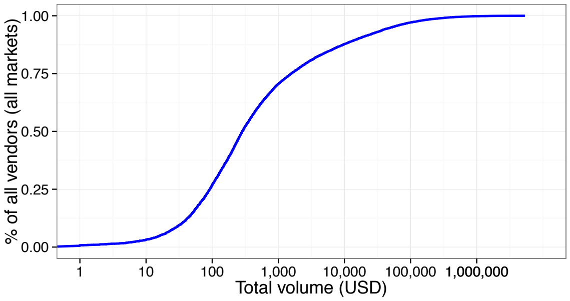


Figure 11: **Seller volumes.** A very small fraction of sellers generate significant profit. On average, a typical seller only makes a couple of hundreds dollars.

intervals. The key findings here are that half of the sellers are only present for 220 days or less; half of the aliases only exist for 172 days or less. More interesting is the "long-tail" phenomenon we observe: a number (more than 10%) of sellers have been active *throughout the entire measurement interval*. More generally approximately 25% of all sellers are "in it for the long run," and remain active (with various aliases on various marketplaces) for years.

Volumes per vendor In an effort to obtain a more clear understanding of how vendors operate, we aggregated unique feedback left for products by vendor. We used this to calculate the total value of the transactions for items sold by each vendor and then grouped these vendor aliases to yield the total value of transactions for each seller. Figure 11 plots the CDF of sellers by the total value of their transactions. About 70% of all sellers never managed to sell more than $1,000 worth of products. Another 18% of sellers were observed to sell between $1,000 and $10,000 but only about 2% of vendors managed to sell more than $100,000. In fact, 35 sellers were observed selling over $1,000,000 worth of product and the top 1% most successful vendors were responsible for 51.5% of all the volume transacted. Some of these sellers, like "SuperTrips" (or to a lesser extent, "Nod") from Silk Road, have been arrested, and numbers released in connection with these arrests are consistent with our findings [4, 6].

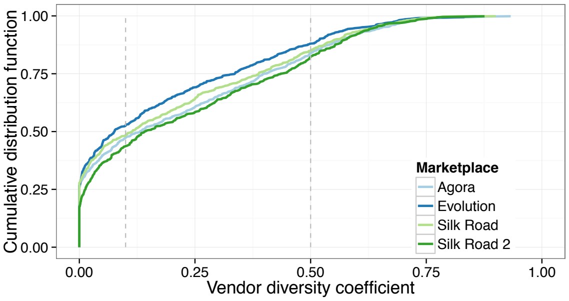


Figure 12: **Vendor diversity**

There is a clear discrepancy between sellers that experiment in the marketplaces and those who manage to leverage it to operate a successful business. Going forward, we define any seller that we have observed selling in excess of $10,000 to be successful. This allows us to draw conclusions only about vendors that have had a meaningful impact on the marketplace ecosystem. Now that we know how much sellers are selling, we wish to understand *what* they are selling. Once again we group feedback by vendor but this time we also use the classifier to categorize the items that were being sold and aggregate by category. Let $\mathscr{C}$ be the set of normalized item categories for each seller and $\mathscr{S}$ be the set of all sellers across all marketplaces. So, $|\mathscr{C}| = 16$, and $|\mathscr{S}| = 9,386$. Define $\mathscr{C}_i(s_j)$ as the normalized value of the i-th category for seller j such that $\forall s_j \in S,\ \sum_{i=1}^{|\mathscr{C}|} \mathscr{C}_i(s_j) = 1$. Then, we define the coefficient of diversity for a seller s_j as:

$$c_d = \left(1 - \max_i \left(\mathscr{C}_i(s_j)\right)\right) \frac{|\mathscr{C}|}{|\mathscr{C}| - 1} \ .$$

Intuitively, the coefficient of diversity is measuring how invested a seller is into their most popular category, normalized so that $c_d \in [0, 1]$. When evaluating the categories that different sellers are invested in, it only makes sense to consider successful sellers as less significant sellers are volatile and greatly influenced by an individual sale in some category.

Figure 12 plots the CDF of the coefficient of diversity for sellers from Evolution, Silk Road, Silk Road 2 and Agora that sold more than $10,000 total. From Figure 12 we argue that there are roughly three types of sellers. The first type of seller with a coefficient of diversity between 0 and 0.1 is highly specialized, and sells exactly one type of product. About half of all sellers are highly specialized and indicates that the seller has access to a steady long-term supply of some type of product. About one third of all vendors who specialize sell cannabis, another third sell digital goods, and the last third sell in the various other categories. While digital goods is a relatively small share of the total marketplace ecosystem, it tends to attract vendors that specialize. This is likely due to the domain expertise required for actions such as manufacturing fake IDs or stealing credit cards. The second type of seller has a diversity coefficient of between 0.1 and 0.5 and generally specializes in two or three types of products. The most common two categories to simultaneously specialize in are ecstasy and psychedelics – i.e., primarily recreational and club drugs. The third type of vendor has a diversity coefficient greater than 0.5 and has no specialty but rather sells a variety of items. These types of sellers may be networks of users with access to many different sources, or may be involved in arbitrage between markets.

PGP deployment We conclude our discussion of vendor behavior by looking in more detail at their security practices. While we cannot easily assess their overall operational security, we consider a very simple proxy for security behavior: the availability of a valid PGP key. From our data set, we extracted 7,717 PGP keys. Most vendors use keys of appropriate length, even though we did observe a couple of oddities (e.g., a 2,047-bit key!) that might indicate an incorrect use of the software. Inspired by Heninger et al. [20] and Lenstra et al. [25] we checked all pairs of keys to determine whether or not they had common primes. We did not find any, which either suggests that GPG software was always properly used and with a good random number generator, or, more likely, that our dataset is too small to contain evidence of weak keys.

We then plot in Figure 13 the fraction of vendors, over time, that have (at least) one usable PGP key. We take an extremely inclusive view of PGP deployment here: as long as a vendor has advertised a valid PGP key for one or her active aliases, we consider they are using PGP. As vendors deal with highly sensitive information such as postal delivery addresses of their customers, we would expect close to 100% deployment. We see that, despite improvements, this is not the case. In the original Silk Road, only approximately 2/3 to 3/4 of vendors had a valid PGP key listed. During the upheaval of the 2013 Fall, with many marketplaces opening and shutting down quickly, we see that PGP deployment is very low. When the situation stabilizes in January 2014, we observe an increase in PGP adoption; interestingly, *after* Operation Onymous, adoption seems even higher, which can be construed as an evolutionary argument: marketplaces that support and encourage PGP use by their sellers (such as Evolution and Agora) might have been also more secure in other respects, and more resilient against takedowns. Shortly before the Evolution shutdown, PGP deployment on Agora and Evolution was close to 90%.

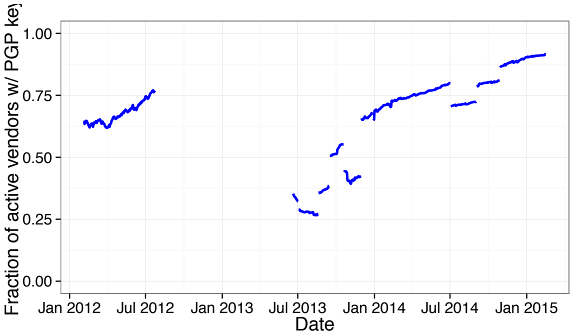


Figure 13: **PGP deployment over time.**

5 Discussion

A study of this kind brings up a number of important discussion points. We focus here on what we consider are the most salient ones: validation, ethics, and potential public policy take-aways.

5.1 Validation

Scientific measurements should be amenable to validation. Unfortunately, here, ground truth is rarely available, which in turn makes validation extremely difficult. Marketplace operators indeed generally do not publish metrics such as seller numbers or traffic volumes. However, in certain cases, we have limited information that we can use for spot-checking estimates.

Ross Ulbricht trial evidence (Silk Road) In October 2013, a San Francisco man by the name of Ross Ulbricht was arrested and charged as being the operator of Silk Road [8]. A large amount of data was subsequently entered into evidence used during his trial, which took place in January 2015. In particular, evidence contained relatively detailed accounting entries found on Mr. Ulbricht's laptop, and claimed to pertain to Silk Road. Chat transcripts (evidence GX226A, GX227C) place weekly volumes at $475,000/week in late March 2012 for instance: this is consistent with the data previously reported [13] and which we use for documenting Silk Road 1. Evidence GX250 contains a personal ledger which apparently faithfully documents Silk Road sales commissions. Projecting the data listed during the time of the previous study [13] ($680,279) over a year yields a yearly projection of about $1.2M; Christin's estimates were of $1.1M [13]. This hints that the technique of using feedback as a sales proxy, which we reuse here, produces reliable estimates.

Blake Benthall criminal complaint (Silk Road 2) In November 2014, another San Francisco man by the name of Blake Benthall was arrested and charged with being "Defcon," the Silk Road 2.0 administrator. The criminal complaint against Mr. Benthall [7] reports that in September 2014, the administrator, talking to an undercover agent actually working on Silk Road 2's staff, reports around $6M of monthly sales; and later amends this number to $8M. This corresponds to a daily sales volume of $200,000–$250,000 which is very close to what we report in Figure 5 for Silk Road 2 at that given time.

Leaked Agora seller page In December 2014, it was revealed that an Agora vendor page had been scraped and leaked on Pastebin [21]. This vendor page in particular contains a subset of all the vendor's transactions; one can estimate precisely the amount for that specific vendor on June 5, 2014 to $3,460. Checking in our database, our instantaneous estimate credits that seller with $3,408 on the day – which, considering Bitcoin exchange fluctuations is pretty much identical to the ground truth.

5.2 Ethics of data collection

We share much of the ethical concerns and views documented in previous work [13]. Our data collection, in particular, is massive, and could potentially put some strain on the Tor network, not to mention marketplace servers themselves. However, even though it is hard to assess we believe that our measurements represent a small fraction of all traffic that is going to online anonymous marketplaces. As discussed in Section 3 we are attempting to balance accuracy of the data collection with a light-weight enough crawling strategy to avoid detection – or worse, impacting the very operations we are trying to measure. In addition, we are contributing Tor relays with long uptimes on very fast networks to "compensate" for our own massive use of the network. Our work takes a number of steps to remain neutral. We certainly do not want to facilitate vendor or marketplace operator arrests. This is not just an ethical question, but is also a scientific one: our measurements, to be sound, should not impact the subject(s) being measured [23].

5.3 Public-policy take-aways

The main outcome of this work, we hope, is a critical evaluation of meaningful public policy toward online anonymous marketplaces. While members of Congress have routinely called for the take down of "brazen" online marketplaces, it is unclear that this is the most pragmatic use of taxpayer money.

In fact, our measurements suggest that the ecosystem appears quite resilient to law enforcement take-downs. We see this without ambiguity in response to the (original) Silk Road take-down; and while it is too early to tell the long-lasting impacts of Operation Onymous, its main effect so far seems to have been to consolidate transactions in the two dominant marketplaces at the time of the take-down. More generally, economics tell us that because user demand for drugs online is present (and quite massive), enterprising individuals will seemingly always be interested in accommodating this demand.

A natural question is whether the cat-and-mouse game between law enforcement and marketplace operators could end with the complete demise of online anonymous marketplaces. Our results suggest it is unlikely. Thus, considering the expenses incurred in very lengthy investigations and the level of international coordination needed in operations like Operation Onymous, the time may be ripe to investigate alternative solutions.

Reducing demand through prevention is certainly an alternative worth exploring on a global public policy level, but, from a law enforcement perspective, even active intervention could be much more targeted, e.g., toward seizing highly dangerous products while in transit. A number of documented successes in using traditional police work against sellers of hazardous substances (e.g., [35]) or large-scale dealers (e.g., [4, 6] among many others) show that law enforcement is not powerless to address the issue in the physical world.

6 Related work

The past decade has seen a large number of detailed research efforts aiming at gathering actual measurements from various online criminal ecosystems in order to devise meaningful defenses; see, e.g., [13, 14, 22, 26, 27, 28, 29, 32, 40, 41]. Anderson et al. [11] and Thomas et al. [37] provide a very good overview of the field. Closest among these papers to our work, McCoy et al. obtained detailed measurements of online pharmaceutical affiliates, showing that individual networks grossed between USD 12.8 million/year to USD 67.7 million/year. In comparison, the long-term rough average we see here is in the order of $150–180M/year for the entire online anonymous marketplace ecosystem. In other words, online marketplaces have seemingly surpassed more "traditional" ways of delivering illicit narcotics.

With respect to specific measurements of online anonymous marketplaces, the present paper builds up on our previous work [13]. Surprisingly few other efforts exist attempting to quantitatively characterize the economics of online anonymous marketplaces. Of note, Aldridge and Décary-Hétu [10] complement our original volume estimates by showing revised numbers of around $90M/year for Silk Road in 2013 right before its take-down. This is roughly in line with our own measurements, albeit slightly more conservative (Figure 5 shows about $300K/day for Silk Road in summer 2013.) More recent work by Dolliver [17] tries to assess the volumes on Silk Road 2.0. While she does not report volumes, her seller numbers are far smaller than ours, and we suspect her scrapes might have been incomplete. Looking at the problem from a different angle, Meiklejohn et al. [31] provide a detailed analysis of transaction traceability in the Bitcoin network, and show which addresses are related to Silk Road, which in turn could be a useful way of assessing the total volumes of that marketplace. A follow up paper [30] shows that purported Bitcoin "anonymity" (i.e., unlinkability) is greatly overstated, even when using newer mixing primitives.

On the customer side, Barratt et al. [12] provide an insightful survey of Silk Road patrons, showing that a lot of them associate with the "party culture," which is corroborated by our results showing that cannabis and ecstasy correspond to roughly half of the sales; likewise Van Hout and Bingham provide valuable insights into individual participants [39]. Our research complements these efforts by providing a macro-level view of the ecosystem.

7 Conclusions

Even though anonymous online marketplaces are a relatively recent development in the overall online crime ecosystem, our longitudinal measurements show that in the short four years since the development of the original Silk Road, total volumes have reached up to $700,000 daily (averaged over 30-day windows) and are generally stable around $300,000-$500,000 a day, far exceeding what had been previously reported. More remarkably, anonymous marketplaces are extremely resilient to take-downs and scams – highlighting the simple fact that economics (demand) plays a dominant role. In light of our findings, we suggest a re-evaluation of intervention policies against anonymous marketplaces. Given the high demand for the products being sold, it is not clear that take-downs will be effective; at least we have found no evidence they were. Even if one went to the impractical extreme of banning anonymous networks, demand would probably simply move to other channels, while some of the benefits associated with these markets (e.g., reduction in risks of violence at the retail level) would be lost. Instead, a focus on reducing consumer demand, e.g., through prevention, might be worth considering; likewise, it would be well-worth investigating whether more targeted interventions (e.g., at the seller level) have had measurable effects on the overall ecosystem. While our paper does not answer these questions, we believe that the data collection methodology we described, as well as some of the data we have collected, may enable further research in the field.

Acknowledgments

This research was partially supported by the National Science Foundation under ITR award CCF-0424422 (TRUST) and SaTC award CNS-1223762; and by the Department of Homeland Security Science and Technology Directorate, Cyber Security Division (DHS S&T/CSD), the Government of Australia and SPAWAR Systems Center Pacific via contract number N66001-13-C-0131. This paper represents the position of the authors and not that of the aforementioned agencies. We thank our anonymous reviewers and our shepherd, Damon McCoy, for feedback that greatly improved the manuscript.

References

[1] Darknet stats. https://dnstats.net/.

[2] Grams: Search the darknet. http://grams7enufi7jmdl.onion.

[3] Scrapy: An open source web scraping framework for Python. http://scrapy.org.

[4] United States of America vs. Steven Lloyd Sadler and Jenna M. White, Nov. 2013. United States District Court, Western District of Washington at Seattle. Criminal Complaint MJ13-487.

[5] Silk Road 2.0 'hack' blamed on Bitcoin bug, all funds stolen, Feb. 2014. http://www.forbes.com/sites/andygreenberg/2014/02/13/silk-road-2-0-hacked-using-bitcoin-bug-all-its-funds-stolen/.

[6] Silk Road online drug dealer pleads guilty to trafficking, May 2014. http://www.cbsnews.com/news/silk-road-online-drug-dealer-pleads- guilty-to-trafficking/.

[7] United States of America vs. Blake Benthall, Oct. 2014. United States District Court, Southern District of New York. Sealed Complaint 14MAG2427.

[8] United States of America vs. Ross William Ulbricht, Feb. 2014. United States District Court, Southern District of New York. Indictment 14CRIM068.

[9] Bitcoin "exit scam": deep-web market operators disappear with $12m, Mar. 2015. http://www.theguardian.com/technology/2015/mar/18/bitcoin-deep-web-evolution-exit-scam-12-million-dollars/.

[10] Aldridge, J., and Décary-Hétu, D. Not an "Ebay for drugs": The cryptomarket "Silk Road" as a paradigm shifting criminal innovation. *Available at SSRN 2436643* (2014).

[11] Anderson, R., Barton, C., Böhme, R., Clayton, R., Van Eeten, M. J., Levi, M., Moore, T., and Savage, S. Measuring the cost of cybercrime. In *The economics of information security and privacy*. Springer, 2013, pp. 265–300.

[12] Barratt, M. J., Ferris, J. A., and Winstock, A. R. Use of silk road, the online drug marketplace, in the united kingdom, australia and the united states. *Addiction 109*, 5 (2014), 774–783.

[13] Christin, N. Traveling the Silk Road: A measurement analysis of a large anonymous online marketplace. In *Proceedings of the 22nd World Wide Web Conference (WWW'13)* (Rio de Janeiro, Brazil, May 2013), pp. 213–224.

[14] Christin, N., Yanagihara, S., and Kamataki, K. Dissecting one click frauds. In *Proc. ACM CCS'10* (Chicago, IL, Oct. 2010).

[15] Digital Citizens Alliance. Busted, but not broken: The state of Silk Road and the darknet marketplaces, Apr. 2014.

[16] Dingledine, R., Mathewson, N., and Syverson, P. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium* (San Diego, CA, Aug. 2004).

[17] Dolliver, D. Evaluating drug trafficking on the Tor network: Silk Road 2, the sequel. *International Journal of Drug Policy* (2015).

[18] Greenberg, A. An interview with a digital drug lord: The Silk Road's Dread Pirate Roberts (Q&A), Aug. 2013. `http://www.forbes.com/sites/andygreenberg/2013/08/14/an-interview-with-a-digital-drug-lord-the-silk-roads-dread-pirate-roberts-qa/`.

[19] Greenberg, A. Five men arrested in dutch crackdown on Silk Road copycat, Feb. 2014. `http://www.forbes.com/sites/andygreenberg/2014/02/12/five-men-arrested-in-dutch-crackdown-on-silk-road-copycat/`.

[20] Heninger, N., Durumeric, Z., Wustrow, E., and Halderman, J. A. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *Proceedings of the 21st USENIX Security Symposium* (Bellevue, WA, Aug. 2012).

[21] Impost_r. Boosie5150 questionable security practices - Agora account compromised in june. `https://www.reddit.com/r/DarkNetMarkets/comments/2oisq0/boosie5150_questionable_security_practices_agora/`.

[22] John, J., Yu, F., Xie, Y., Abadi, M., and Krishnamurthy, A. deSEO: Combating search-result poisoning. In *Proceedings of USENIX Security 2011* (San Francisco, CA, Aug. 2011).

[23] Kanich, C., Levchenko, K., Enright, B., Voelker, G., and Savage, S. The Heisenbot uncertainty problem: challenges in separating bots from chaff. In *Proceedings of USENIX LEET'08* (San Francisco, CA, Apr. 2008).

[24] Kaplan, E., and Meier, P. Nonparametric estimation from incomplete observations. *Journal of the American Statistical Association 53* (1958), 457–481.

[25] Lenstra, A., Hughes, J. P., Augier, M., Bos, J. W., Kleinjung, T., and Wachter, C. Ron was wrong, Whit is right. Tech. rep., IACR, 2012.

[26] Levchenko, K., Chachra, N., Enright, B., Felegyhazi, M., Grier, C., Halvorson, T., Kanich, C., Kreibich, C., Liu, H., McCoy, D., Pitsillidis, A., Weaver, N., Paxson, V., Voelker, G., and Savage, S. Click trajectories: End-to-end analysis of the spam value chain. In *Proceedings of IEEE Security and Privacy* (Oakland, CA, May 2011).

[27] Li, Z., Alrwais, S., Wang, X., and Alowaisheq, E. Hunting the red fox online: Understanding and detection of mass redirect-script injections. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (Oakland'14)* (San Jose, CA, May 2014).

[28] Lu, L., Perdisci, R., and Lee, W. SURF: Detecting and measuring search poisoning. In *Proceedings of ACM CCS 2011* (Chicago, IL, Oct. 2011).

[29] McCoy, D., Pitsillidis, A., Jordan, G., Weaver, N., Kreibich, C., Krebs, B., Voelker, G., Savage, S., and Levchenko, K. Pharmaleaks: Understanding the business of online pharmaceutical affiliate programs. In *Proceedings of USENIX Security 2012* (Bellevue, WA, Aug. 2012).

[30] Meiklejohn, S., and Orlandi, C. Privacy-enhancing overlays in bitcoin. In *Proceedings of the 2015 BITCOIN research workshop* (Puerto Rico, Jan. 2015).

[31] Meiklejohn, S., Pomarole, M., Jordan, G., Levchenko, K., McCoy, D., Voelker, G. M., and Savage, S. A fistful of bitcoins: characterizing payments among men with no names. In *Proceedings of the ACM/USENIX Internet measurement conference* (Barcelona, Spain, Oct. 2013), pp. 127–140.

[32] Moore, T., Leontiadis, N., and Christin, N. Fashion crimes: Trending-term exploitation on the web. In *Proceedings of ACM CCS 2011* (Chicago, IL, Oct. 2011).

[33] Nakamoto, S. Bitcoin: a peer-to-peer electronic cash system, Oct. 2008. Available from `http://bitcoin.org/bitcoin.pdf`.

[34] Sankin, A. Sheep marketplace scam reveals everything that's wrong with the deep web, Dec. 2013. `http://www.dailydot.com/crime/sheep-marketplace-scam-shut-down/`.

[35] Sterbenz, C. 20-year-old gets 9 years in prison for trying to poison people all over the world, Feb. 2014. `http://www.businessinsider.com/r-florida-man-gets-nine-years-prison -in-new-jersey-over-global-poison-plot-2015-2`.

[36] Sutherland, W. J. *Ecological Census Techniques: A Handbook*. Cambridge University Press, 1996.

[37] Thomas, K., Huang, D., Wang, D., Bursztein, E., Grier, C., Holt, T., Kruegel, C., McCoy, D., Savage, S., and Vigna, G. Framing dependencies introduced by underground commoditization. In *Proceedings (online) of the Workshop on Economics of Information Security (WEIS)* (June 2015).

[38] U.S. Attorney's Office, Southern District of New York. Dozens of online "dark markets" seized pursuant to forfeiture complaint filed in Manhattan federal court in conjunction with the arrest of the operator of Silk Road 2.0, Nov. 2014. `http://www.justice.gov/usao/nys/pressreleases/November14/DarkMarketTakedown.php`.

[39] Van Hout, M. C., and Bingham, T. silk road, the virtual drug marketplace: A single case study of user experiences. *International Journal of Drug Policy 24*, 5 (2013), 385–391.

[40] Wang, D., Der, M., Karami, M., Saul, L., McCoy, D., Savage, S., and Voelker, G. Search + seizure: The effectiveness of interventions on seo campaigns. In *Proceedings of ACM IMC'14* (Vancouver, BC, Canada, Nov. 2014).

[41] Wang, D., Voelker, G., and Savage, S. Juice: A longitudinal study of an SEO botnet. In *Proceedings of NDSS'13* (San Diego, CA, Feb. 2013).

Under-Constrained Symbolic Execution: Correctness Checking for Real Code

David A. Ramos
ramos@cs.stanford.edu

Dawson Engler
engler@csl.stanford.edu

Stanford University

Abstract

Software bugs are a well-known source of security vulnerabilities. One technique for finding bugs, symbolic execution, considers all possible inputs to a program but suffers from scalability limitations. This paper uses a variant, *under-constrained* symbolic execution, that improves scalability by directly checking individual functions, rather than whole programs. We present UC-KLEE, a novel, scalable framework for checking C/C++ systems code, along with two use cases. First, we use UC-KLEE to check whether patches introduce crashes. We check over 800 patches from BIND and OpenSSL and find 12 bugs, including two OpenSSL denial-of-service vulnerabilities. We also verify (with caveats) that 115 patches do not introduce crashes. Second, we use UC-KLEE as a generalized checking framework and implement checkers to find memory leaks, uninitialized data, and unsafe user input. We evaluate the checkers on over 20,000 functions from BIND, OpenSSL, and the Linux kernel, find 67 bugs, and verify that hundreds of functions are leak free and that thousands of functions do not access uninitialized data.

1 Introduction

Software bugs pervade every level of the modern software stack, degrading both stability and security. Current practice attempts to address this challenge through a variety of techniques, including code reviews, higher-level programming languages, testing, and static analysis. While these practices prevent many bugs from being released to the public, significant gaps remain.

One technique, testing, is a useful sanity check for code correctness, but it typically exercises only a small number of execution paths, each with a single set of input values. Consequently, it misses bugs that are only triggered by other inputs.

Another broad technique, static analysis, is effective at discovering many classes of bugs. However, static analysis generally uses abstraction to improve scalability and cannot reason precisely about program values and pointer relationships. Consequently, static tools often miss deep bugs that depend on specific input values.

One promising technique that addresses the limitations of both testing and static analysis is symbolic execution [4, 5, 40]. A symbolic execution tool conceptually explores all possible execution paths through a program in a bit-precise manner and considers all possible input values. Along each path, the tool determines whether any combination of inputs could cause the program to crash. If so, it reports an error to the developer, along with a concrete set of inputs that will trigger the bug.

Unfortunately, symbolic execution suffers from the well-known *path explosion* problem since the number of distinct execution paths through a program is often exponential in the number of if-statements or, in the worst case, infinite. Consequently, while symbolic execution often examines orders of magnitude more paths than traditional testing, it typically fails to exhaust all interesting paths. In particular, it often fails to reach code deep within a program due to complexities earlier in the program. Even when the tool succeeds in reaching deep code, it considers only the input values satisfying the few paths that manage to reach this code.

An alternative to whole-program symbolic execution is *under-constrained* symbolic execution [18, 42, 43], which directly executes an arbitrary function within the program, effectively *skipping* the costly path prefix from `main` to this function. This approach reduces the number and length of execution paths that must be explored. In addition, it allows library and OS kernel code without a `main` function to be checked easily and thoroughly.

This paper presents UC-KLEE, a scalable framework implementing under-constrained symbolic execution for C/C++ systems code without requiring a manual specification or even a single testcase. We apply this framework to two important use cases. First, we use it to check whether patches to a function introduce new bugs, which may or may not pose security vulnerabilities. Ironically, patches intended to fix bugs or eliminate security vulnerabilities are a frequent source of them. In many cases, UC-KLEE can *verify* (up to a given input bound and with standard caveats) that a patch does not introduce new crashes to a function, a guarantee not possible with existing techniques.

Second, we use UC-KLEE as a general code checking framework upon which specific checkers can be implemented. We describe three example checkers we implemented to find memory leaks, uses of uninitialized data, and unsanitized uses of user input, all of which *may* pose security vulnerabilities. Additional checkers may be added to our framework to detect a wide variety of bugs along symbolic, bit-precise execution paths through functions deep within a program. If UC-KLEE exhaustively checks all execution paths through a function, then it has effectively verified (with caveats) that the function passes the check (e.g., no leaks).

We evaluated these use cases on large, mature, and security-critical code. We validated over 800 patches from BIND [3] and OpenSSL [36] and found 12 bugs, including two OpenSSL denial-of-service vulnerabilities [12, 16]. UC-KLEE verified that 115 patches did not introduce new crashes, and it checked thousands of paths and achieved high coverage even on patches for which it did not exhaust all execution paths.

We applied our three built-in checkers to over 20,000 functions from BIND, OpenSSL, and the Linux kernel and discovered 67 new bugs, several of which appear to be remotely exploitable. Many of these were latent bugs that had been missed by years of debugging effort. UC-KLEE also exhaustively verified (with caveats) that 771 functions from BIND and OpenSSL that allocate heap memory do not cause memory leaks, and that 4,088 functions do not access uninitialized data.

The remainder of this paper is structured as follows: § 2 presents an overview of under-constrained symbolic execution; § 3 and § 4 discuss using UC-KLEE for validating patches and generalized checking, respectively; § 5 describes implementation tricks; § 6 discusses related work; and § 7 concludes.

2 Overview

This paper builds upon our earlier work on UC-KLEE [43], an extension to the KLEE symbolic virtual machine [5] designed to support equivalence verification and under-constrained symbolic inputs. Our tool checks C/C++ code compiled as *bitcode* (intermediate representation) by the LLVM compiler [29]. As in KLEE, it performs bit-accurate symbolic execution of the LLVM bitcode, and it executes any functions called by the code. Unlike KLEE, UC-KLEE begins executing code at an arbitrary function chosen by the user, rather than `main`.

With caveats (described in § 2.2), UC-KLEE provides verification guarantees on a per-path basis. If it exhausts all execution paths, then it has verified that a function has the checked property (e.g. that a patch does not introduce any crashes or that the function does not leak memory) up to the given input size.

Directly invoking functions within a program presents new challenges. Traditional symbolic execution tools generate input values that represent external input sources (e.g., command-line arguments, files, etc.). In most cases, a correct program should reject invalid external inputs rather than crash. By contrast, individual functions typically have *preconditions* imposed on their inputs. For example, a function may require that pointer arguments be non-null. Because UC-KLEE directly executes functions without requiring their preconditions to be specified by the user, the inputs it considers may be a superset (over-approximation) of the legal values handled by the function. Consequently, we denote UC-KLEE's symbolic inputs as *under-constrained* to reflect that they are missing preconditions (constraints).

While this technique allows previously-unreachable code to be deeply checked, the missing preconditions may cause *false positives* (spurious errors) to be reported to the user. UC-KLEE provides both automated heuristics and an interface for users to manually silence these errors by lazily specifying input preconditions using simple C code. In our experience, even simple annotations may silence a large number of spurious errors (see § 3.2.5) and this effort is orders of magnitude less work than eagerly providing a full specification for each function.

2.1 Lazy initialization

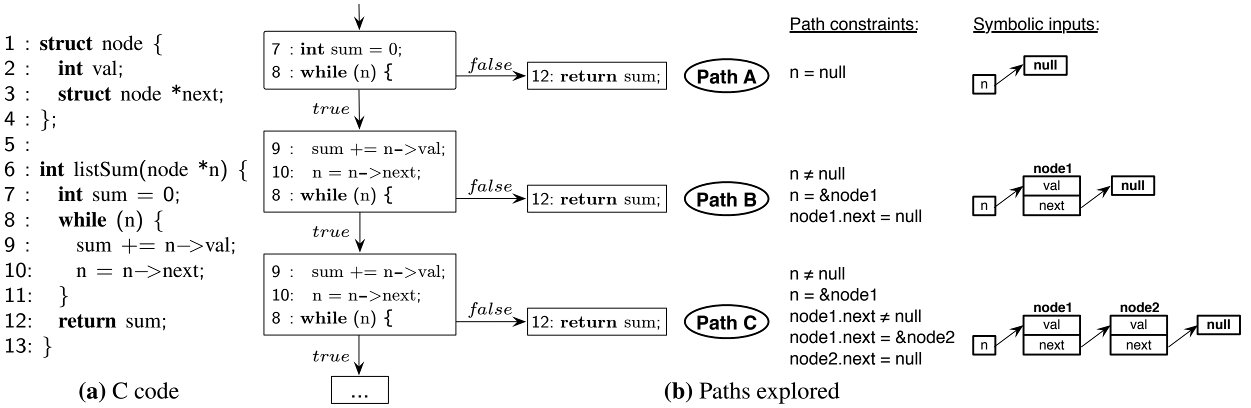
UC-KLEE automatically generates a function's symbolic inputs using lazy initialization [26, 46], which avoids the need for users to manually construct inputs, even for complex, pointer-rich data structures. We illustrate lazy initialization by explaining how UC-KLEE executes the example function `listSum` in Figure 1(a), which sums the entries in a linked list. Figure 1(b) summarizes the three execution paths we explore. For clarity, we elide error checks that UC-KLEE normally performs at memory accesses, division/remainder operations, and assertions.

UC-KLEE first creates an under-constrained symbolic value to represent the sole argument `n`. Although `n` is a pointer, it begins in the *unbound* state, not yet pointing to any object. UC-KLEE then passes this symbolic argument to `listSum` and executes as follows:

Line 7 The local variable `sum` is assigned a concrete value; no special action is taken.

Line 8 The code checks whether the symbolic variable `n` is non-null. At this point, UC-KLEE forks execution and considers both cases. We first consider the false path where $n = null$, (Path A). We then return to the true path where $n \neq null$ (Path B). On Path A, UC-KLEE adds $n = null$ as a path constraint and skips the loop.

Line 12 Path A returns `0` and terminates.



```
1 : struct node {
2 :   int val;
3 :   struct node *next;
4 : };
5 :
6 : int listSum(node *n) {
7 :   int sum = 0;
8 :   while (n) {
9 :     sum += n->val;
10:     n = n->next;
11:   }
12:   return sum;
13: }
```

(a) C code

(b) Paths explored

Figure 1: Example code fragment analyzed by UC-KLEE.

We now consider Path B.

Line 8 UC-KLEE adds the constraint $n \neq null$ and enters the loop.

Line 9 The code dereferences the pointer `n` for the first time on Path B. Because `n` is *unbound*, UC-KLEE allocates a new block of memory, denoted $node1$, to satisfy the dereference and adds the constraint $n = \&node1$ to *bind* the pointer `n` to this object. At this point, `n` is no longer unbound, so subsequent dereferences of that pointer will resolve to $node1$ rather than trigger additional allocations. The (symbolic) contents of $node1$ are marked as unbound, allowing future dereferences of pointers in this object to trigger allocations. This recursive process is the key to lazy initialization. Next, `sum` is incremented by the symbolic value $node1.val$.

Line 10 `n` is set to the value $node1.next$. Path B then returns to the loop header.

Line 8 The code tests whether `n` (set to $node1.next$) is non-null. UC-KLEE forks execution and considers both cases. We first consider $node1.next = null$, which we still refer to as Path B. We will then return to the true path where $node1.next \neq null$ (Path C). On Path B, $node1.next = null$ is added as a path constraint and execution exits the loop.

Line 12 Path B returns $node1.val$ and terminates.

We now consider Path C.

Line 8 UC-KLEE adds $node1.next \neq null$ as a path constraint, and Path C enters the loop.

Line 9 Path C dereferences the unbound symbolic pointer $node1.next$, which triggers allocation of a new object $node2$. This step illustrates the unbounded nature of many loops. To prevent UC-KLEE from allocating an unbounded number of objects as input, the tool accepts a command-line option to limit the depth of an input-derived data structure (*k*-bounding [17]). When a path attempts to exceed this limit, our tool silently terminates it. For this example, assume a depth limit of two, which causes UC-KLEE to terminate Path D (not shown) at line 9 during the next loop iteration.

Line 10 `n` is set to the value $node2.next$.

Line 8 UC-KLEE forks execution and adds the path constraint $node2.next = null$ to Path C.

Line 12 Path C returns $node1.val + node2.val$ and exits.

This example illustrates a simple but powerful recursive technique to automatically synthesize data structures from under-constrained symbolic input. Figure 2 shows an actual data structure our tool generated as input for one of the BIND bugs we discovered (Figure 5). The edges between each object are labeled with the field names contained in the function's debug information and included in UC-KLEE's error report.

2.2 Limitations

Because we build on our earlier version of UC-KLEE, we inherit its limitations [43]. The more important examples are as follows. The tool tests compiled code on a specific platform and does not consider other build configurations. It does not handle assembly (see § 4 for how we skip inline assembly), nor symbolic floating point operations. In addition, there is an explicit assumption that input-derived pointers reference unique objects (no aliasing, and no cyclical data structures), and the tool assigns distinct concrete addresses to allocated objects.

When checking whether patches introduce bugs, UC-KLEE aims to detect crashing bugs and does not look for performance bugs, differences in system call arguments, or concurrency errors. We can only check patches that do not add, remove, or reorder fields in data structures or change the type signatures of patched functions. We plan to extend UC-KLEE to support such patches by implementing a type map that supplies identical inputs to each version of a function in a "field aware" manner. How-

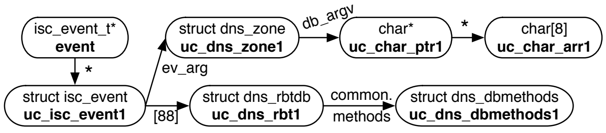


Figure 2: BIND data structure allocated by UC-KLEE.

ever, our current system does not support this, and we excluded such patches from our experiments.

3 Patch checking

To check whether a patch introduces new crashing bugs, UC-KLEE symbolically executes two compiled versions of a function: P, the unpatched version, and P', the patched version. If it finds any execution paths along which P' crashes but P does not (when given the same symbolic inputs), it reports a potential bug in the patch.

Recall that due to missing input preconditions, we cannot simply assume that all crashes are bugs. Instead, UC-KLEE looks for paths that exhibit *differing* crash behavior between P and P', which usually share an identical set of preconditions. Even if UC-KLEE does not know these preconditions, in practice, real code tends to show *error equivalence* [43], meaning that P and P' both crash (or neither crashes) on illegal inputs. For example, if a precondition requires a pointer to be non-null and both versions dereference the pointer, then P and P' will both crash when fed a null pointer as an argument.

In prior work, UC-KLEE [43] verified the equivalence of small library routines, both in terms of crashes and outputs. While detecting differences in functionality may point to interesting bugs, these discrepancies are typically meaningful only to developers of the checked code. Because this paper evaluates our framework on large, complex systems developed by third parties, we limit our discussion to crashes, which objectively point to bugs.

To check patches, UC-KLEE automatically generates a *test harness* that sets up the under-constrained inputs and invokes P and P'. Figure 3 shows a representative test harness. Lines 2–3 create an under-constrained input `n`. Line 4 calls `fooB` (P'). Note that UC-KLEE invokes P' before P to facilitate path pruning (§ 3.1). Line 5 discards any writes performed by `fooB` but preserves the path constraints so that `fooA` (P) will see the same initial memory contents and follow the corresponding path. Line 6 invokes `fooA`.

```
1 : int main() {
2 :    node *n;
3 :    ucklee_make_uc(&n);
4 :    fooB(n); /* run P' */
5 :    ucklee_reset_address_space();
6 :    fooA(n); /* run P */
7 :    return 0;
8 : }
```

Figure 3: Test harness.

If a path through `fooB` crashes, UC-KLEE unwinds the stack and resumes execution at line 5. If `fooA` also crashes on this path, then the two functions are crash equivalent and no error is reported. However, if `fooA` returns from line 6 *without* crashing, we report an error to the user as a possible bug in `fooB`. For this use case, we do not report errors in which `fooA` (P) crashes but `fooB` (P') does not, which suggest bugs *fixed* by a patch.

3.1 Path pruning

UC-KLEE employs several path pruning techniques to target errors and avoid uninteresting paths. The underlying UC-KLEE system includes a *static cross-checker* that walks over the LLVM [29] control flow graph, conservatively marking regions of basic blocks that differ between the original function P and the patched function P'. This algorithm is fairly straightforward, and we elide details for brevity. UC-KLEE soundly prunes paths that:

1. have never executed a "differing" basic block, and
2. cannot reach a differing basic block from their current program counter and call stack.

The second condition uses an inter-procedural reachability analysis from the baseline UC-KLEE system. Paths meeting both of these criteria are safe to prune because they will execute identical instruction sequences.

In addition, UC-KLEE introduces pruning techniques aimed specifically at detecting errors introduced by a patch. As our system executes P' (`fooB` in Figure 3), it prunes paths that either:

1. return from P' without triggering an error, or
2. trigger an error without reaching differing blocks.

In the first case, we are only concerned with errors *introduced* by the patch. In the second case, P and P' would both trigger the error.

Error uniquing. Our system aggressively uniques errors by associating each path executing P with the program counter (PC) of the error that occurred in P'. Once our system executes a non-error path that returns from P (and reports the error in P'), it prunes all current and future paths that hit the same error (PC and type) in P'. In practice, this enabled our system to prune thousands of redundant error paths.

3.2 Evaluation

We evaluated UC-KLEE on hundreds of patches from BIND and OpenSSL, two widely-used, security critical systems. Each codebase contains about 400,000 lines of C code, making them reasonable measures of UC-KLEE's scalability and robustness. For this experiment, we used a maximum symbolic object size of 25,000 bytes and a maximum symbolic data structure depth of 9 objects.

3.2.1 Patch selection and code modifications

We tried to avoid selection bias by using two complete sets of patches from the `git` repositories for recent stable branches: BIND 9.9 from 1/2013 to 3/2014 and OpenSSL 1.0.1 from 1/2012 to 4/2014. Many of the patches we encountered modified more than one function; this section uses *patch* to refer to changes to a single function, and *commit* to refer to a complete changeset.

We excluded all patches that: only changed copyright information, had build errors, modified build infrastructure only, removed dead functions only, applied only to disabled features (e.g., win32), patched only BIND `contrib` features, only touched regression/unit tests, or used variadic functions. We also eliminated all patches

Codebase	Function	Type	Cause	New	Vulnerability
BIND	`receive_secure_db`	assert fail	double lock acquisition	✓	
BIND	`save_nsec3param`	assert fail	uninitialized struct	✓	
BIND	`configure_zone_acl`	assert fail	inconsistent null argument handling	✓	
BIND	`isc_lex_gettoken`	assert fail	input parsing logic	✓	
OpenSSL	`PKCS5_PBKDF2_HMAC`	uninitialized pointer dereference	uninitialized struct		
OpenSSL	`dtls1_process_record`	assert fail	inconsistent null check		
OpenSSL	`tls1_final_finish_mac`	null pointer dereference	unchecked return value	✓	
OpenSSL	`do_ssl3_write`	null pointer dereference	callee side effect after null check	✓	CVE-2014-0198
OpenSSL	`PKCS7_dataDecode`	null pointer dereference	unchecked return value	✓	
OpenSSL	`EVP_DecodeUpdate`	out-of-bounds array access	negative count passed to memcpy	✓	CVE-2015-0292
OpenSSL	`dtls1_buffer_record`	use-after-free	improper error handling	✓	
OpenSSL	`pkey_ctrl_gost`	uninitialized pointer dereference	improper error handling	✓	

Figure 4: Summary of bugs UC-KLEE reported while checking patches. *New* indicates that the bug was previously unknown.

that yielded identical code after compiler optimizations. Because of tool limitations, we excluded patches that changed input datatypes (§ 2.2). Finally, to avoid inflating our verification numbers, we excluded three BIND commits that patched 200-300 functions each by changing a pervasive linked-list macro and/or replacing all uses of `memcpy` with `memmove`. Neither of these changes introduced any errors and, given their near-trivial modifications, shed little additional light on our tool's effectiveness. This yielded 487 patches from BIND and 324 patches from OpenSSL, both from 177 distinct commits to BIND and OpenSSL (purely by coincidence).

We compiled *patched* and *unpatched* versions of the codebase for each revision using an LLVM 2.7 toolchain. We then ran UC-KLEE over each patch for one hour. Each run was allocated a single Intel Xeon E5645 2.4GHz core and 4GB of memory on a compute cluster running 64-bit Fedora Linux 14. For these runs, we configured UC-KLEE to target crashes only in patched routines or routines they call. While this approach allows UC-KLEE to focus on the most likely source of errors, it does not detect bugs caused by the outputs of a function, which may trigger crashes elsewhere in the system (e.g., if the function unexpectedly returns null). UC-KLEE can report such differences, but we elide that feature in this paper.

Code modifications. In BIND and OpenSSL, we canonicalized several macros that introduced spurious code differences such as the `__LINE__`, `VERSION`, `SRCID`, `DATE`, and `OPENSSL_VERSION_NUMBER` macros. To support function-call annotations (§ 3.2.5) in BIND, we converted four preprocessor macros to function calls.

For BIND, we disabled expensive assertion-logging code and much of its debug `malloc` functionality, which UC-KLEE already provided. For OpenSSL, we added a new build target that disabled reference counting and address alignment. The reference counting caused many false positives; UC-KLEE reported double free errors due to unknown preconditions on an object's reference count.

3.2.2 Bugs found

From the patches we tested, UC-KLEE uncovered three previously unknown bugs in BIND and eight bugs in OpenSSL, six of which were previously unknown. These bugs are summarized in Figure 4.

```
1 :  LOCK_ZONE(zone);
2 :  if (DNS_ZONE_FLAG(zone, DNS_ZONEFLG_EXITING)
3 :      || !inline_secure(zone)) {
4 :    result = ISC_R_SHUTTINGDOWN;
5 :    goto unlock;
6 :  }
7 :  . . .
8 :  if (result != ISC_R_SUCCESS)
9 :    goto failure; /* ← bypasses UNLOCK_ZONE */
10:  . . .
11: unlock:
12:  UNLOCK_ZONE(zone);
13: failure:
14:  dns_zone_idetach(&zone);
```

Figure 5: BIND locking bug found in `receive_secure_db`.

Figure 5 shows a representative double-lock bug in BIND found by cross-checking. The patch moved the `LOCK_ZONE` earlier in the function (line 1), causing existing error handling code that jumped to `failure` (line 9) to bypass the `UNLOCK_ZONE` (line 12). In this case, the subsequent call to `dns_zone_idetach` (line 14) reacquires the already-held lock, which triggers an assertion failure. This bug was one of several we found that involved infrequently-executed error handling code. Worse, BIND often hides `goto failure` statements inside a `CHECK` macro, which was responsible for a bug we discovered in the `save_nsec3param` function (not shown). We reported the bugs to the BIND developers, who promptly confirmed and fixed them. These examples demonstrate a key benefit of UC-KLEE: it explores non-obvious execution paths that would likely be missed by a human developer, either because the code is obfuscated or an error condition is overlooked.

UC-KLEE is not limited to finding new bugs introduced by the patches; it can also find old bugs in patched code. We added a new mode where UC-KLEE flags errors that occur in both P and P' if the error *must* occur for all input values following that execution path (*must-fail* error described in § 3.2.5). This approach allowed us to find one new bug in BIND and four in OpenSSL. It also re-confirmed a number of bugs found by cross-checking above. This mode could be used to find bugs in functions that have not been patched, but we did not use it for that purpose in this paper.

Figure 6 shows a representative must-fail bug, a previously unknown null pointer dereference (denial-of-service) vulnerability we discovered in OpenSSL's

```
1 : if (wb−>buf == NULL) /* ← null pointer check */
2 :     if (!ssl3_setup_write_buffer(s))
3 :         return −1;
4 : . . .
5 : /* If we have an alert to send, lets send it */
6 : if (s−>s3−>alert_dispatch) {
7 :     /* call sets wb→buf to NULL */
8 :     i=s−>method−>ssl_dispatch_alert(s);
9 :     if (i <= 0)
10:         return(i);
11:     /* if it went, fall through and send more stuff */
12: }
13: . . .
14: unsigned char *p = wb−>buf; /* ← p = NULL */
15: *(p++)=type&0xff; /* ← null pointer dereference */
```

Figure 6: OpenSSL null pointer bug in `do_ssl3_write`.

`do_ssl3_write` function that led to security advisory CVE-2014-0198 [12] being issued. In this case, a developer attempted to prevent this bug by explicitly checking whether `wb->buf` is null (line 1). If the pointer is null, `ssl3_setup_write_buffers` allocates a new buffer (line 2). On line 6, the code then handles any pending alerts [20] by calling `ssl_dispatch_alert` (line 8). This call has the subtle side effect of freeing the write buffer when the common `SSL_MODE_RELEASE_BUFFERS` flag is set. After freeing the buffer, `wb->buf` is set to null (not shown), triggering a null pointer dereference on line 15.

This bug would be hard to find with other approaches. The write buffer is freed by a chain of function calls that includes a recursive call to `do_ssl3_write`, which one maintainer described as “sneaky” [44]. In contrast to static techniques that could not reason precisely about the recursion, UC-KLEE proved that under the circumstances when both an alert is pending and the release flag is set, a null pointer dereference *will* occur. This example also illustrates the weaknesses of regression testing. While a developer may write tests to make sure this function works correctly when an alert is pending *or* when the release flag is set, it is unlikely that a test would exercise these conditions simultaneously. Perhaps as a direct consequence, this vulnerability was nearly six years old.

3.2.3 Patches verified

In addition to finding new bugs, UC-KLEE exhaustively verified all execution paths for 67 (13.8%) of the patches in BIND, and 48 (14.8%) of the patches in OpenSSL. Our system effectively verified that, up to the given input bound and with the usual caveats, these patches did not introduce any new crashes. This strong result is not possible with imprecise static analysis or testing.

The median instruction coverage (§ 3.2.4) for the exhaustively verified patches was 90.6% for BIND and 100% for OpenSSL, suggesting that these patches were thoroughly tested. Only six of the patches in BIND and one in OpenSSL achieved very low (0-2%) coverage. We determined that UC-KLEE achieved low coverage on these patches due to dead code (2 patches); an insufficient symbolic input bound (2 patches); comparisons between input pointers (we assume no aliasing, 1 patch); symbolic `malloc` size (1 patch); and a trivial stub function that was optimized away (1 patch).

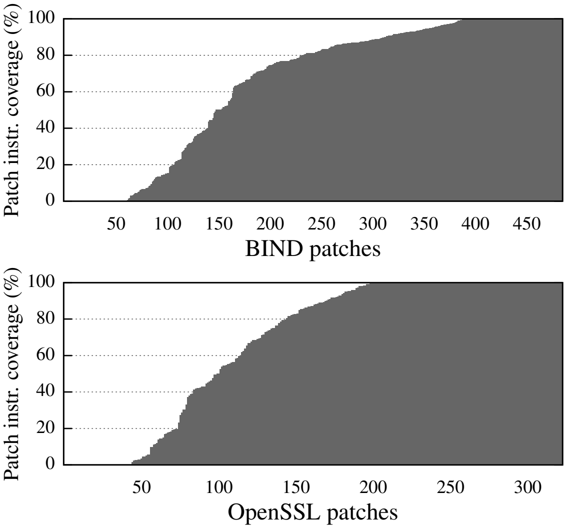


Figure 7: Coverage of patched instructions: 100% coverage for 98 BIND patches (20.1%) and 124 OpenSSL patches (38.3%). Median was 81.1% for BIND, 86.9% for OpenSSL.

3.2.4 Patches partially verified

This section measures how thoroughly we check non-terminating patches using two metrics: (1) instruction coverage, and (2) number of execution paths completed.

We conservatively measure instruction coverage by counting the number of instructions that differ in P' from P and then computing the percentage of these instructions that UC-KLEE executes at least once. Figure 7 plots the instruction coverage. The median coverage was 81.1% for BIND and 86.9% for OpenSSL, suggesting that UC-KLEE thoroughly exercised the patched code, even when it did not exhaust all paths.

Figure 8 plots the number of completed execution paths for each patch we did not exhaustively verify (§ 3.2.3) that hit at least one patched instruction. These graphs exclude 31 patches for BIND and 32 patches for OpenSSL for which our system crashed during the one hour execution window. The crashes were primarily due to bugs in our tool and memory exhaustion/blowup caused by symbolically executing cryptographic ciphers.

For the remaining patches, UC-KLEE completed a median of 5,828 distinct paths per patch for BIND and 1,412 for OpenSSL. At the upper end, 154 patches for BIND (39.6%) and 79 for OpenSSL (32.4%) completed over 10,000 distinct execution paths. At the bottom end, 58 patches for BIND (14.9%) and 46 for OpenSSL (18.9%) completed zero execution paths. In many cases, UC-KLEE achieved high coverage on these patches but neither detected errors nor ran the non-error paths to com-

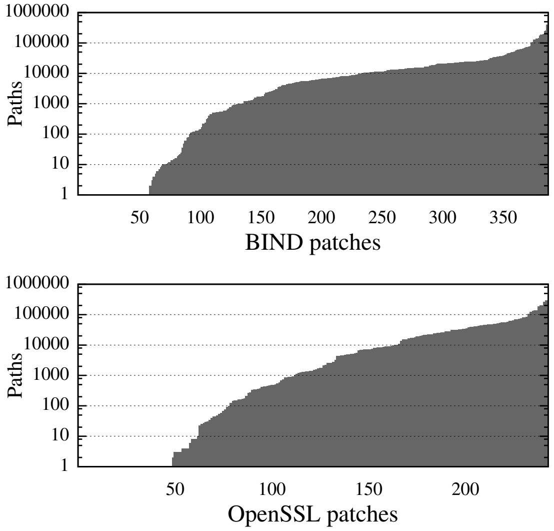


Figure 8: Completed execution paths (log scale). Median was 5,828 paths per patch for BIND and 1,412 for OpenSSL. Top quartile was 17,557 paths for BIND and 21,859 for OpenSSL.

pletion. A few reasons we observed for paths not running to completion included query timeouts, unspecified symbolic function pointers, or ineffective search heuristics.

These numbers should only be viewed as a crude approximation of thoroughness; they do not measure the independence between the paths explored (greater is preferable). On the other hand, they grossly undercount the number of distinct concrete values each symbolic path reasons about simultaneously. One would generally expect that exercising 1,000 or more paths through a patch, where each path simultaneously tests all feasible values, represents a dramatic step beyond the current standard practice of running the patch on a few tests.

3.2.5 False positives

This section describes our experience in separating true bugs from *false positives*, which were due to missing input preconditions. The false positives we encountered were largely due to three types of missing preconditions:

1. Data structure invariants, which apply to all instances of a data structure (e.g., a parent node in a binary search tree has a greater value than its left child).
2. State machine invariants, which determine the sequence of allowed values and the variable assignments that may exist simultaneously (e.g., a counter increases monotonically).
3. API invariants, which determine the legal inputs to API entry points (e.g., a caller must not pass a null pointer as an argument).

Figure 9 illustrates a representative example of a false positive from BIND, which was caused by a missing data structure invariant. The `isc_region_t` type consists of a buffer and a length, but UC-KLEE has no knowledge that the two are related. The code selects a valid buffer length at line 14, the shorter of the two buffers. At line 17, the code calls `memcmp` and supplies this length. Inside `memcmp`, UC-KLEE reported hundreds of false positives involving out-of-bounds memory reads. These errors occurred on false paths where the buffer pointed to by the `base` field was smaller than the associated `length` field.

```
1 : typedef struct isc_region {
2 :     unsigned char * base;
3 :     unsigned int    length;
4 : } isc_region_t;
5 :
6 : int isc_region_compare(isc_region_t *r1, isc_region_t *r2) {
7 :   unsigned int l;
8 :   int result;
9 :
10:   REQUIRE(r1 != NULL);
11:   REQUIRE(r2 != NULL);
12:
13:   /* chooses min. buffer length */
14:   l = (r1−>length < r2−>length) ? r1−>length : r2−>length;
15:
16:   /* memcmp reads out-of-bounds */
17:   if ((result = memcmp(r1−>base, r2−>base, l)) != 0)
18:     return ((result < 0) ? −1 : 1);
19:   else
20:     return ((r1−>length == r2−>length) ? 0 :
21:       (r1−>length < r2−>length) ? −1 : 1);
22: }
```

Figure 9: Example false positive in BIND. UC-KLEE does not associate `length` field with buffer pointed to by `base` field. Consequently, UC-KLEE falsely reports that `memcmp` (line 17) reads out-of-bounds from `base`.

UC-KLEE manages false positives using two approaches: manual annotations and automated heuristics.

Manual annotations. UC-KLEE supports two types of manual annotations: (1) data type annotations, and (2) function call annotations. Both are written in C and compiled with LLVM. UC-KLEE invokes data type annotations at the end of a path, prior to emitting an error. These are associated with named data types and specify invariants on symbolic inputs of that type (inferred from debug information when available). For the example above, we added the following simple annotation for the `isc_region_t` data type:

```
INVARIANT(r−>length <= OBJECT_SIZE(r−>base));
```

The `INVARIANT` macro requires that the condition hold. If it is infeasible (cannot be true) on the current path, UC-KLEE emits an error report with a flag indicating that the annotations have been violated. We use this flag to filter out uninteresting error reports. This one simple annotation allowed us to filter 623 errors, which represented about 7.5% of all the errors UC-KLEE reported for BIND.

Function call annotations are used to run specific code immediately prior to calling a function. For example, we wrote a function call annotation for BIND that runs before each call to `isc_mutex_lock`, with the same arguments:

```
void annot_isc_mutex_lock(isc_mutex_t *mp) {
  EXPECT(*mp == 0);
}
```

	P' **only**			P **and** P'		
	Reports		**Patches**	**Reports**		**Patches**
Heuristic	Tot.	Bugs		Tot.	Bugs	
Total errors	2446	3	141	5829	-	260
Manual annotations	1419	3	125	1378	-	153
must-fail	44	3	8	1378	-	153
concrete-fail	26*	2	6*	878	-	110
belief-fail	35*	3	7*	1053	-	127
excluding inputs	30*	3	7*	852	-	102
True bugs	**3***	**3**	**3***	**1**	**1**	**1**

(a) BIND (487 patches, 4 distinct bugs)

	P' **only**			P **and** P'		
	Reports		**Patches**	**Reports**		**Patches**
Heuristic	Tot.	Bugs		Tot.	Bugs	
Total errors	1423	5	79	579	11	125
Manual annotations	1286	5	79	451	11	124
must-fail	41	5	22	451	11	124
concrete-fail	14*	5	12*	224	11	98
belief-fail	25*	5	18*	316	11	117
excluding inputs	17*	5	11*	90*	11	47*
True bugs	**5***	**5**	**4***	**11***	**11**	**10***

(b) OpenSSL (324 patches, 8 distinct bugs)

Figure 10: Effects of heuristics on false positives. *Tot.* indicates the total number of reports, of which *Bugs* are true errors; *Patches* indicates the number of patches that reported at least one error. P' *only* refers to errors that occurred only in function P'; *P and* P' occurred in both versions. Indent indicates successive heuristics; * indicates that we reviewed all the reports manually.

Macro	**Description**
`INVARIANT(condition)`	Add `condition` as a path constraint; kill path if infeasible.
`EXPECT(condition)`	Add `condition` as a path constraint if feasible; otherwise, ignore.
`IMPLIES(a, b)`	Logical implication: $a \rightarrow b$.
`HOLDS(a)`	Returns true if condition `a` *must* hold.
`MAY_HOLD(a)`	Returns true if condition `a` *may* hold.
`SINK(e)`	Forces `e` to be evaluated; prevents compiler from optimizing it away.
`VALID_POINTER(ptr)`	Returns true if `ptr` is valid; false otherwise.
`OBJECT_SIZE(ptr)`	Returns the size of the object pointed to by `ptr`; kills path if pointer is invalid.

Figure 11: C annotation macros.

The `EXPECT` macro adds the specified path constraint only if the condition is feasible on the current path and elides it otherwise. In this example, we avoid considering cases where the mutex is already locked. However, this annotation has no effect if the condition is *not* feasible (i.e., the lock has definitely been acquired along this path). This annotation allows UC-KLEE to detect errors in lock usage while suppressing false positives under the assumption that if a function attempts to acquire a lock supplied as input, then a likely input precondition is that the lock is not already held. This annotation did not prevent us from finding the BIND locking bug in `receive_secure_db` shown in Figure 5.

Figure 11 summarizes the convenience macros we provided for expressing annotations using C code. While annotations may be written using arbitrary C code, these macros provide a simple interface to functionality not expressible with C itself (e.g., determining the size of a heap object using `OBJECT_SIZE`). The `HOLDS` and `MAY_HOLD` macros allow code to check the feasibility of a Boolean expression without causing UC-KLEE to fork execution and trigger path explosion.

For BIND, we wrote 13 function call annotations and 31 data type annotations (about 400 lines of C). For OpenSSL, we wrote six data type annotations and no function call annotations (60 lines). We applied a single set of annotations for each codebase to all the patches we tested. In our experience, most of these annotations were simple to specify and often suppressed many false positives. We felt the level of effort required was reasonable compared to the sizes of the codebases we checked. We added annotations lazily, in response to false positives.

Figure 10 illustrates the effects of the annotations and heuristics on the error reports for BIND and OpenSSL. The P' *only* column describes errors that only occurred in the patched function, while *P and* P' describes errors that occurred in both versions. In this experiment, we are primarily concerned with bugs introduced by a patch, so our discussion describes P' *only* unless otherwise noted.

The manual annotations suppressed 42% of the reports for BIND but only 9.6% for OpenSSL. We attribute this difference to the greater effort we expended writing manual annotations for BIND, for which the automated heuristics were less effective without the annotations.

Automated heuristics. We tried numerous heuristics to reduce false reports. UC-KLEE augments each error report with a list of the heuristics that apply. The **must-fail** heuristic identifies errors that *must* occur for all input values following that execution path, since these are often true errors [18]. For example, assertion failures are must-fail when the condition must be false.

A variation on the must-fail heuristic is the **belief-fail** heuristic, which uses a form of belief analysis [19]. The intuition behind this heuristic is that if a function *contradicts* itself, it likely has a bug. For example, if the code checks that a pointer is null and then dereferences the pointer, it has a bug, regardless of any input preconditions. On the other hand, a function is generally agnostic to the assumptions made by the functions it calls. For example, if `strcmp` checks whether two strings have the same address, the caller does not acquire this belief, even if the path constraints now indicate that the two addresses match. Following this intuition, the belief-fail heuristic identifies errors that occur for all input values satisfying the *belief set*, which is the set of constraints (i.e., branch conditions) added within the current function or inherited from its caller, but not its callees. We track belief sets for each stack frame.

A second variation on must-fail is **concrete-fail**, which indicates that an assertion failure or memory error was triggered by a concrete (non-symbolic) condition or pointer, respectively. In practice, this heuristic and belief-fail were the most effective.

These heuristics reduced the total number of reports to a small enough number that we were able to inspect them all manually. While only 8.6% of the belief-fail errors for BIND and 20% of those for OpenSSL were true bugs, the total number of these errors (60) was manageable relative to the number of patches we tested (811). In total, the annotations and belief-fail heuristic eliminated 98.6% of false positives for BIND and 98.2% for OpenSSL.

A subset of the belief-fail errors were caused by reading past the end of an input buffer, and none of these were true bugs. Instead, they were due to paths reaching the input bound we specified. In many cases, our system would emit these errors for any input bound because they involved unbounded loops (e.g., `strlen`). The *excluding inputs* row in Figure 10 describes the subset of belief-fail errors not related to input buffers. This additional filter produced a small enough set of *P and P′* errors for OpenSSL that we were able to manually inspect them, discovering a number of additional bugs. We note that the *true errors* listed in Figure 10 constitute 12 distinct bugs; some bugs showed up in multiple error reports.

4 Generalized checking

In addition to checking patches, UC-KLEE provides an interface for rule-based checkers to be invoked during symbolic path exploration. These checkers are similar to tools built using dynamic instrumentation systems such as Valgrind [34] or Pin [30]. Unlike these frameworks, however, UC-KLEE applies its checkers to all possible paths through a function, not to a single execution path through a program. In addition, UC-KLEE considers all possible input values along each path, allowing it to discover bugs that might be missed when checking a single set of concrete inputs.

Conceptually, our framework is similar to WOODPECKER [8], a KLEE-based tool that allows system-specific checkers to run on top of (whole program) symbolic execution. In this paper, however, we focus on generic checkers we implemented for rules that apply to many systems, and we directly invoked these checkers on individual functions deep within each codebase.

UC-KLEE provides a simple interface for implementing checkers by deriving from a provided C++ base class. This interface provides hooks for a checker to intercept memory accesses, arithmetic operations, branches, and several types of errors UC-KLEE detects.

A user invoking UC-KLEE provides a compiled LLVM module and the name of a function to check. We refer to this function as the *top-level* function. Generally, the module has been linked to include all functions that might be called by the top-level function. When UC-KLEE encounters a function call, it executes the called function. When UC-KLEE encounters a call to a function missing from the LLVM module, however, it may optionally skip over the function call rather than terminate the path with an error message. When UC-KLEE skips a function call, it creates a new under-constrained value to represent the function's return value, but it leaves the function's arguments unchanged. This approach *under-approximates* the behaviors that the missing function might perform (e.g., writing to its arguments or globals). Consequently, UC-KLEE may miss bugs and cannot provide verification guarantees when functions are missing.

We briefly experimented with an alternative approach in which we overwrote the skipped function's arguments with new under-constrained values, but this over-approximation caused significant path explosion, mostly involving paths that could not arise in practice.

In addition to missing functions due to scalability limitations, we also encountered inline assembly (Linux kernel only) and unresolved symbolic function pointers. We skipped these two cases in the same manner as missing functions. For all three cases, UC-KLEE provides a hook to allow a checker to detect when a call is being skipped and to take appropriate actions for that checker.

In the remainder of this section, we describe each checker, followed by our experimental results in § 4.4.

4.1 Leak checker

Memory leaks can lead to memory exhaustion and pose a serious problem for long-running servers. Frequently, they are exploitable as denial-of-service vulnerabilities [10, 13, 14]. To detect memory leaks (which may or may not be remotely exploitable, depending on their location within a program), we implemented a leak checker on top of UC-KLEE. The leak checker considers a heap object to be leaked if, after returning from the top-level function, the object is not reachable from a *root set* of pointers. The root set consists of a function's (symbolic) arguments, its return value, and all global variables. This checker is similar to the leak detection in Purify [23] or Valgrind's memcheck [34] tool, but it thoroughly checks all paths through a specific function, rather than a single concrete path through a whole program.

When UC-KLEE encounters a missing function, the leak checker finds the set of heap objects that are reachable from each of the function call's arguments using a precise approach based on pointer referents [42, 43]. It then marks these objects as *possibly escaping*, since the missing function could capture pointers to these objects and prevent them from becoming unreachable. At the end of each execution path, the leak checker removes any possibly escaping objects from the set of leaked objects.

Doing so allows it to report only true memory leaks, at the cost of possibly omitting leaks when functions are missing. However, UC-KLEE may still report false leaks along invalid execution paths due to missing input preconditions. Consider the following code fragment:

```
1 : char* leaker() {
2 :     char *a = (char*) malloc(10); /* not leaked */
3 :     char *b = (char*) malloc(10); /* maybe leaked */
4 :     char *c = (char*) malloc(10); /* leaked! */
5 :
6 :     bar(b); /* skipped call to bar */
7 :     return a;
8 : }
```

When UC-KLEE returns from the function `leaker`, it inspects the heap and finds three allocated objects: `a`, `b`, and `c`. It then examines the root set of objects. In this example, there are no global variables and `leaker` has no arguments, so the root set consists only of `leaker`'s return value. UC-KLEE examines this return value and finds that the pointer `a` is live (and therefore not leaked). However, neither `b` nor `c` is reachable. It then looks at its list of *possibly escaping* pointers due to the skipped call to `bar` on line 6, which includes `b`. UC-KLEE subtracts `b` from the set of leaked objects and reports back to the user that `c` has been leaked. While this example is trivial, UC-KLEE discovered 37 non-trivial memory leak bugs in BIND, OpenSSL, and the Linux kernel (§ 4.4).

4.2 Uninitialized data checker

Functions that access uninitialized data from the stack or heap exhibit undefined or non-deterministic behavior and are particularly difficult to debug. Additionally, the prior contents of the stack or heap may hold sensitive information, so code that operates on these values may be vulnerable to a loss of confidentiality.

UC-KLEE includes a checker that detects accesses to uninitialized data. When a function allocates stack or heap memory, the checker fills it with special *garbage* values. The checker then intercepts all loads, binary operations, branches, and pointer dereferences to check whether any of the operands (or the result of a load) contain garbage values. If so, it reports an error to the user.

In practice, loads of uninitialized data are often intentional; they frequently arise within calls to `memcpy` or when code manipulates bit fields within a C `struct`. Our evaluation in § 4.4 therefore focuses on branches and dereferences of uninitialized pointers.

When a call to a missing function is skipped, the uninitialized data checker *sanitizes* the function's arguments to avoid reporting spurious errors in cases where missing functions write to their arguments.

4.3 User input checker

Code that handles untrusted user input is particularly prone to bugs that lead to security vulnerabilities since an attacker can supply any possible input value to exploit the code. Generally, UC-KLEE treats inputs to a function as *under-constrained* because they may have unknown preconditions. For cases where inputs originate from untrusted sources such as network packets or user-space data passed to the kernel, however, the inputs can be considered *fully-constrained*. This term indicates that the set of legal input values is known to UC-KLEE; in this case, any possible input value may be supplied. If any value triggers an error in the code, then the error is likely to be exploitable by an attacker, assuming that the execution path is feasible (does not violate other preconditions).

UC-KLEE maintains *shadow memory* (metadata) associated with each symbolic input that tracks whether each symbolic byte is under-constrained or fully-constrained. UC-KLEE provides an interface for system-specific C annotations to mark untrusted inputs as fully-constrained by calling the function `ucklee_clear_uc_byte`. This function sets the shadow memory for each byte to the *fully-constrained* state.

UC-KLEE includes a system-configurable user input checker that intercepts all errors and adds an `UNSAFE_INPUT` flag to errors caused by fully-constrained inputs. For memory access errors, the checker examines the pointer to see if it contains fully-constrained symbolic values. For assertion failures, it examines the assertion condition. For division-by-zero errors, it examines the divisor.

In all cases, the checker inspects the fully-constrained inputs responsible for an error and determines whether any path constraints compare the inputs to under-constrained data (originating elsewhere in the program). If so, the checker assumes that the constraints *may* properly sanitize the input, and it suppresses the error. Otherwise, it emits the error. This approach avoids reporting spurious errors to the user, at the cost of missing errors when inputs are partially (but insufficiently) sanitized.

We designed this checker primarily to find security vulnerabilities similar to the OpenSSL "Heartbleed" vulnerability [1, 11] from 2014, which passed an untrusted and unsanitized length argument to `memcpy`, triggering a severe loss of confidentiality. In that case, the code never attempted to sanitize the length argument. To test this checker, we ran UC-KLEE on an old version of OpenSSL without the fix for this bug and confirmed that our checker reports the error.

4.4 Evaluation

We evaluated UC-KLEE's checkers on over 20,000 functions from BIND, OpenSSL, and the Linux kernel. For BIND and OpenSSL, we used UC-KLEE to check all functions except those in the codebases' `test` directories. We used the same minor code modifications described in § 3.2.1, and we again used a maximum input

	Leak Checker				Uninitialized Data Checker					User Input Checker			
	Funcs.	Bugs	Reports	False	Funcs.	Bugs	Pointer Reports	Pointer False	Branch Reports	Funcs.	Bugs	Reports	False
BIND	6239	9	138	2.2%	6239	3	0	-	244*	6239	0	67	100%
OpenSSL	6579	5	$272^{\dagger}$	90.1%	6579	6	197	92.90%	564*	6579	0	5	100%
Linux kernel	5812	23	127	76.4%	7185	10	72	83.30%	494*	1857	11	145	80.0%

Figure 12: Summary of results from running UC-KLEE checkers on *Funcs* functions from each codebase. *Bugs* shows the number of distinct true bugs found (67 total). *Reports* shows the total number of errors reported by UC-KLEE in each category (multiple errors may point to a single bug). *False* reports the percentage of errors reported that did not appear to be true bugs (i.e., false positives). †excludes reports for obfuscated ASN.1 code. *denotes that we inspected only a handful of errors for that category.

```
1 : int gssp_accept_sec_context_upcall(struct net *net,
2 :                                     struct gssp_upcall_data *data) {
3 :   ...
4 :   ret = gssp_alloc_receive_pages(&arg);
5 :   ...
6 :   gssp_free_receive_pages(&arg);
7 :   ...
8 : }
9 : int gssp_alloc_receive_pages(struct gssx_arg_accept_sec_context *arg) {
10:   arg−>pages = kzalloc(...);
11:   ...
12:   return 0;
13: }
14: void gssp_free_receive_pages(struct gssx_arg_accept_sec_context *arg) {
15:   for (i = 0; i < arg−>npages && arg−>pages[i]; i++)
16:     __free_page(arg−>pages[i]);
17:   /* missing: kfree(arg−>pages); */
18: }
```

Figure 13: Linux kernel memory leak in RPCSEC_GSS protocol implementation used by NFS server-side AUTH_GSS.

size of 25,000 bytes and a depth bound of 9 objects.

For the Linux kernel, we included functions relevant to each checker, as described below. Unlike our evaluation in § 3.2, we did not use any manual annotations to suppress false positives. We ran UC-KLEE for up to five minutes on each function from BIND and the Linux kernel, and up to ten minutes on each OpenSSL function. We used the same machines as in § 3.2.

For BIND, we checked version 9.10.1-P1 (12/2014). For OpenSSL, we checked version 1.0.2 (1/2015). For the Linux kernel, we checked version 3.16.3 (9/2014).

Figure 12 summarizes the results. UC-KLEE discovered a total of 67 previously-unknown bugs[1]: 12 in BIND, 11 in OpenSSL, and 44 in the Linux kernel. Figure 14 lists the number of functions that UC-KLEE exhaustively verified (up to the given input bound and with caveats) as having each property. We omit verification results from the Linux kernel because UC-KLEE skipped many function calls and inline assembly, causing it to under-approximate the set of possible execution paths and preventing it from making any verification guarantees. We did link each Linux kernel function with other modules from the same directory, however, as well as the `mm/vmalloc.c` module.

	No leaks	No `malloc`	No uninitialized data
BIND	388	1776	2045
OpenSSL	383	1648	2043

Figure 14: Functions verified (with caveats) by UC-KLEE.

4.4.1 Leak checker

The leak checker was the most effective. It reported the greatest number of bugs (37 total) and the lowest false positive rate. Interestingly, only three of the 138 leak reports for BIND were spurious errors, a false positive rate of only 2.2%. For OpenSSL, we excluded 269 additional reports involving the library's obfuscated ASN.1 [25] parsing code, which we could not understand. Of the remaining 272 reports, the checker found five bugs but had a high false positive rate of 90.1%.

For the Linux kernel, we wrote simple C annotations (about 60 lines) to intercept calls to `kmalloc`, `vmalloc`, `kfree`, `vfree`, and several similar functions, and to forward these to UC-KLEE's built-in `malloc` and `free` functions. Doing so allowed us to track memory management without the overhead of symbolically executing the kernel's internal allocators. We then ran UC-KLEE on all functions that directly call these allocation functions.

Our system discovered 23 memory leaks in the Linux kernel. One particularly interesting example (Figure 13) involved the SunRPC layer's server-side implementation of `AUTH_GSS` authentication for NFS. Each connection triggering an upcall causes 512 bytes allocated at line 10 to be leaked due to a missing `kfree` that should be present around line 17. Since this leak may be triggered by remote connections, it poses a potential denial-of-service (memory exhaustion) vulnerability. The NFS maintainers accepted our patch to fix the bug.

UC-KLEE found that at least 2909 functions in BIND and at least 3700 functions in OpenSSL (or functions they call) allocate heap memory. As shown in Figure 14, UC-KLEE verified (with caveats) that 388 functions in BIND and 383 in OpenSSL allocate heap memory but do not leak it. Our system also verified that 1776 functions in BIND and 1648 functions in OpenSSL do not allocate heap memory, making them trivially leak-free.

4.4.2 Uninitialized data checker

The uninitialized data checker reported a total of 19 new bugs. One illustrative example, shown in Figure 15, in-

[1] A complete list of the bugs we discovered is available at: `http://cs.stanford.edu/~daramos/usenix-sec-2015`

```
1 :   points = OPENSSL_malloc(sizeof (EC_POINT*)*(num + 1));
2 :   ...
3 :   for (i = 0; i < num; i++) {
4 :     if ((points[i] = EC_POINT_new(group)) == NULL)
5 :       goto err; /* leaves 'points' only partially initialized */
6 :   }
7 :   ...
8 : err:
9 :   ...
10:   if (points) {
11:     EC_POINT **p;
12:     for (p = points; *p != NULL; p++)
13:       EC_POINT_free(*p); /* dereference/free of uninitialized pointer */
14:     OPENSSL_free(points);
15:   }
```

Figure 15: OpenSSL dereference/`free` of uninitialized pointer in `ec_wNAF_precompute_mult` function.

volves OpenSSL's elliptic curve cryptography. If the call to `EC_POINT_new` on line 4 fails, the code jumps to line 8, leaving the `points` array partially uninitialized. Line 13 then passes uninitialized pointers from the array to `EC_POINT_free`, which dereferences the pointers and passes them to `free`, potentially corrupting the heap. This is one of many bugs that we found involving infrequently executed error-handling code, a common source of security bugs.

UC-KLEE discovered an interesting bug (Figure 16) in BIND's UDP port randomization fix for Kaminsky's cache poisoning attack [9]. To prevent spoofed DNS replies, BIND must use unpredictable source port numbers. The `dispatch_createudp` function calls the `get_udpsocket` function at line 9, which selects a pseudorandom number generator (PRNG) at line 18 based on whether we are using a UDP or TCP connection. However, the `socktype` field isn't initialized in `dispatch_createudp` until line 12, meaning that the PRNG selection is based on uninitialized data. While it appears that the resulting port numbers are sufficiently unpredictable despite this bug, this example illustrates UC-KLEE's ability to find errors with potentially serious security implications.

For the Linux kernel, we checked the union of the functions we used for the leak checker and the user input checker (discussed below) and found 10 bugs.

Due to time limitations, we exhaustively inspected only the most serious category of errors: uninitialized pointers. The checker reported too many uninitialized branches for us to examine completely, but we did inspect a few dozen of these errors in an ad-hoc manner. All three of the bugs from BIND and one bug from the Linux kernel fell into this category. The remaining bugs were uninitialized pointer errors. We did not inspect the error reports for binary operations or load values.

Finally, our system verified (with caveats) that about a third of the functions from BIND (2045) and OpenSSL (2043) do not access uninitialized data. We believe that providing this level of guarantee on such a high percentage of functions with almost no manual effort is a strong result not possible with existing tools.

```
1 : #define DISP_ARC4CTX(disp) \
2 :   ((disp)->socktype == isc_sockettype_udp) \
3 :   ? (&(disp)->arc4ctx) : (&(disp)->mgr->arc4ctx)
4 : static isc_result_t dispatch_createudp(..., unsigned int attributes, ...) {
5 :   ...
6 :   result = dispatch_allocate(mgr, maxrequests, &disp);
7 :   ...
8 :   if ((attributes & DNS_DISPATCHATTR_EXCLUSIVE) == 0) {
9 :     result = get_udpsocket(mgr, disp, ...);
10:     ...
11:   }
12:   disp->socktype = isc_sockettype_udp; /* late initialization */
13:   ...
14: }
15: static isc_result_t get_udpsocket(..., dns_dispatch_t *disp, ...) {
16:   ...
17:   /* PRNG selected based on uninitialized 'socktype' field */
18:   prt = ports[dispatch_uniformrandom(DISP_ARC4CTX(disp), nports)];
19:   ...
20: }
```

Figure 16: BIND non-deterministic PRNG selection bug.

4.4.3 User input checker

The user input checker required us to identify data originating from untrusted sources. Chou [6] observed that data swapped from network byte order to host byte order is generally untrusted. We applied this observation to OpenSSL and used simple annotations (about 40 lines of C) to intercept calls to `n2s`, `n2l`, `n2l3`, `n2l6`, `c2l`, and `c2ln`, and mark the results fully-symbolic. We also applied a simple patch to OpenSSL to replace byte-swapping macros with function calls so that UC-KLEE could use our annotations. We hope to explore automated ways of identifying untrusted data in future work.

For BIND, we annotated (about 50 lines) the byte-swapping functions `ntohs` and `ntohl`, along with `isc_buffer_getuint8` and three other functions that generally read from untrusted buffers.

For the Linux kernel, we found that many network protocols store internal state in network byte order, leading to spurious errors if we consider these to be untrusted. Instead, we annotated (about 40 lines) the `copy_from_user` function and `get_user` macro (which we converted to a function call). In addition, we used an option in UC-KLEE to mark all arguments to the system call handlers `sys_*` as untrusted. Finally, we used UC-KLEE to check the 1502 functions that directly invoke `copy_from_user` and `get_user`, along with the 355 system call handlers in our build.

Reassuringly, this checker did not discover any bugs in the latest versions of BIND or OpenSSL. We attribute this both to the limited amount of data we marked as untrusted and to our policy of suppressing errors involving *possibly* sanitized data (see § 4.3). However, we were able to detect the 2014 "Heartbleed" vulnerability [1, 11] when we ran our system on an old version of OpenSSL.

Interestingly, we did discover 11 new bugs in the Linux kernel. Seven of these bugs were division- or

```
1 : static int dg_dispatch_as_host(..., struct vmci_datagram *dg) {
2 :   /* read length field from userspace datagram */
3 :   dg_size = VMCI_DG_SIZE(dg);
4 :   ...
5 :   dg_info = kmalloc(sizeof(*dg_info) +
6 :           (size_t) dg−>payload_size, GFP_ATOMIC);
7 :   ...
8 :   /* unchecked memcpy length; read overrun */
9 :   memcpy(&dg_info−>msg, dg, dg_size);
10:   ...
11: }
```

Figure 17: Linux kernel VMware Communication Interface driver unchecked `memcpy` length (buffer overread) bug.

remainder-by-zero operations that would trigger floating-point exceptions and crash the kernel. The remaining four bugs are out-of-bounds dereferences.

Figure 17 shows a buffer overread bug we discovered in the kernel driver for the VMware Communication Interface (VMCI) that follows a pattern nearly identical to "Heartbleed." The userspace datagram `dg` is read using `copy_from_user`. The code then allocates a destination buffer on line 5 and invokes `memcpy` on line 9 without sanitizing the `dg_size` field read from the datagram. An attacker could potentially use this bug to copy up to 69,632 bytes of private kernel heap memory and send it from the host OS to the guest OS. Fortunately, this vulnerability is only exploitable by code running locally on the host OS. The maintainers quickly patched this bug.

Figure 18 shows an unsanitized remainder-by-zero bug we found in the kernel driver for the CEPH distributed filesystem. The check at line 6 aims to prevent this bug with a 64-bit comparison, but the divisor at line 8 uses only the low 32 bits of the untrusted `stripe_unit` field (read from userspace using `copy_from_user`). A value such as `0xffffffff00000000` would pass the check but result in a remainder-by-zero error. An unprivileged local attacker could potentially issue an `ioctl` system call to crash the machine. We notified the developers, who promptly fixed the bug.

Because of the ad-hoc nature of this checker, we did not use it to exhaustively verify any properties about the functions we checked.

5 Implementation

This section details optimizations and techniques we implemented to scale our framework and address problems we encountered while applying it to large systems.

5.1 Object sizing

Recall that when an unbound symbolic pointer is dereferenced, UC-KLEE must allocate memory and bind the pointer to it. One challenge in implementing this functionality is picking a useful object size to allocate. If the size is too small, later accesses to this object may trigger out-of-bounds memory errors. On the other hand, a size that is too large can hide legitimate errors. We handled this tradeoff using two approaches.

```
1 : static long _validate_layout(..., struct ceph_ioctl_layout *l) {
2 :   ...
3 :   /* validate striping parameters */
4 :   if ((l−>object_size & ˜PAGE_MASK) ||
5 :        (l−>stripe_unit & ˜PAGE_MASK) ||
6 :        (l−>stripe_unit != 0 && /* ← 64-bit check */
7 :                                   /* 32-bit divisor: */
8 :         ((unsigned)l−>object_size % (unsigned)l−>stripe_unit)))
9 :           return −EINVAL;
10:   ...
11: }
```

Figure 18: Linux kernel CEPH distributed filesystem driver remainder-by-zero bug in `ioctl` handler.

The first approach, which we used for our experiment in § 3.2, implemented a form of *backtracking*. At each unbound pointer dereference, UC-KLEE *checkpoints* the execution state and chooses an initial allocation size using a heuristic that examines any available type information [42]. If the path later reads out-of-bounds from this object, UC-KLEE (1) emits the error to the user, and (2) restores the checkpoint and uses an allocation size large enough to satisfy the most recent memory access. UC-KLEE records the sequence of branches taken after each checkpoint, and it forces the path to *replay* the sequence of branches after increasing the allocation size. In practice, replaying branches exposed many sources of non-determinism in the baseline KLEE tool and its system modeling code, which we were able to eliminate through significant development effort.

An alternative approach that we recently incorporated into UC-KLEE is to use symbolically-sized objects, rather than selecting a single concrete size. Doing so avoids the need for backtracking in most cases by simultaneously considering many possible object sizes. At each memory access, UC-KLEE determines whether the offset could exceed the object's symbolic size. If so, it emits an error to the user. It also considers a path on which the offset does not exceed this bound and adds a path constraint that sets a lower bound on the object's size. We used this approach for our evaluation in § 4.4.

5.2 Error reporting

With whole program symbolic execution, symbolic inputs typically represent unstructured strings or byte arrays from command line arguments or file contents. In this case, an error report typically contains a single set of concrete inputs that trigger the error, along with a backtrace. With under-constrained symbolic execution, however, the inputs are often complex, pointer-rich data structures since UC-KLEE directly executes individual functions within a program. In this case, a single set of concrete values is not easily understood by a user, nor can it be used to trivially reproduce the error outside of UC-KLEE because pointer inputs expect memory objects (i.e., stack, heap, and globals) to be located at specific addresses.

To provide more comprehensible error reports, UC-

KLEE emits a *path summary* for each error. The path summary provides a complete listing of the source code executed along the path, along with the path constraints added by each line of source. The path constraints are expressed in a C-like notation and use the available LLVM debug information to determine the types and names of each field. Below we list example constraints that UC-KLEE included with error reports for BIND (§ 3.2):

```
Code:        REQUIRE(VALID_RBTDB(rbtdb));
Constraint:  uc_dns_rbtdb1.common.impmagic == 1380074548

Code:        if (source->is_file)
Constraint:  uc_inputsource1.is_file == 0

Code:        if (c == EOF)
Constraint:  uc_var2[uc_var1.current + 1] == 255
```

5.3 General KLEE optimizations

We added several scalability improvements to UC-KLEE that apply more broadly to symbolic execution tools. To reduce path explosion in library functions such as `strlen`, we implemented special versions that avoid forking paths by using symbolic if-then-else constructs. We also introduced scores of rules to simplify symbolic expressions [42]. We elide further details due to space.

5.3.1 Lazy constraints

During our experiments, we faced query timeouts and low coverage for several benchmarks that we traced to symbolic division and remainder operations. The worst cases occurred when an unsigned remainder operation had a symbolic value in the denominator. To address this challenge, we implemented a solution we refer to as *lazy constraints*. Here, we defer evaluation of expensive queries until we find an error. In the common case where an error does not occur or two functions exhibit crash equivalence along a path, our tool avoids ever issuing potentially expensive queries. When an error is detected, the tool re-checks that the error path is feasible (otherwise the error is invalid).

Figure 19(a) shows a simple example. With *eager constraints* (the standard approach), the if-statement at line 2 triggers an SMT query involving the symbolic integer division operation `y / z`. This query may be expensive, depending on the other path constraints imposed on `y` and `z`. To avoid a potential query timeout, UC-KLEE introduces a lazy constraint (Figure 19(b)). On line 1, it replaces the result of the integer division operation with a new, unconstrained symbolic value `lazy_x` and adds the lazy constraint `lazy_x = y / z` to the current path. At line 2, the resulting SMT query is the trivial expression `lazy_x > 10`. Because `lazy_x` is unconstrained, UC-KLEE will take both the *true* and *false* branches following the if-statement. One of these branches may violate the constraints imposed on `y` and `z`, so UC-KLEE must check that the lazy constraints are consistent with the full set of path constraints prior to emitting any errors to the user (i.e., if the path later crashes).

```
1 : int x = y / z;
2 : if (x > 10) /* query: y / z > 10 */
3 :    ...
```

(a) Eager constraints (standard)

```
1 : int x = lazy_x; /* adds lazy constraint: lazy_x = y / z */
2 : if (x > 10) /* query: lazy_x > 10 */
3 :    ...
```

(b) Lazy constraints

Figure 19: Lazy constraint used for integer division operation.

In many cases, the delayed queries are more efficient than their eager counterparts because additional path constraints added after the division operation have narrowed the solution space considered by the SMT solver. If our tool determines that the path is infeasible, it silently terminates the path. Otherwise, it reports the error to the user.

5.4 Function pointers

Systems such as the Linux kernel, BIND, and OpenSSL frequently use function pointers within `struct` types to emulate object-oriented methods. For example, different function addresses may be assigned depending on the version negotiated for an SSL/TLS connection [20]. This design poses a challenge for our technique because symbolic inputs contain symbolic function pointers. When our tool encounters an indirect call through one of these pointers, it is unclear how to proceed.

We currently require that users specify concrete function pointers to associate with each type of object (as the need arises). When our tool encounters an indirect call through a symbolic pointer, it looks at the object's debug type information. If the user has defined function pointers for that type of object, our tool executes the specified function. Otherwise, it reports an error to the user and terminates the path. The user can leverage these errors to specify function pointers only when necessary. For BIND, we found that most of these errors could be eliminated by specifying function pointers for only six types: three for memory allocation, and three for internal databases. For OpenSSL, we specified function pointers for only three objects: two related to support for multiple SSL/TLS versions, and one related to I/O.

When running UC-KLEE's checkers, we optionally allow the tool to skip unresolved function pointers, which allows it to check more code but prevents verification guarantees for the affected functions (see § 4).

6 Related work

This paper builds on prior work in symbolic execution [4], particularly KLEE [5] and our early work on UC-KLEE [43]. Unlike our previous work, which targeted small library routines, this paper targets large systems and supports generalized checking.

Other recent work has used symbolic execution to check patches. DiSE [39] performs whole program symbolic execution but prunes paths unaffected by a patch. Differential Symbolic Execution (DSE) [38] and regression verification [21] use abstraction to achieve scalability but may report false differences. By contrast, our approach soundly executes complete paths through each patched function, eliminating this source of false positives. Impact Summaries [2] complement our approach by soundly pruning paths and ignoring constraints unaffected by a patch.

SymDiff [27] provides a scalable solution to check the equivalence of two programs with fixed loop unrolling but relies on imprecise, uninterpreted functions. Differential assertion checking (DAC) [28] is the closest to our work and applies SymDiff to the problem of detecting whether properties that hold in P also hold in P', a generalization of crash equivalence. However, DAC suffers from the imprecisions of SymDiff and reports false differences when function calls are reordered by a patch. Abstract semantic differencing [37] achieves scalability through clever abstraction but, as with SymDiff, suffers additional false positives due to over-approximation.

Recent work has used symbolic execution to generate regression tests exercising the code changed by a patch [41, 31, 32]. While they can achieve high coverage, these approaches use existing regression tests as a starting point and greedily redirect symbolic branch decisions toward a patch, exploring only a small set of execution paths. By contrast, our technique considers all possible intermediate program values as input (with caveats).

Dynamic instrumentation frameworks such as Valgrind [34] and PIN [30] provide a flexible interface for checkers to examine a program's execution at runtime and flag errors. However, these tools instrument a single execution path running with concrete inputs, making them only as effective as the test that supplies the inputs.

Similar to our use of generalized checking in UC-KLEE is WOODPECKER [8], which uses symbolic execution to check system-specific rules. Unlike UC-KLEE, WOODPECKER applies to whole programs, so we expect it would not scale well to large systems. However, WOODPECKER aggressively prunes execution paths that are redundant with respect to individual checkers, a technique that would be useful in UC-KLEE.

Prior work in memory leak detection has used static analysis [45], dynamic profiling [24], and binary rewriting [23]. Dynamic tools such as Purify [23] and Valgrind [34] detect a variety of memory errors at runtime, including uses of uninitialized data. CCured [33] uses a combination of static analysis and runtime checks to detect pointer errors. Our user input checker relates to prior work in dynamic taint analysis, including TaintCheck [35] and Dytan [7].

7 Conclusions and future work

We have presented UC-KLEE, a novel framework for validating patches and applying checkers to individual C/C++ functions using under-constrained symbolic execution. We evaluated our tool on large-scale systems code from BIND, OpenSSL, and the Linux kernel, and we found a total of 79 bugs, including two OpenSSL denial-of-service vulnerabilities.

One avenue for future work is to employ UC-KLEE as a tool for finding general bugs (e.g., out-of-bounds memory accesses) in a single version of a function, rather than cross-checking two functions or using specialized checkers. Our preliminary experiments have shown that this use case results in a much higher rate of false positives, but we did find a number of interesting bugs, including the OpenSSL denial-of-service attack for which advisory CVE-2015-0291 [15, 22, 42] was issued.

In addition, we hope to further mitigate false positives by using ranking schemes to prioritize error reports, and by inferring invariants to reduce the need for manual annotations. In fact, many of the missing input preconditions can be thought of as consequences of a weak type system in C. We may target higher-level languages in the future, allowing our framework to assume many built-in invariants (e.g., that a length field corresponds to the size of an associated buffer).

Acknowledgements

The authors would like to thank Joseph Greathouse and the anonymous reviewers for their valuable feedback. In addition, the authors thank the LibreSSL developers for their quick responses to our bug reports, along with Evan Hunt and Sue Graves of ISC for granting us access to the BIND `git` repository before it became public. This work was supported by DARPA under agreements 1190029-276707 and N660011024088, by the United States Air Force Research Laboratory (AFRL) through contract FA8650-10-C-7024, and by a National Science Foundation Graduate Research Fellowship under grant number DGE-0645962. The views expressed in this paper are the authors' own.

References

[1] Alert (TA14-098A): OpenSSL 'Heartbleed' vulnerability (CVE-2014-0160). `https://www.us-cert.gov/ncas/alerts/TA14-098A`, April 2014.

[2] BACKES, J., PERSON, S., RUNGTA, N., AND TKACHUK, O. Regression verification using impact summaries. In *Proc. of SPIN Symposium on Model Checking of Software (SPIN)* (2013).

[3] BIND. `https://www.isc.org/downloads/bind/`.

[4] BOYER, R. S., ELSPAS, B., AND LEVITT, K. N. Select – a formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices 10*, 6 (June 1975), 234–45.

[5] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex sys-

tems programs. In *Proc. of Symp. on Operating Systems Design and Impl (OSDI)* (2008).

[6] Chou, A. On detecting heartbleed with static analysis. `http://security.coverity.com/blog/2014/Apr/on-detecting-heartbleed-with-static-analysis.html`, 2014.

[7] Clause, J., Li, W., and Orso, A. Dytan: a generic dynamic taint analysis framework. In *Proc. of Intl. Symp. on Software Testing and Analysis (ISSTA)* (2007).

[8] Cui, H., Hu, G., Wu, J., and Yang, J. Verifying systems rules using rule-directed symbolic execution. In *Proc. of Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2013).

[9] CVE-2008-1447: DNS Cache Poisoning Issue ("Kaminsky bug"). `https://kb.isc.org/article/AA-00924`.

[10] CVE-2012-3868. `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-3868`, Jul 2012.

[11] CVE-2014-0160. `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0160`, April 2014.

[12] CVE-2014-0198. `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0198`, May 2014.

[13] CVE-2014-3513. `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-3513`, Oct 2014.

[14] CVE-2015-0206. `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-0206`, Jan 2015.

[15] CVE-2015-0291. `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-0291`, Mar 2015.

[16] CVE-2015-0292. `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-0292`, Mar 2015.

[17] Deng, X., Lee, J., and Robby. Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *Proc. of the 21st IEEE International Conference on Automated Software Engineering* (2006), pp. 157–166.

[18] Engler, D., and Dunbar, D. Under-constrained execution: making automatic code destruction easy and scalable. In *Proc. of the Intl. Symposium on Software Testing and Analysis (ISSTA)* (2007).

[19] Engler, D., Yu Chen, D., Hallem, S., Chou, A., and Chelf, B. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proc. of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)* (2001).

[20] Freier, A. *RFC 6101: The Secure Sockets Layer (SSL) Protocol Version 3.0.* Internet Engineering Task Force (IETF), Aug 2011.

[21] Godlin, B., and Strichman, O. Regression verification: proving the equivalence of similar programs. *Software Testing, Verification and Reliability 23*, 3 (2013), 241–258.

[22] Goodin, D. OpenSSL warns of two high-severity bugs, but no Heartbleed. *Ars Technica* (March 2015).

[23] Hastings, R., and Joyce, B. Purify: Fast detection of memory leaks and access errors. In *Proc. of the USENIX Winter Technical Conference (USENIX Winter '92)* (Dec. 1992), pp. 125–138.

[24] Hauswirth, M., and Chilimbi, T. M. Low-overhead memory leak detection using adaptive statistical profiling. In *Proc. of the Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2004).

[25] International Telecommunication Union. *ITU-T Recommendation X.680: Abstract Syntax Notation One (ASN.1): Specification of basic notation*, Nov 2008.

[26] Khurshid, S., Pasareanu, C. S., and Visser, W. Generalized symbolic execution for model checking and testing. In *Proc. of Intl. Conf. on Tools and Algos. for the Construction and Analysis of Sys.* (2003).

[27] Lahiri, S., Hawblitzel, C., Kawaguchi, M., and Rebelo, H. SymDiff: A language-agnostic semantic diff tool for imperative programs. In *Proc. of Intl. Conf. on Computer Aided Verification (CAV)* (2012).

[28] Lahiri, S. K., McMillan, K. L., Sharma, R., and Hawblitzel, C. Differential assertion checking. In *Proc. of Joint Meeting on Foundations of Software Engineering (FSE)* (2013).

[29] Lattner, C., and Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of the Intl. Symp. on Code Generation and Optimization (CGO)* (2004).

[30] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)* (2005).

[31] Marinescu, P. D., and Cadar, C. High-coverage symbolic patch testing. In *Proc. of Intl. SPIN Symp. on Model Checking Software* (2012).

[32] Marinescu, P. D., and Cadar, C. KATCH: High-coverage testing of software patches. In *Proc. of 9th Joint Mtg. on Foundations of Software Engineering (FSE)* (2013).

[33] Necula, G. C., McPeak, S., and Weimer, W. Ccured: type-safe retrofitting of legacy code. In *Proc. of Symp. on Principles of Programming Languages (POPL)* (2002).

[34] Nethercote, N., and Seward, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proc. of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI '07)* (June 2007), pp. 89–100.

[35] Newsome, J., and Song, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of Network and Distributed Systems Security Symp. (NDSS)* (2005).

[36] OpenSSL. `https://www.openssl.org/source`.

[37] Partush, N., and Yahav, E. Abstract semantic differencing for numerical programs. In *Proc. of Intl. Static Analysis Symposium (SAS)* (2013).

[38] Person, S., Dwyer, M. B., Elbaum, S., and Păsăreanu, C. S. Differential symbolic execution. In *Proc. of ACM SIGSOFT Intl. Symposium on Foundations of Software Engineering (FSE)* (2008), pp. 226–237.

[39] Person, S., Yang, G., Rungta, N., and Khurshid, S. Directed incremental symbolic execution. In *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)* (2011).

[40] Păsăreanu, C. S., and Rungta, N. Symbolic PathFinder: Symbolic execution of java bytecode. In *Proc. of the IEEE/ACM International Conf. on Automated Software Engineering (ASE)* (2010).

[41] Qi, D., Roychoudhury, A., and Liang, Z. Test generation to expose changes in evolving programs. In *Proc. of IEEE/ACM Intl. Conf. on Automated Software Engineering (ASE)* (2010).

[42] Ramos, D. A. *Under-constrained symbolic execution: correctness checking for real code*. PhD thesis, Stanford University, 2015.

[43] Ramos, D. A., and Engler, D. R. Practical, low-effort equivalence verification of real code. In *Proc. of Intl. Conf. on Computer Aided Verification (CAV)* (2011).

[44] Unangst, T. Commit e76e308f (tedu): on today's episode of things you didn't want to learn. `http://anoncvs.estpak.ee/cgi-bin/cgit/openbsd-src/commit/lib/libssl?id=e76e308f`, Apr 2014.

[45] Xie, Y., and Aiken, A. Context- and path-sensitive memory leak detection. In *Proc. of the Intl. Symp. on Foundations of Software Engineering (FSE)* (2005).

[46] Xie, Y., and Aiken, A. Scalable error detection using boolean satisfiability. In *Proc. of the 32nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)* (2005), pp. 351–363.

TaintPipe: Pipelined Symbolic Taint Analysis

Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu
College of Information Sciences and Technology
The Pennsylvania State University
{jum310, dwu, gzx102, jow5222, pliu}@ist.psu.edu

Abstract

Taint analysis has a wide variety of compelling applications in security tasks, from software attack detection to data lifetime analysis. Static taint analysis propagates taint values following all possible paths with no need for concrete execution, but is generally less accurate than dynamic analysis. Unfortunately, the high performance penalty incurred by dynamic taint analyses makes its deployment impractical in production systems. To ameliorate this performance bottleneck, recent research efforts aim to decouple data flow tracking logic from program execution. We continue this line of research in this paper and propose *pipelined symbolic taint analysis*, a novel technique for parallelizing and pipelining taint analysis to take advantage of ubiquitous multi-core platforms. We have developed a prototype system called TaintPipe. TaintPipe performs very lightweight runtime logging to produce compact control flow profiles, and spawns multiple threads as different stages of a pipeline to carry out symbolic taint analysis in parallel. Our experiments show that TaintPipe imposes low overhead on application runtime performance and accelerates taint analysis significantly. Compared to a state-of-the-art inlined dynamic data flow tracking tool, TaintPipe achieves 2.38 times speedup for taint analysis on SPEC 2006 and 2.43 times for a set of common utilities, respectively. In addition, we demonstrate the strength of TaintPipe such as natural support of multi-tag taint analysis with several security applications.

1 Introduction

Taint analysis is a kind of program analysis that tracks some selected data of interest (taint seeds), e.g., data originated from untrusted sources, propagates them along program execution paths according to a customized policy (taint propagation policy), and then checks the taint status at certain critical location (taint sinks). It has been shown to be effective in dealing with a wide range of security problems, including software attack prevention [25, 40], information flow control [45, 34], data leak detection [49], and malware analysis [43], to name a few.

Static taint analysis [1, 36, 28] (STA) is performed prior to execution and therefore it has no impact on runtime performance. STA has the advantage of considering multiple execution paths, but at the cost of potential imprecision. For example, STA may result in either under-tainting or over-tainting [32] when merging results at control flow confluence points. Dynamic taint analysis (DTA) [25, 13, 27], in contrast, propagates taint as a program executes, which is more accurate than static taint analysis since it only considers the actual path taken at run time. However, the high runtime overhead imposed by dynamic taint propagation has severely limited its adoption in production systems. The slowdown incurred by conventional dynamic taint analysis tools [25, 13] can easily go beyond 30X times. Even with the state-of-the-art DTA tool based on Pin [20], typically it still introduces more than 6X slowdown.

The crux of the performance penalty comes from the strict coupling of program execution and data flow tracking logic. The original program instructions mingle with the taint tracking instructions, and usually it takes 6–8 extra instructions to propagate a taint tag in shadow memory [11]. In addition, the frequent "context switches" between the original program execution and its corresponding taint propagation lead to register spilling and data cache pollution, which add further pressure to runtime performance. The proliferation of multi-core systems has inspired researchers to decouple taint tracking logic onto spare cores in order to improve performance [24, 31, 26, 15, 17, 9]. Previous work can be classified into two categories. The first category is hardware-assisted approaches. For example, Speck [26] needs OS level support for speculative execution and rollback. Ruwase et al. [31] employ a customized hardware for logging a program trace and delivering it to other idle cores for inspection. Nagarajan et al. [24] utilize a hardware first-in first-out buffer to speed up communication between cores. Although they can achieve an appealing performance, the requirement of special hardware prevents them from being adopted using commodity hardware.

The second category is software-only methods that work with binary executables on commodity multi-core hardware [15, 17, 9]. These software-only solutions rely on dynamic binary instrumentation (DBI) to decouple dynamic taint analysis from program execution. The program execution and parallelized taint analysis have to be properly synchronized to transfer the runtime values that are necessary for taint analysis. Although these approaches look promising, they fail to achieve expected performance gains due to the large amounts of communication data and frequent synchronizations between the original program execution thread (or process) and its corresponding taint analysis thread (or process). Recent work ShadowReplica [17] creates a secondary shadow thread from primary application thread to run DTA in parallel. ShadowReplica conducts an offline optimization to generate optimized DTA logic code, which reduces the amount of information that needs to be communicated, and thus dramatically improves the performance. However, as we will show later, the performance improvement achieved by this “primary & secondary” thread model is fixed and cannot be improved further when more cores are available. Furthermore, in many security related tasks (e.g., binary de-obfuscation and malware analysis), precise static analysis for the offline optimization needed by ShadowReplica may not be feasible.

In this paper, we exploit another style of parallelism, namely pipelining. We propose a novel technique, called TaintPipe, for parallel data flow tracking using *pipelined symbolic taint analysis*. In principle, TaintPipe falls within the second category of taint decoupling work classified above. Essentially, in TaintPipe, threads form multiple pipeline stages, working in parallel. The execution thread of an instrumented application acts as the source of pipeline, which records information needed for taint pipelining, including the control flow data and the concrete execution states when the taint seeds are first introduced. To further reduce the online logging overhead, we adopt a compact profile format and an N-way buffering thread pool. The application thread continues executing and filling in free buffers, while multiple worker threads consume full buffers asynchronously. When each logged data buffer becomes full, an inlined call-back function will be invoked to initialize a taint analysis engine, which conducts taint analysis on a segment of straight-line code concurrently with other worker threads. Symbolic memory access addresses are determined by resolving indirect control transfer targets and approximating the ranges of the symbolic memory indices.

To overcome the challenge of propagating taint tags in a segment without knowing the incoming taint state, TaintPipe performs *segmented symbolic taint analysis*. That is, the taint analysis engine assigned to each segment calculates taint states symbolically. When a concrete taint state arrives, TaintPipe then updates the related taint states by replacing the relevant symbolic taint tags with their correct values. We call this *symbolic taint state resolution*. According to the segment order, TaintPipe sequentially computes the final taint state for every segment, communicates to the next segment, and performs the actual taint checks. Optimizations such as function summary and taint basic block cache offer enhanced performance improvements. Moreover, different from previous DTA tools, supporting bit-level and multi-tag taint analysis are straightforward for TaintPipe. TaintPipe does not require redesign of the structure of shadow memory; instead, each taint tag can be naturally represented as a symbolic variable and propagated with negligible additional overhead.

We have developed a prototype of TaintPipe, a pipelined taint analysis tool that decouples program execution and taint logic, and parallelizes taint analysis on straight-line code segments. Our implementation is built on top of Pin [23], for the pipelining framework, and BAP [5], for symbolic taint analysis. We have evaluated TaintPipe with a variety of applications such as the SPEC CINT2006 benchmarks, a set of common utilities, a list of recent real-life software vulnerabilities, malware, and cryptography functions. The experiments show that TaintPipe imposes low overhead on application runtime performance. Compared with a state-of-the-art inlined dynamic taint analysis tool, TaintPipe achieves overall 2.38 times speedup on SPEC CINT2006, and 2.43 times on a set of common utility programs, respectively. The efficacy experiments indicate that TaintPipe is effective in detecting a wide range of real-life software vulnerabilities, analyzing malicious programs, and speeding up cryptography function detection with multi-tag propagation. Such experimental evidence demonstrates that TaintPipe has potential to be employed by various applications in production systems. The contributions of this paper are summarized as follows:

- We propose a novel approach, TaintPipe, to efficiently decouple conventional inlined dynamic taint analysis by pipelining symbolic taint analysis on segments of straight-line code.
- Unlike previous taint decoupling work, which suffers from frequent communication and synchronization, we demonstrate that with very lightweight runtime value logging, TaintPipe rivals conventional inlined dynamic taint analysis in precision.

- Our approach does not require any specific hardware support or offline preprocessing, so TaintPipe is able to work on commodity hardware instantly.
- TaintPipe is naturally a multi-tag taint analysis method. We demonstrate this capability by detecting cryptography functions in binary with little additional overhead.

The remainder of the paper is organized as follows. Section 2 provides background information and an overview of our approach. Section 3 and Section 4 describe the details of the system design, online logging, and pipelined segmented symbolic taint analysis. We present the evaluation and application of our approach in Section 5. We discuss a few limitations in Section 6. We then present related work in Section 7 and conclude our paper in Section 8.

2 Background

In this section, we discuss the background and context information of the problem that TaintPipe seeks to solve. We start by comparing TaintPipe with the conventional inlined taint analysis approaches, and we then present the differences between the previous "primary & secondary" taint decoupling model and the pipelined decoupling style in TaintPipe.

2.1 Inlined Analysis vs. TaintPipe

Figure 1 ("Inlined DTA") illustrates a typical dynamic taint analysis mechanism based on dynamic binary instrumentation (DBI), in which the original program code and taint tracking logic code are tightly coupled. Especially, when dynamic taint analysis runs on the same core, they compete for the CPU cycles, registers, and cache space, leading to significant performance slowdown. For example, "context switch" happens frequently between the original program instructions and taint tracking instructions due to the starvation of CPU registers. This means there will be a couple of instructions, mostly inserted per program instruction, to save and restore those register values to and from memory. At the same time, taint tracking instructions themselves (e.g., shadow memory mapping) are already complicated enough. One taint shadow memory lookup operation normally needs 6–8 extra instructions [11].

Our approach, analogous to the hardware pipelining, decouples taint logic code to multiple spare cores. Figure 1 ("TaintPipe") depicts TaintPipe's framework, which consists of two concurrently running parts: 1) the instrumented application thread performing lightweight online logging and acting as the source of the pipeline; 2) multiple worker threads as different stages of the pipeline to perform symbolic taint analysis. Each horizontal bar with gray color indicates a working thread. We start online logging when the predefined taint seeds are introduced to the application. The collected profile is passed to a worker thread. Each worker thread constructs a straight-line code segment and then performs taint analysis in parallel. In principle, fully parallelizing dynamic taint analysis is challenging because there are strong serial data dependencies between the taint logic code and application code [31]. To address this problem, we propose *segmented symbolic taint analysis* inside each worker thread whenever the explicit taint information is not available, in which the taint state is symbolically calculated. The symbolic taint state will be updated later when the concrete data arrive. In addition to the control flow profile, the explicit execution state when the taint seeds are introduced is recorded as well. The purpose is to reduce the number of fresh symbolic taint variables.

We use a motivating example to introduce the idea of segmented symbolic taint analysis. Figure 2 shows an example for symbolic taint analysis on a *straight-line code segment*, which is a simplified code snippet of the `libtiff` buffer overflow vulnerability (CVE-2013-4231). Assume when a worker thread starts taint analysis on this code segment (Figure 2(a)), no taint state for the input data ("size" and "num" in our case) is defined. Instead of waiting for the explicit information, we treat the unknown values as taint symbols (`symbol1` for "size" and `symbol2` for "num", respectively) and summarize the net effect of taint propagation in the segment. The symbolic taint states are shown in Figure 2(b). When the explicit taint states are available, we resolve the symbolic taint states by replacing the taint symbols with their real taint tags or concrete values (Figure 2(c)). After that, we continue to perform concrete taint analysis like conventional DTA. Note that here we show pseudo-code for ease of understanding, while TaintPipe works on binary code.

Compared with inlined DTA, the application thread under TaintPipe is mainly instrumented with control flow profile logging code, which is quite lightweight. Therefore, TaintPipe results in much lower application runtime overhead. On the other hand, the execution of taint logic code is decoupled to multiple pipeline stages running in parallel. The accumulated effect of TaintPipe's pipeline leads to a substantial speedup on taint analysis.

2.2 "Primary & Secondary" Model

Some recent work [15, 17, 9] offloads taint logic code from the application (primary) thread to another shadow (secondary) thread and runs them on separate cores. At the same time, the primary thread communicates with

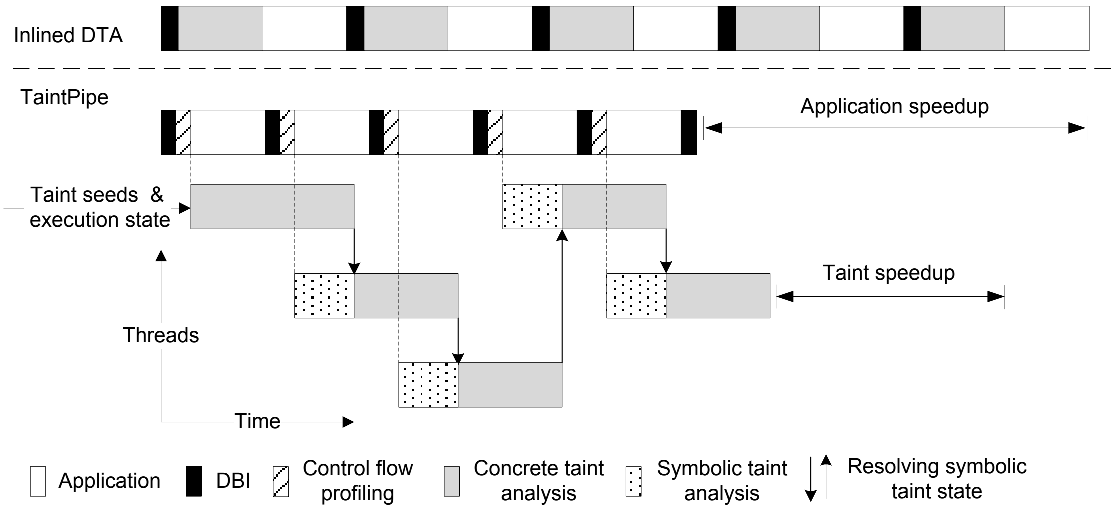


Figure 1: Inlined dynamic taint analysis vs. TaintPipe.

```
size = getc(infile);
A = -1;
B = size + 1;
C = (1 << size) - 1;
D = num & C;
```

(a) Code segment

Output	Taint state
A	0
B	symbol1 + 1
C	(1 << symbol1) − 1
D	symbol2 & ((1 << symbol1) − 1)

(b) Symbolic taint state

Output	Taint state
A	0
B	tag1 + 1
C	(1 << tag1) − 1
D	(1 << tag1) − 1

(c) Resolving symbolic taint state

Figure 2: An example of symbolic taint analysis on a code segment: (a) code segment; (b) symbolic taint states, the input value `size` and `num` are labeled as `symbol1` and `symbol2`, respectively; (c) resolving symbolic taint states when `size` is tainted as `tag1` and `num` is a constant value (num = 0xffffffff).

the secondary thread to convey the necessary information (e.g., the addresses of memory operations and control transfer targets) for performing taint analysis. However, this model suffers from frequent communication between the primary and secondary thread. In principle, every memory address that is loaded or stored has to be logged and transferred. Due to the frequent synchronization with the primary thread and the extra instructions to access shadow memory, taint logic execution in the secondary thread is typically slower than the application execution. As a result, the delay for each taint operation could be accumulated, leading to an delay proportional to the original execution. ShadowReplica [17] partially addresses this drawback by performing advanced offline static optimizations on the taint logic code to reduce the runtime overhead. However, in many security analysis scenarios, precise static analysis and optimizations over taint logic code are not feasible, e.g., reverse engineering and malware forensics. In such cases, program static features such as control flow graphs are possibly obfuscated.

In TaintPipe, we record compact control flow information to reconstruct *straight-line* code, in which all the targets of direct and indirect jumps have been resolved. However, we do not record or transfer the addresses of memory operations. Our key observation is that most addresses of memory operations can be inferred from the straight-line code. For example, if a basic block is ended with an indirect jump instruction `jmp eax`, we can quickly know the value of `eax` from the straight-line code. In this way, all the other memory indirect access calculated through `eax` (before it is updated) can be determined. For instance, we can infer the memory load address for the instruction: `mov ebx, [4*eax + 16]`. Even when the index of a memory lookup is a symbol, with the taint states and path predicates of the straight-line code, we can often narrow down the symbolic memory addresses to a small range in most cases.

Since TaintPipe's data communication is lightweight, TaintPipe can achieve nearly constant delay given enough number of worker threads. The upper limit number of worker threads is also bounded, which equals roughly the ratio of the taint analysis execution time over the application thread execution time for each segment.

Due to TaintPipe's pipelining design, it is possible that TaintPipe may detect an attack some time after the real attack has happened. However, this trade-off does not prevent TaintPipe from practically supporting a broad variety of security applications, such as attack forensic investigation and post-fact intrusion detection, which do not require strict runtime security enforcement. It is worth noting that different from ShadowReplica, TaintPipe does not depend on extensive static analysis to reduce data communication. Therefore, TaintPipe has a wider range of applications in speeding up analyzing obfuscated binaries, as static analysis of obfuscated binaries is of great challenge.

3 Design

3.1 Architecture

Figure 3 illustrates the architecture of TaintPipe. We have built the pipelining framework on top of a dynamic binary instrumentation tool, enabling TaintPipe to work with unmodified program binaries. The steps followed by TaintPipe for pipelining taint analysis are:

1. TaintPipe takes in a binary along with the taint seeds as input. The instrumented application thread starts execution with lightweight online logging for control flow and other information (Section 4.1.1).

2. Then the instrumented program is executed together with a multithreaded logging tool to efficiently deliver the logged data to memory (Section 4.1.2).

3. When the profile buffer becomes full, a taint analysis engine will be invoked for online pipelined taint analysis (Section 4.2.1).

4. The generated log data are then used to construct straight-line code, which helps to solve many precision loss problems in static taint analysis. In this stage, we generate a segment of executed code blocks for each logged data buffer. The memory addresses that are accessed through indirect jump targets are also resolved (Section 4.2.2).

5. The taint analysis engine will further translate straight-line code to taint operations, which avoid precision loss and support both multi-tag and bit-level taint analysis (Section 4.2.3).

6. With the constructed taint operations, TaintPipe performs pipelined symbolic taint analysis. When a thread finishes taint analysis with an explicit taint state, it synchronizes with its following thread to resolve the symbolic taint state (Section 4.2.4).

3.2 Segmented Symbolic Taint Analysis

In this section, we analyze symbolic taint analysis from a theoretical point of view to justify the correctness of our pipelining scheme. In order to formalize *segmented symbolic taint analysis*, we use the following notations:

1. Let σ denote a taint state, which maps variables to their taint tags.

2. Let $\mathscr{A}(\sigma,S)$ denote a symbolic taint analysis $\mathscr{A}$ on a straight-line code segment S, with an initial taint state σ. We use $\mathscr{A}_\sigma(S)$ for convenience.

Note that the straight-line code segment S has no control transfer statement. Conceptually, S only contains one type of statements, namely assignment statements. Of course, from the implementation point of view, there may be other types of statements, but they can all be regarded as assignment statements. For example, as we will show in Section 4.2.3, our taint operations contain assignment operations, laundering operations, and arithmetic operations. The latter two operations can be derived from taint assignment operations.

Based on the semantics of assignment statements, we define symbolic taint analysis for an assignment statement as follows:

$$\mathscr{A}_\sigma(x := e) = \sigma[x \mapsto e^t] \tag{1}$$

where e^t denotes the taint tag of e, and $[\cdot]$ is the taint state update operator. If x is a new variable, the taint state σ is extended with a new mapping from x to its taint. If x occurs in the taint state σ, for the variables in the domain of σ whose symbolic taint expressions depend on x, their symbolic taint expressions will be updated or recomputed with the new taint value of x.

Assume $\sigma_1 = \mathscr{A}_\sigma(i_1)$ for a statement i, then the symbolic taint analysis for two sequential statements $i_1;i_2$ is:

$$\mathscr{A}_\sigma(i_1;i_2) = \mathscr{A}_{\sigma_1}(i_2) \tag{2}$$

Assume straight-line code segment $S_1 = (i_1;S_1')$. We can then deduce the symbolic taint analysis on two sequential segments $S_1;S_2$ as follows:

$$\begin{aligned} &\mathscr{A}_\sigma(S_1;S_2)\\ =&\mathscr{A}_\sigma((i_1;S_1');S_2)\\ =&\mathscr{A}_\sigma(i_1;(S_1';S_2))\\ =&\cdots\\ =&\mathscr{A}_{\mathscr{A}_\sigma(S_1)}(S_2)\end{aligned} \tag{3}$$

That is, given $\mathscr{A}_\sigma(S_1) = \sigma_1$ and $\mathscr{A}_\varepsilon(S_2) = \sigma_2$, where ε is an empty taint state, Eq. 3 leads to:

$$\mathscr{A}_\sigma(S_1;S_2) = \sigma_2[\sigma_1] \tag{4}$$

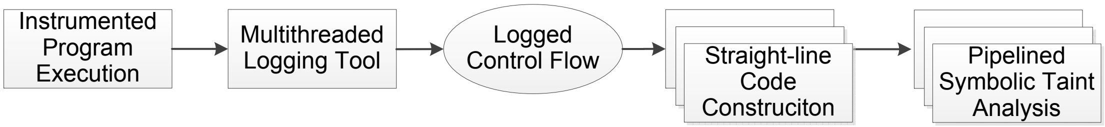

Instrumented Program Execution
Multithreaded Logging Tool
Logged Control Flow
Straight-line Code Construciton
Pipelined Symbolic Taint Analysis

Figure 3: Architecture.

Here, we misuse the taint state update operator $[\cdot]$ and apply it to a taint state map, instead of a single taint variable update. With Eq. 4, we can perform segmented taint analysis in parallel or in a pipeline style. For two segments $S_1;S_2$, assume the starting taint state is σ_0. We start two threads, one compute $\mathscr{A}_{\sigma_0}(S_1)$ and the other computes $\mathscr{A}_{\varepsilon}(S_2)$, where ε is an empty taint state. Assume the result of the first thread analysis is σ_1 and the result of the second is σ_2. The symbolic taint analysis of $S_1;S_2$ is $\sigma_2[\sigma_1]$, that is, the right hand side of Eq. 4. Eq. 4 forms the foundation of our segmented taint analysis in a pipeline style.

4 Implementation

To demonstrate the efficacy of TaintPipe, we have developed a prototype on top of the dynamic binary instrumentation framework Pin [23] (version 2.12) and the binary analysis platform BAP [5] (version 0.8). The online logging and pipelining framework are implemented as Pin tools, using about 3,100 lines of C/C++ code. The taint operation constructors are built on BAP IL (intermediate language). TaintPipe's taint analysis engine is based on BAP's symbolic execution module, using about 4,400 lines of Ocaml and running concurrently with Pin tools. We utilize Ocaml's functor polymorphism so that taint states can be instantiated in either concrete or symbolic style. All of the functionality implemented in taint analysis engine are wrapped as function calls. To support communication between Pin tools and taint analysis engine, we develop a lightweight RPC interface so that each worker thread can directly call Ocaml code. The saving and loading of the taint cache lookup table is implemented using the Ocaml Marshal API, which encodes IL expressions as sequences of compact bytes and then stores them in a disk file.

Dynamic binary instrumentation tools tend to inline compact and branch-less code to the final translated code. For the code with conditional branches, DBI emits a function call instead, which introduces additional overhead. Therefore, we carefully design our instrumentation code to favor DBI's code inlining. To fully reduce online logging overhead, we also utilize Pin-specific optimizations. We leverage Pin's fast buffering APIs for efficient data buffering. For example, the inlined `INS_InsertFillBuffer()` writes the control flow profile directly to the given buffer; the callback function registered in `PIN_DefineTraceBuffer()` processes the buffer when it becomes full or thread exits. Besides, we force Pin to use the fastcall x86 calling convention to avoid emitting stack-based parameter loading instructions (i.e., push and pop). Currently Pin-tools do not support the Pthreads library. Thus we employ Pin Thread API to spawn multiple worker threads. We also implement a counting semaphore based on Pin's locking primitives to assist thread synchronization. Additionally, TaintPipe can be extended to support multithreaded applications with no difficulty by assigning one taint pipeline for each application thread.

4.1 Logging

TaintPipe's pipeline stages consist of multiple threads. The thread of instrumented application (producer) serves as the source of pipeline, and a number of Pin internal threads act as worker threads to perform symbolic taint analysis on the data collected from the application thread. Note that unlike application threads, worker threads are not JITed and therefore execute natively. One of the major drawbacks of previous dynamic taint analysis decoupling approaches is the large amount of information collected in the application thread and the high overhead of communication between the application thread and analysis thread. To address these challenges, TaintPipe performs lightweight online logging to record information required for pipelined taint analysis. The logged data comprise control flow profile and the concrete execution state when taint seeds are first introduced, which is the starting point of our pipelined taint analysis. The initial execution state, consisting of concrete context of registers and memory, (e.g., CR0∼CR4, EFLAGS and addresses of initial taint seeds), is used to reduce the number of fresh symbolic taint variables.

We take major two steps to reduce the application thread slowdown: First, we adopt a compact profile structure so that the profile buffer contains logged data as much as possible, and it is quite simple to recover the entry address of each basic block as well. Second, we apply the "one producer, multiple consumers" model and N-way buffering scheme to process full buffers asynchronously, which allows application to continue execution while pipelined taint analysis works in parallel. We will discuss each step in the following sub-sections.

4.1.1 Lightweight Online Logging

Besides the initial execution state when the taint seeds are introduced, TaintPipe collects control flow information, which is represented as a sequence of basic blocks executed. Conceptually, we can use a single bit to record the direction of conditional jump [29], which leads to a much more compact profile. However, reconstruction straight-line code from 1 bit profile is more complicated to make it fit for offline analysis. Zhao et al. [47] proposed *Detailed Execution Profile* (DEP), a 2-byte profile structure to represent 4-byte basic block address on x86-32 machine. In DEP, a 4-byte address is divided into two parts: *H-tag* for the 2 high bytes and *L-tag* for 2 low bytes. If two successive basic blocks have the same H-tag, only L-tag of each basic block enters the profile buffer; otherwise a special tag 0x0000 followed by the new H-tag will be logged into the buffer.

We extend DEP's scheme to support REP-prefix instructions. A number of x86 instructions related to string operations (e.g., MOVS, LODS) with REP-prefix are executed repeatedly until the counter register (ecx) counts down to 0. Dynamic binary instrumentation tools [23, 4] normally treat a REP-prefixed instruction as an implicit loop and generate a single instruction basic block in each iteration. In our evaluation, there are several cases that unrolling such REP-prefix instructions would be a performance bottleneck. We address this problem by adding additional escape tags to represent such implicit loops. Figure 4 presents an example of the control flow profile we adopted. The left part shows a segment of straight-line code containing 1028 basic blocks, and 1024 out of them are due to REP-prefixed instruction repetitions. Our profile (the right side of Figure 4) encodes such case with two consecutive escape tags (0xffff), followed by the number of iterations (0x0400).

We note that it is usually unnecessary to turn on the logging all the time. For example, when application starts executing, many functions are only used during loading. At that time, no sensitive taint seed is introduced. Therefore we perform on-demand logging to record control flow profile when necessary. As application starts running, we only instrument limited functions to inspect the various input channels that taint could be introduced into the application (taint seeds). Such taint seeds include standard input, files and network I/O. Besides, users can customize other values as taint seeds, such as function return values or parameters. When the pre-defined taint seeds are triggered, we turn on the control flow profile logging. At the same time, we save the current execution state to be used in the pipelined taint analysis. Many well-known library functions have explicit semantics, which facilitates us to selectively turn off logging inside these functions and propagate taint correspondingly at function level. We will discuss this issue further in Section 4.2.3.

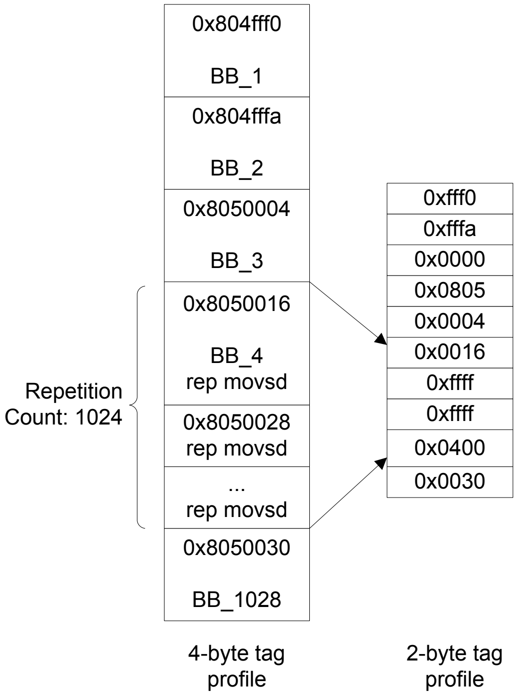


Figure 4: An example of 2-byte tag profile.

4.1.2 N-way Buffering Scheme

Since TaintPipe's online logging is lightweight, application (producer) thread's execution speed is typically faster than the processing speed of worker threads. To mitigate this bottleneck, we employed "one producer, multiple consumers" model and *N-way buffering scheme* [46]. At the center of our design is a thread pool, which is subdivided into n linked buffers, and the producer thread and multiple worker threads work on different buffers simultaneously. More specifically, when the instrumented application thread starts running, we first allocate n linked empty buffers ($n > 1$). At the same time, n Pin internal threads (worker threads) are spawned. Each worker thread is bound to one buffer and communicates with the application thread via semaphores. When a buffer becomes full, the application thread will release the full buffer to its corresponding worker thread and then continue to fill in the next available empty buffer. Given a full profile buffers, a worker thread will send it to a taint analysis engine to perform concrete/symbolic taint analysis in parallel. After that, the worker thread will release the profile buffer back to the application thread and wait for processing the next full buffer.

It is apparent that the availability of unused worker threads and the size of profile buffer will affect overall performance of TaintPipe (both application execution time and pipelined taint analysis) significantly. In Section 5.1, we will conduct a series of experiments to find the optimal values for these two factors.

4.2 Symbolic Taint Analysis

4.2.1 Taint Analysis Engine

When the application thread releases a full profile buffer, a worker thread is waked up to capture the profile buffer and then communicates with a taint analysis engine for pipelined taint analysis. The taint analysis engine will first convert the control flow profile to a segment of straight-line intermediate language (IL) code and then translates the IL code to even simpler taint logic operations. The translations are cached for efficiency at taint basic block level. The key components of taint analysis engine are illustrated in Figure 5.

The core of TaintPipe's taint analysis engine is an abstract taint analysis processor, which simulates a segment of taint operations and updates the taint states accordingly. The taint state structure contains two contexts: virtual registers keeping track of symbolic taint tags for register, and taint symbolic memory for symbolic taint tags in memory. The taint symbolic memory design is like the two-level page table structure and each page of memory consists of symbolic taint formulas rather than concrete values. After the initialization of the symbolic taint inputs, the engines perform taint analysis either concretely or symbolically in a pipeline style.

4.2.2 Straight-line Code Construction

Given the control flow profiles, recovering each basic block's H-tag and L-tag is quite straightforward. A basic block's entry address is the concatenation of its corresponding H-tag and L-tag [47]. The taint analysis engine should only execute the instructions required for taint propagation. Otherwise, the work thread may run much slower than the application thread. On the other side, due to the cumbersome x86 ISA, precisely propagating taint for the complex x86 instructions is an arduous work, especially for some instructions with side effect of conditional taint (e.g., `CMOV`). To achieve these two goals, we first extract the x86 instructions sequence from the application binary and then lift them to BIL [5], a RISC-like intermediate language. Since we know exactly the execution sequence, the sequence is a straight-line code. We have removed all the direct and indirect control transfer instructions and substituted them with control transfer target assertion statements.

After resolving an indirect control transfer, we go one step further to determine all the memory operation addresses which depend on this indirect control transfer target. For example, after we know the target of `jmp eax`, we continue to trace the use-def chain of `eax` for each memory load or store operation whose address is calculated through this `eax`. With the initial execution state (containing addresses of taint seeds) and indirect control transfer target resolving, we are able to decide most of the memory operation addresses.

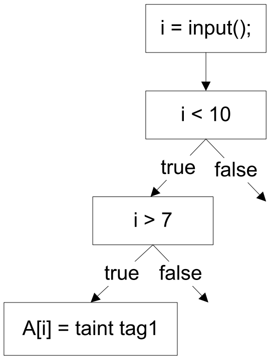


Figure 6: A path predicate constrains symbolic memory access within the boundary of $7 < i < 10$.

For some applications such as word processing, a symbolic taint input may be used as a memory lookup index. Without any constraint, a symbolic memory index could point to any memory cell. Inspired by the index-based memory model proposed by Cha et al. [8], we attempt to narrow down the symbolic memory accesses to a small range with symbolic taint states and path predicates. We first leverage value set analysis [2] to limit the range of a symbolic memory access and then refine the range by querying a constraint solver. The path predicate along the straight-line code usually limits the scope of symbolic memory access. Figure 6 shows such an example where the path predicate restricts the symbolic memory index i within a range such that $7 < i < 10$. When propagating a taint tag to the memory cell referenced by i, we conservatively taint all the possible memory slots, that is, `A[8]` and `A[9]` in Figure 6 will be tainted as `tag1`. In Section 5.3, we will demonstrate that our symbolic memory index solution only introduces marginal side effects.

4.2.3 Taint Operation Generation

Based on BIL statements, we construct taint operations. Taint operations inside a basic block are formed as "taint basic block" [37], which are cached for efficiency. To make the best of cache effect, we merge the basic blocks with only one predecessor and one successor. Since BIL explicitly reveals the side effect of intricate x86 instructions, it is easy to perform intra-block optimizations to get rid of redundant taint operations. Therefore, our taint

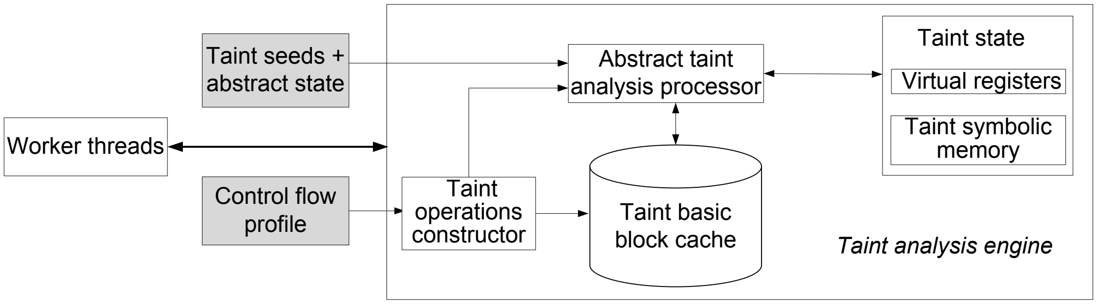


Figure 5: The structure of taint analysis engine.

operations are simple and accurate. Currently our taint operations mainly consist of three types of operations for tracking taint data:

1. Assignment operations: The operations in this category are involved in copying values between registers and registers/memory. We simply assign the taint tag of source operand to the destination operand.

2. Laundering operations: The operations are used to clean the taint tag of the destination operand. For example, `xor eax,eax` will clean the taint result of `eax`. We identify all laundering operations in taint basic blocks and substitute them with assignment operations.

3. Arithmetic and logic operations: This category of operations are the most difficult to handle. We emulate arithmetic and logic operations on the taint tags to capture their real semantics.

Figure 7 presents an example of taint operations for a basic block. TaintPipe's symbolic taint operations outperform conventional DTA approaches in three ways. First and foremost, multi-tag taint analysis is straightforward for TaintPipe. Each symbolic variable can naturally represent a taint tag (see Line 1 and Line 2 in Figure 7). Second, previous DTA tools mostly adopted a "short circuiting" method to handle arithmetic operations, that is, the destination is tainted if at least one of the source operands is tainted regardless of the real semantics. However, in many scenarios, it will lead to precision loss. Check the code at Line 4 and 5 in Figure 7, value *d* will always be zero since *b* is the negation of *a*. Unfortunately, some previous work may label *d* as tainted incorrectly [41]. Third, different from related work [20, 17], TaintPipe supports bit-level taint for EFLAGS register, representing whether a bit of the EFLAGS is tainted or not due to side effects. Recent work has demonstrated the value of bit-level taint in binary code de-obfuscation [42].

```
int a, b, c, d;
1: a = read ();          1: Taint (a) = tag1;
2: c = read ();          2: Taint (c) = tag2;
3: c = c xor c;          3: Taint (c) = 0;
4: b = ~ a;              4: Taint (b) = ~ tag1;
5: d = a & b;            5: Taint (d) = 0;
```

(a) a basic block (b) a taint basic block

Figure 7: Example: taint operations.

Table 1: Function summary.

Category	Function
No tainting	strcmp, strncmp, memcmp, strlen, strchr, strstr, strpbrk, strcspn, qsort, rand, time, clock, ctime
Function level	strcat, strncat, strcpy, strncpy, memcpy, memmove, strtok, atoi, itoa, abs tolower, toupper

Another major optimization we adopt is so called "function summary". As many well-known library functions have explicit semantics (e.g., atoi, strlen), we generate a summary of each function and propagate taint correspondingly at function level. Table 1 lists two types of function summary TaintPipe supports: 1) Functions within "no tainting" category do not have any side effect on taint state. We can safely turn off logging when executing them. 2) Some functions do propagate taint from an input parameter to output. We still turn off logging and update taint state correspondingly when these functions return.

4.2.4 Symbolic Taint State Resolution

In TaintPipe's pipeline framework, a worker thread may perform taint analysis concretely or symbolically in parallel. When a worker thread completes taint analysis with concrete taint tags, the final taint state it maintains is deterministic. Then it synchronizes with the subsequent worker thread to resolve the symbolic taint state maintained by the latter. Taint states are allocated in a shared memory area so that multiple threads can access them easily. Basically, given the concrete taint state at the beginning of a code segment, we replace a symbolic taint tag with the appropriate starting value (either a taint tag or a concrete value). We further update all the symbolic formula containing that symbolic taint tag. For example, the logic AND formula in Figure 2(b) will be simplified to a single taint tag. After that, the subsequent thread switches to the concrete taint analysis and continue processing the left segment code.

In this order, the taint states of each segment will be resolved and updated one by one. The defined taint policy (e.g., a function return value should not be tainted) is checked along the concrete taint analysis as well. A tainted sink is identified if it contains a symbolic formula; multiple tags are determined by counting the number of different symbols in the formula. Note that a previous hardware-assisted approach [31] utilized a separate "master" processor to update each segment's taint status sequentially. However, as pointed out by the paper, when there are more than a few "worker" processors, the master processor will become the bottleneck. Our approach amortizes the workload of the master processor to each worker thread.

5 Experimental Evaluation

We conducted experiments with several goals in mind. First, we wanted to choose optimal values for two factors that may affect TaintPipe's performance, namely control flow profile buffer size and the number of worker threads. Then we studied overall runtime overhead when running TaintPipe on the SPEC2006 int benchmarks and a number of common utilities. We also compared TaintPipe with a highly optimized inline dynamic taint analysis tool. At the same time, we wanted to make sure TaintPipe is effective in speeding up various security analysis tasks and can compete with conventional inlined dynamic taint analysis in precision. To this end, we demonstrated three compelling applications: 1) detecting software attacks; 2) tracking information flow in obfuscated malicious programs; and 3) identifying cryptography functions with multi-tag propagation.

5.1 Experiment Setup

Our experiment platform contains two Intel Xeon E5-2690 processors, 128GB of memory and a 250GB solid state drive, running Ubuntu12.04. Each processor is equipped with 8 2.9GHz cores, 16 hyper threads and 20MB L3 cache. The performance data reported in this section are all mean values with 5 repetitions.

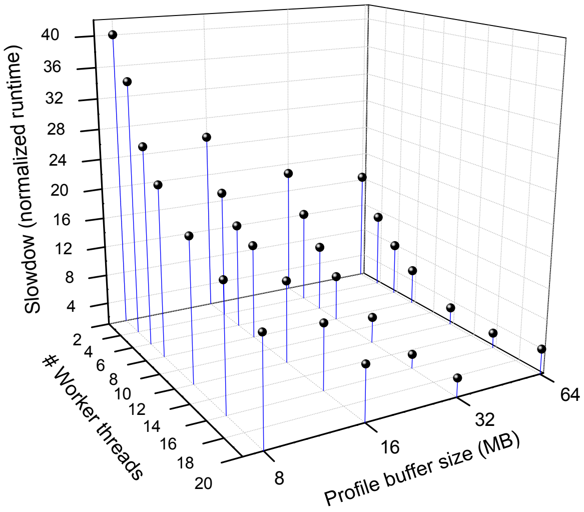


Figure 8: Optimal buffer size and number of worker threads.

TaintPipe's performance is affected by the size of the profile buffers and the number of worker threads. Generally, the more worker threads and larger profile buffers, the less possibility that an application is suspended to wait for a free buffer. On the other hand, our taint analysis engine has to take longer time to process larger segment code. We conducted a series of tests with the SPEC CPU2006 int benchmarks, under different settings of these two variables. We dynamically adjust the number of worker threads from 2 to 20 (2, 4, 6, 8, 12, 16 and 20), and profile size from 8MB to 64MB (8MB, 16MB, 32MB and 64MB). Figure 8 displays the experimental results. Roughly, as number of worker threads and buffer size increase, the application slowdown reduces. That is mainly because large buffer sizes allow application thread continue to fill up and worker threads spend less time on synchronization. After a certain point (buffer size $\geq$ 32MB and number of worker threads $>$ 16), overhead increases slightly. Two factors prevent TaintPipe from achieving more speedup. First, taint analysis engine slows down when processing large code segment. Second, more worker threads introduce larger communication latency when resolving symbolic taint states. According to the results, we set the two factors as their optimal values (32MB buffer size and 16 worker threads), which will be used in the following experiments.

5.2 Performance

To evaluate the performance gains achieved by pipelining taint logic, we compared TaintPipe with a state-of-the-art tool, libdft [20], which performs inlined dynamic taint analysis based on Pin ("libdft" bar). In addition, we developed a simple tool to measure the slowdown im-

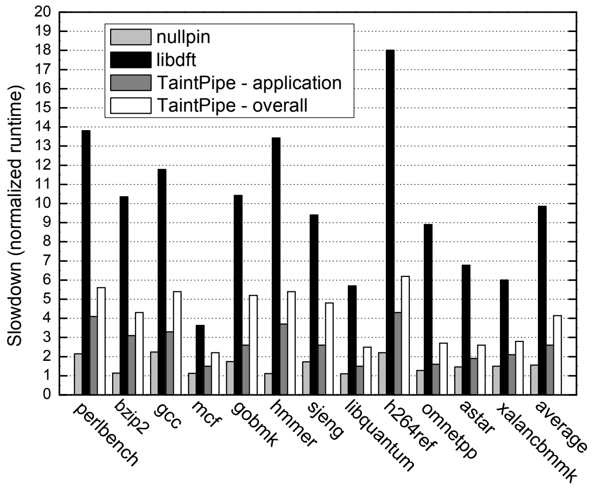


Figure 9: Slowdown on SPEC CPU2006.

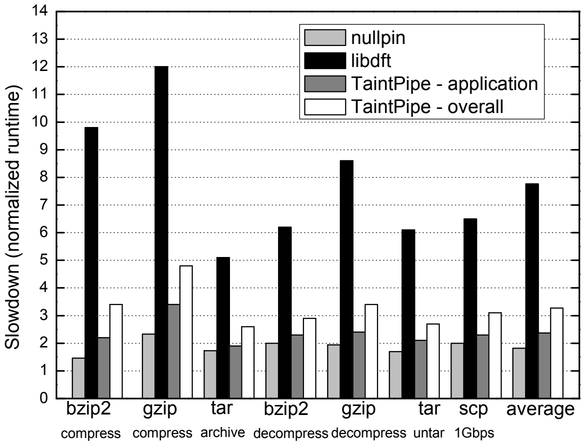


Figure 10: Slowdown on common Linux utilities.

posed exclusively by TaintPipe. It runs a program under Pin without any form of analysis ("nullpin" bar). The "TaintPipe - application" bar represents the running time of instrumented application thread alone, and "TaintPipe - overall" corresponds to the overall overhead when both the application thread and pipelined worker threads are running. The major reason we reported "TaintPipe - application" and "TaintPipe - overall" time separately is to show the two improvements, namely "Application speedup" and "Taint speedup" (see Figure 1). Since the application thread typically runs faster than the worker threads, the "TaintPipe - overall" time is actually dominated by the worker threads. Therefore, usually the "TaintPipe - overall" time represents the relative time spent by worker threads as well. The times reported in this section are all normalized to native execution, that is, application running time without dynamic binary instrumentation.

SPEC CPU2006. Figure 9 shows the normalized execution times when running the SPEC CPU2006 int benchmark suite under TaintPipe. On average, the instrumented application thread enforces a 2.60X slowdown to native execution, while the overall slowdown of TaintPipe is 4.14X. If we take Pin's environment runtime overhead ("nullpin" bar) as the baseline, we can see TaintPipe imposes 2.67X slowdown ("TaintPipe - overall" / "nullpin") and libdft introduces 6.4X slowdown—this number is coincident to the observation that propagating a taint tag normally requires extra 6–8 instructions [30, 11]. In summary, TaintPipe outperforms inlined dynamic taint analysis drastically: 2.38X faster than the inlined dynamic taint analysis, and 3.79X faster in terms of application execution. In the best case (*h264ref*), the application speedup under TaintPipe exceeds 4.18X.

Utilities. We also evaluated TaintPipe on four common Linux utilities, which were not chosen randomly. These four utilities represent three kinds of workloads: I/O bounded (`tar`), CPU bounded (`bzip2` and `gzip`), and the case in-between (`scp`). We applied `tar` to archive and extract the GNU Core utilities package (version 8.13) (∼50MB), then we employed `bzip2` and `gzip` to compress and decompress the archive file. Finally we utilized `scp` to copy the archive file over a 1Gbps link. As shown in Figure 10, TaintPipe reduced slowdown of dynamic taint analysis from 7.88X to 3.24X, by a factor of 2.43 on average.

Effects of Optimizations. In this experiment, we quantify the effects of taint logic optimizations we presented in Section 4.2, which are paramount for optimized TaintPipe performance. Figure 11 shows the impact of these optimizations when applied cumulatively on SPEC CPU2006 and the set of common utilities. The "unopt" bar approximates an un-optimized TaintPipe, which does not adopt any optimization method. The "O1" bar indicates the optimization of function summary, reducing application slowdown notably by 26.6% for SPEC CPU2006 and 25.0% for the common utilities. The "O2" bar captures the effect of taint basic block cache, leading to a further reduction by 19.0% and 22.9% for SPEC and utilities, respectively. Intra-block optimizations, denoted by "O3", offer further improvement, 12.0% with SPEC and 11.6% with the utilities).

Table 2: Tested software vulnerabilities.

Program	Vulnerability	CVE ID	# Taint Bytes		
			libdft	Temu	TaintPipe
Nginx	Validation Bypass	CVE-2013-4547	45	45	45
Micro_httpd	Validation Bypass	CVE-2014-4927	80	85	80
Tiny Server	Validation Bypass	CVE-2012-1783	125	126	125
Regcomp	Validation Bypass	CVE-2010-4052	1,124	1,148	1,180
Libpng	Denial Of Service	CVE-2014-0333	72	72	72
Gzip	Integer Underflow	CVE-2010-0001	94	112	96
Grep	Integer Overflows	CVE-2012-5667	608	682	653
Coreutils	Buffer Overflow	CVE-2013-0221	252	260	256
Libtiff	Buffer Overflow	CVE-2013-4231	268	286	290
WaveSurfer	Buffer Overflow	CVE-2012-6303	384	418	406
Boa	Information Leak	CVE-2009-4496	164	164	164
Thttpd	Information Leak	CVE-2009-4491	328	328	328

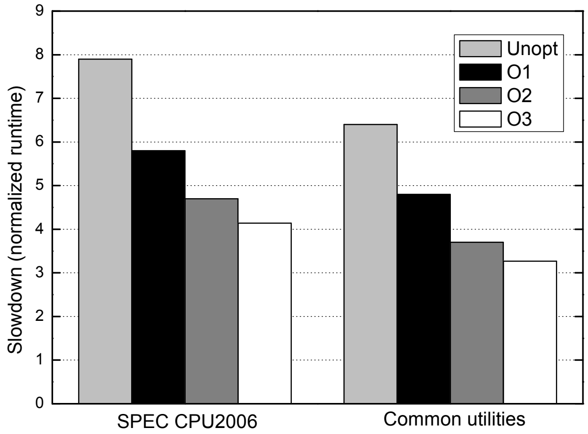


Figure 11: The impact of optimizations to speed up TaintPipe when applied cumulatively: O1 (function summary), O2 (O1 + taint basic block cache), O3 (O2 + intra-block optimizations).

Table 3: Malware samples and taint graphs.

Sample	Type	Taint Graph		Control Flow Obfuscation
		Node #	Edge #	
Svat	Virus	90	62	
RST	Virus	154	82	
Agent	Rootkit	624	402	✓
KeyLogger	Trojan	554	368	✓
Subsevux	Backdoor	1648	764	✓
Tsunami	Backdoor	734	534	✓
Keitan	Backdoor	618	482	✓
Fireback	Backdoor	1038	620	✓

5.3 Security Applications

Software Attack Detection. One important application of taint analysis is to define taint policies, and ensure they are not violated during taint propagation. We tested TaintPipe with 12 recent software exploits listed in Table 2, which covers a wide range of real-life software vulnerabilities. For example, the vulnerabilities in `nginx`, `micro_httpd`, and `tiny server` allow remote attackers to bypass input validation and crash the program. The `libtiff` buffer overflow vulnerability leads to an out of bounds loop limit via a malformed gif image. Both `boa` and `thttpd` write data to a log file without sanitizing non-printable characters, which may be exploited to execute arbitrary commands. Since we have detailed vulnerability reports, we can easily mark the locations of taint sinks in the straight-line code and set corresponding taint policies.

In our evaluation, TaintPipe did not generate any false positives and successfully identified taint policy violations while incurring only small overhead. At the same time, we evaluated the accuracy of TaintPipe. To this end, we counted the total number of tainted bytes in the taint state when taint analysis hit the taint sinks. Column 4 $\sim$ 6 of Table 2 show the number of taint bytes when running libdft, Temu [44] and TaintPipe, respectively. Compared with the inlined dynamic taint analysis tools (libdft and Temu), TaintPipe's symbolic taint analysis achieves almost the same results in 8 cases and introduces only a few additional taint bytes in the other 4 cases. We attribute this to our conservative approach to handling of symbolic memory indices. The evaluation data show that TaintPipe does not result in over-tainting [32] and rivals the inlined dynamic taint analysis at the same level of precision.

Table 4: Cryptographic function detection time.

Algorithm	TaintPipe (s)	Temu (s)
TEA	3.8 (<1.1X)	15.2 (2.2X)
AES-CBC	12.3 (1.2X)	125.6 (3.8X)
Blowfish	4.5 (<1.1X)	21.4 (2.5X)
MD5	7.4 (<1.1X)	35.1 (2.6X)
SHA-1	8.8 (1.1X)	40.2 (3.3X)

Generating Taint Graphs for Malware. We ran 8 malware samples collected from `VX Heavens`[1] with TaintPipe.[2] Similar to Panorama [43], we tracked information flow and generated a *taint graph* for each sample. In a taint graph, nodes represent taint seeds or instructions operating on taint data, and a directed edge indicates an explicit data flow dependency between two nodes. Taint graph faithfully describes intrinsic malicious intents, which can be used as malware specification to detect suspicious samples [12]. The statistics of our testing results are presented in Table 3. It is worth noting that 6 out of 8 malware samples are applied with various control flow obfuscation methods (the fifth column), such as opaque predicates, control flow flattening, obfuscated control transfer targets, and call stack tampering. As a result, the control flow graphs are heavily cluttered. For example, malware samples `Keitan` and `Fireback` have a relatively high ratio of indirect jumps (e.g., `jmp eax`). Typically it is hard to precisely infer the destination of an indirect jump statically. Thus, the taint logic optimization methods that rely on accurate control flow graph [17, 18] will fail. In contrast, our approach does not rely on control flow graph and therefore we analyzed these obfuscated malware samples smoothly.

Cryptography Function Detection. Malware authors often use cryptography algorithms to encrypt malicious code, sensitive data, and communication. Detecting cryptography functions facilitates malware analysis and forensics. Recent work explored the *avalanche effect* to quickly identify possible cryptography functions by observing the input-output dependency with multi-tag taint analysis. That is, each byte in the encrypted message is dependent on almost all bytes of input data or key [7, 21, 48]. However, multi-tag dynamic taint analysis normally has to sacrifice more shadow memory and imposes much higher runtime overhead than single-tag dynamic taint analysis. Recall that multi-tag propagation is handled transparently in TaintPipe. In this experiment, we applied TaintPipe to detect such avalanche effects in binary code. We utilized the test case suite of Crypto++ library[3] and tested 5 cryptography algorithms. Each byte of the plain messages was labeled as a different taint tag. We compared TaintPipe with Temu [44], which supports multiple byte-to-byte taint propagation as well.[4] The detection time is shown in Table 4. We also reported the ratio of multi-tag's running time to single-tag's. The results show that TaintPipe is able to detect cryptographic functions with little additional overhead (less than 1.1X on average), while Temu's multi-tag propagation imposes a significant slowdown (2.9X to single-tag propagation on average).

6 Discussions and Limitations

Since TaintPipe's pipelining design leads to an asynchronous taint check, TaintPipe may detect a violation of taint policy after the real attack happens. One possible solution is to provide *synchronous* policy enforcement at critical points (e.g., indirect jump and system call sites). In that case, we can explicitly suspend the application thread, and wait for the worker threads to complete. Our current design spawns worker threads in the same process of running both Pin and the application. In the future, we plan to replace the worker threads with different processes to increase isolation.

As TaintPipe may perform symbolic taint analysis when explicit taint states are not available, TaintPipe exhibits similar limitations as symbolic execution of binaries. Recent work MAYHEM [8] proposes an advanced index-based memory model to deal with symbolic memory index. We plan to extend our symbolic memory index handling in the future. TaintPipe recovers the straight-line code by logging basic block entry address. However, with malicious self-modifying code, the entry address may not uniquely identify a code block. To address this issue, we can augment TaintPipe by logging the real executed instructions at the expense of runtime performance overhead.

Our focus is to demonstrate the feasibility of pipelined symbolic taint analysis. We have not fully optimized the symbolic taint analysis part which we believe can be greatly improved in terms of performance based on our current prototype. As our taint analysis engine simulates the semantics of taint operations, the speed of taint analysis is slow. One future direction is to execute concrete taint analysis natively like micro execution [16] and switch to the interpretation-style when performing symbolic taint analysis. Currently TaintPipe requires large share memory to reduce communication overhead between different pipeline stages. Therefore, our approach is more suitable for large servers with sufficient memory.

[1] http://vxheaven.org

[2] All these 8 samples are not packed. To analyze packed binaries, we can start TaintPipe when the unpacking procedure arrives at the original entry point.

[3] http://www.cryptopp.com/

[4] libdft does not support multi-tag taint analysis.

7 Related Work

In this section we first present previous work on static and dynamic taint analysis. Our work is a hybrid of these two analyses. Then we introduce previous efforts on taint logic code optimization, which benefits our taint operation generation. Finally, we describe recent work on decoupling taint tracking logic from original program execution, which is the closest to TaintPipe's method.

Static and Dynamic Taint Analysis. Since static taint analysis (STA) is performed prior to execution by considering all possible execution paths, it does not affect application runtime performance. STA has been applied to data lifetime analysis for Android applications [1], exploit code detection [36], and binary vulnerability testing [28]. Dynamic taint analysis (DTA) is more precise than static taint analysis as it only propagates taint following the real path taken at run time. DTA has been widely used in various security applications, including data flow policy enforcement [25, 40, 27], reversing protocol data structures [33, 38, 6], malware analysis [39] and Android security [14]. However, an intrinsic limitation of DTA is its significant performance slowdown. Schwartz et al. [32] formally defined the operational semantics for DTA and forward symbolic execution (FSE). Our approach is in fact a hybrid of these techniques. Worker thread conducts concrete taint analysis (like DTA) whenever explicit taint information is available; otherwise symbolic taint analysis (like STA and FSE) is performed.

Taint Logic Optimization. Taint logic code, deciding whether and how to propagate taint, require additional instructions and "context switches". Frequently executing taint logic code incurs substantial overhead. Minemu [3] achieved a decent runtime performance at the cost of sacrificing memory space to speed up shadow memory access. Moreover, Minemu utilized spare SSE registers to alleviate the pressure of general register spilling. As a result, Minemu only worked on 32-bit program. TaintEraser [49] developed function summaries for Windows programs to propagate taint at function level. Libdft [20] introduced two guidelines to facilitate DBI's code inlining: 1) tag propagation code should have no branch; 2) shadow memory updates should be accomplished with a single assignment. Ruwase et al. [30] applied compiler optimization techniques to eliminate redundant taint logic code in hot paths. Jee et al. [19] proposed *Taint Flow Algebra* to summarize the semantics of taint logic for basic blocks. All these efforts to generate optimized taint logic code are orthogonal and complementary to TaintPipe.

Decoupling Dynamic Taint Analysis. A number of researchers have considered the high performance penalty imposed by inlined dynamic taint analysis. They proposed various solutions to decouple taint tracking logic from application under examination [24, 31, 26, 15, 17, 9], which are close in spirit to our proposed approach. Speck [26] forked multiple taint analysis processes from application execution to spare cores by means of speculative execution, and utilized record/replay to synchronize taint analysis processes. Speck required OS level support for speculative execution and rollback. Speck's approach sacrifices processing power to achieve acceleration. Similar to TaintPipe's segmented symbolic taint analysis, Ruwase et al. [31] proposed *symbolic inheritance tracking* to parallelize dynamic taint analysis. TaintPipe differs from Ruwase et al.'s approach in three ways: 1) Their approach was built on top of a log-based architecture [10] for efficient communication with idle cores, while TaintPipe works on commodity multi-core hardware directly. 2) To achieve better parallelization, they adopted a relaxed taint propagation policy to set a binary operation as untainted, while TaintPipe performs full-fledged taint propagation so that we provide stronger security guarantees. 3) They used a separate "master" processor to update each segment's taint status sequentially, while TaintPipe resolves symbolic taint states between two consecutive segments. Our approach could achieve better performance when there are more than a few "worker" processors.

Software-only approaches [15, 17, 9] are the most related to TaintPipe. They decouple dynamic taint analysis to a shadow thread by logging the runtime values that are needed for taint analysis. However, as we have pointed out, these methods [15, 9] may suffer from high overhead of frequent communication between the application thread and shadow thread. Recent work ShadowReplica [17] ameliorates this drawback by adopting fine-grained offline optimizations to remove redundant taint logic code. In principle, it is possible to remove redundant taint logic by means of static offline optimizations. Unfortunately, even static disassembly of stripped binaries is still a challenge [22, 35]. Therefore, the assumption by ShadowReplica that an accurate control flow graph can be constructed may not be feasible in certain scenarios, such as analyzing control flow obfuscated software. We take a different angle to address this issue with lightweight runtime information logging and segmented symbolic taint analysis. We demonstrate the capability of TaintPipe in speeding up obfuscated binary analysis, which ShadowReplica may not be able to handle. Furthermore, ShadowReplica does not support bit-level and multi-tag taint analysis, while TaintPipe handles them naturally.

8 Conclusion

We have presented TaintPipe, a novel tool for pipelining dynamic taint analysis with segmented symbolic taint analysis. Different from previous parallelization work on taint analysis, TaintPipe uses a pipeline style that relies on straight-line code with very few runtime values, enabling lightweight online logging and much lower runtime overhead. We have evaluated TaintPipe on a number of benign and malicious programs. The results show that TaintPipe rivals conventional inlined dynamic taint analysis in precision, but with a much lower online execution slowdown. The performance experiments indicate that TaintPipe can speed up dynamic taint analysis by 2.43 times on a set of common utilities and 2.38 times on SPEC2006, respectively. Such experimental evidence demonstrates that TaintPipe is both efficient and effective to be applied in real production environments.

9 Acknowledgments

We thank the Usenix Security anonymous reviewers and Niels Provos for their valuable feedback. This research was supported in part by the National Science Foundation (NSF) grants CNS-1223710 and CCF-1320605, and the Office of Naval Research (ONR) grant N00014-13-1-0175. Liu was also partially supported by ARO W911NF-09-1-0525.

References

[1] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)* (2014).

[2] BALAKRISHNAN, G., AND REPS, T. WYSINWYX: What You See Is Not What You eXecute. *ACM transactions on programming languages and systems 32*, 6 (2010).

[3] BOSMAN, E., SLOWINSKA, A., AND BOS, H. Minemu: The world's fastest taint tracker. In *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID'11)* (2011).

[4] BRUENING, D., GARNETT, T., AND AMARASINGHE, S. An infrastructure for adaptive dynamic optimization. In *Proceedings of the 2003 international symposium on code generation and optimization (CGO'03)* (2003).

[5] BRUMLEY, D., JAGER, I., AVGERINOS, T., AND SCHWARTZ, E. J. BAP: A binary analysis platform. In *Proceedings of the 23rd international conference on computer aided verification (CAV'11)* (2011).

[6] CABALLERO, J., POOSANKAM, P., KREIBICH, C., AND SONG, D. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM Conference on Computer and Communication Security (CCS'09)* (2009).

[7] CABALLERO, J., POOSANKAM, P., MCCAMANT, S., BABI Ć, D., AND SONG, D. Input generation via decomposition and re-stitching: Finding bugs in malware. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)* (2010).

[8] CHA, S. K., AVGERINOS, T., REBERT, A., AND BRUMLEY, D. Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (2012).

[9] CHABBI, M., PERIYANAYAGAM, S., ANDREWS, G., AND DEBRAY, S. Efficient dynamic taint analysis using multicore machines. Tech. rep., The University of Arizona, May 2007.

[10] CHEN, S., GIBBONS, P. B., KOZUCH, M., AND MOWRY, T. C. Log-based architectures: Using multicore to help software behave correctly. *ACM SIGOPS Operating Systems Review 45*, 1 (2011), 84–91.

[11] CHENG, W., ZHAO, Q., YU, B., AND HIROSHIGE, S. TaintTrace: Efficient flow tracing with dynamic binary rewriting. In *Proceedings of the 11th IEEE Symposium on Computers and Communications (ISCC'06)* (2006).

[12] CHRISTODORESCU, M., JHA, S., AND KRUEGEL, C. Mining specifications of malicious behavior. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE'07)* (2007).

[13] CLAUSE, J., LI, W. P., AND ORSO, A. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2007)* (2007).

[14] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 2010 USENIX Symposium on Operating Systems Design and Implementation (OSDI'10),* (2010).

[15] ERMOLINSKIY, A., KATTI, S., SHENKER, S., FOWLER, L. L., AND MCCAULEY, M. Towards practical taint tracking. Tech. rep., EECS Department, University of California, Berkeley, Jun 2010.

[16] GODEFROID, P. Micro execution. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)* (2014).

[17] JEE, K., KEMERLIS, V. P., KEROMYTIS, A. D., AND PORTOKALIDIS, G. ShadowReplica: Efficient parallelization of dynamic data flow tracking. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security (CCS'13)* (2013).

[18] JEE, K., PORTOKALIDIS, G., KEMERLIS, V. P., GHOSH, S., AUGUST, D. I., AND KEROMYTIS, A. D. A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In *Proceedings of the 19th Internet Society (ISOC) Symposium on Network and Distributed System Security (NDSS)* (2012).

[19] JEE, K., PORTOKALIDIS, G., KEMERLIS, V. P., GHOSH, S., AUGUST, D. I., AND KEROMYTIS, A. D. A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In *Proceedings of the 2012 Network and Distributed System Security Symposium (NDSS'12)* (2012).

[20] KEMERLIS, V. P., PORTOKALIDIS, G., JEE, K., AND KEROMYTIS, A. D. libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'12)* (2012).

[21] Li, X., Wang, X., and Chang, W. CipherXRay: Exposing cryptographic operations and transient secrets from monitored binary execution. *IEEE Transactions on Dependable and Secure Computing 11*, 2 (2014).

[22] Linn, C., and Debray, S. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS'03)* (2003).

[23] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI'05)* (2005).

[24] Nagarajan, V., Kim, H.-S., Wu, Y., and Gupta, R. Dynamic information flow tracking on multicores. In *Proceedings of the 2008 Workshop on Interaction between Compilers and Computer Architectures* (2008).

[25] Newsome, J., and Song, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 2005 Network and Distributed System Security Symposium (NDSS'05)* (2005).

[26] Nightingale, E. B., Peek, D., Chen, P. M., and Flinn, J. Parallelizing security checks on commodity hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)* (2008).

[27] Qin, F., Wang, C., Li, Z., seop Kim, H., Zhou, Y., and Wu, Y. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)* (2006).

[28] Rawat, S., Mounier, L., and Potet, M.-L. Static taint-analysis on binary executables. `http://stator.imag.fr/w/images/2/21/Laurent_Mounier_2013-01-28.pdf`, October 2011.

[29] Renieris, M., Ramaprasad, S., and Reiss, S. P. Arithmetic program paths. 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)* (2005).

[30] Ruwase, O., Chen, S., Gibbons, P. B., and Mowry, T. C. Decoupled lifeguards: Enabling path optimizations for online correctness checking tools. In *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation (PLDI'10)* (2010).

[31] Ruwase, O., Gibbons, P. B., Mowry, T. C., Ramachandran, V., Chen, S., Kozuch, M., and Ryan, M. Parallelizing dynamic information flow tracking lifeguards. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'08)* (2008).

[32] Schwartz, E. J., Avgerinos, T., and Brumley, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy* (2010).

[33] Slowinska, A., Stancescu, T., and Bos, H. Howard: A dynamic excavator for reverse engineering data structures. In *Proceedings of the 2011 Network and Distributed System Security Symposium (NDSS'11)* (2011).

[34] Vachharajani, N., Bridges, M. J., Chang, J., Rangan, R., Ottoni, G., Blome, J. A., Reis, G. A., Vachharajani, M., and August, D. I. RIFLE: An architectural framework for user-centric information-flow security. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'37)* (2004).

[35] Wang, S., Wang, P., and Wu, D. Reassembleable disassembling. In *Proceedings of the 24th USENIX Security Symposium* (2015), USENIX Association.

[36] Wang, X., Jhi, Y.-C., Zhu, S., and Liu, P. STILL: Exploit code detection via static taint and initialization analyses. In *Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC'08)* (2008).

[37] Whelan, R., Leek, T., and Kaeli, D. Architecture-independent dynamic information flow tracking. In *Proceedings of the 22nd international conference on Compiler Construction (CC'13)* (2013).

[38] Wondracek, G., Comparetti, P. M., Kruegel, C., and Kirda, E. Automatic network protocol analysis. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)* (2008).

[39] Xu, M., Malyugin, V., Sheldon, J., Venkitachalam, G., and Weissman, B. ReTrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the 2007 Workshop on Modeling, Benchmarking and Simulation* (2007).

[40] Xu, W., Bhatkar, S., and Sekar, R. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th Conference on USENIX Security Symposium (USENIX'06)* (2006).

[41] Yadegari, B., and Debray, S. Bit-level taint analysis. In *Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation* (2014).

[42] Yadegari, B., Johannesmeyer, B., Whitely, B., and Debray, S. A generic approach to automatic deobfuscation of executable code. In *Proceedings of the 36th IEEE Symposium on Security and Privacy* (2015).

[43] Yin, H., amd M. Egele, D. S., Kruegel, C., and Kirda, E. Panorama: Capturing system-wide information flow for malware detection and analysis. In *ACM Conference on Computer and Communications Security (CCS'07)* (2007).

[44] Yin, H., and Song, D. TEMU: Binary code analysis via whole-system layered annotative execution. Tech. Rep. UCB/EECS-2010-3, EECS Department, University of California, Berkeley, Jan 2010.

[45] Yip, A., Wang, X., Zeldovich, N., and Kaashoek, M. F. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 291–304.

[46] Zhao, Q., Cutcutache, I., and Wong, W.-F. PiPA: Pipelined profiling and analysis on multi-core systems. In *Proceedings of the 2008 International Symposium on Code Generation and Optimization (CGO'08)* (2009).

[47] Zhao, Q., Sim, J. E., Rudolph, L., and Wong, W.-F. DEP: Detailed execution profile. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT'06)* (2006).

[48] Zhao, R., Gu, D., Li, J., and Zhang, Y. Automatic detection and analysis of encrypted messages in malware. In *Proceedings of the 9th China International Conference on Information Security and Cryptology (INSCRYPT'13)* (2013).

[49] Zhu, D. Y., Jung, J., Song, D., Kohno, T., and Wetherall, D. TaintEraser: Protecting sensitive data leaks using application-level taint tracking. *ACM SIGOPS Operating Systems Review 45* (January 2011), 142–154.

Type Casting Verification: Stopping an Emerging Attack Vector

Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee

School of Computer Science
Georgia Institute of Technology

Abstract

Many applications such as the Chrome and Firefox browsers are largely implemented in C++ for its performance and modularity. Type casting, which converts one type of an object to another, plays an essential role in enabling polymorphism in C++ because it allows a program to utilize certain general or specific implementations in the class hierarchies. However, if not correctly used, it may return unsafe and incorrectly casted values, leading to so-called *bad-casting* or *type-confusion* vulnerabilities. Since a bad-casted pointer violates a programmer's intended pointer semantics and enables an attacker to corrupt memory, bad-casting has critical security implications similar to those of other memory corruption vulnerabilities. Despite the increasing number of bad-casting vulnerabilities, the bad-casting detection problem has not been addressed by the security community.

In this paper, we present CAVER, a runtime bad-casting detection tool. It performs program instrumentation at compile time and uses a new runtime type tracing mechanism—the type hierarchy table—to overcome the limitation of existing approaches and efficiently verify type casting dynamically. In particular, CAVER can be easily and automatically adopted to target applications, achieves broader detection coverage, and incurs reasonable runtime overhead. We have applied CAVER to large-scale software including Chrome and Firefox browsers, and discovered 11 previously unknown security vulnerabilities: nine in GNU `libstdc++` and two in Firefox, all of which have been confirmed and subsequently fixed by vendors. Our evaluation showed that CAVER imposes up to 7.6% and 64.6% overhead for performance-intensive benchmarks on the Chromium and Firefox browsers, respectively.

1 Introduction

The programming paradigm popularly known as object-oriented programming (OOP) is widely used for developing large and complex applications because it encapsulates the implementation details of data structures and algorithms into objects; this in turn facilitates cleaner software design, better code reuse, and easier software maintenance. Although there are many programming languages that support OOP, C++ has been the most popular, in particular when runtime performance is a key objective. For example, all major web browsers—Internet Explorer, Chrome, Firefox, and Safari are implemented in C++.

An important OOP feature is type casting that converts one object type to another. Type conversions play an important role in polymorphism. It allows a program to treat objects of one type as another so that the code can utilize certain general or specific features within the class hierarchy. Unlike other OOP languages—such as `Java`—that always verify the safety of a type conversion using runtime type information (RTTI), C++ offers two kinds of type conversions: `static_cast`, which verifies the correctness of conversion at *compile time*, and `dynamic_cast`, which verifies type safety at *runtime* using RTTI. `static_cast` is much more efficient because runtime type checking by `dynamic_cast` is an expensive operation (e.g., 90 times slower than `static_cast` on average). For this reason, many performance critical applications like web browsers, Chrome and Firefox in particular, prohibit `dynamic_cast` in their code and libraries, and strictly use `static_cast`.

However, the performance benefit of `static_cast` comes with a security risk because information at compile time is by no means sufficient to fully verify the safety of type conversions. In particular, upcasting (casting a derived class to its parent class) is always safe, but downcasting (casting a parent class to one of its derived classes) may not be safe because the derived class may not be a subobject of a truly allocated object in downcasting. Unsafe downcasting is better known as *bad-casting* or *type-confusion*.

Bad-casting has critical security implications. First, bad-casting is *undefined behavior* as specified in the C++ standard (5.2.9/11 [26]). Thus, compilers cannot guarantee the correctness of a program execution after bad-casting occurs (more detailed security implication analysis on undefined behavior is provided in §2). In addition to undefined behavior, bad-casting is similar to memory corruption vulnerabilities like stack/heap overflows and use-after-free. A bad-casted pointer violates a programmer's intended pointer semantics, and allows an attacker to corrupt memory beyond the true boundary of an object. For example, a bad-casting vulnerability in Chrome (CVE-2013-0912) was used to win the Pwn2Own 2013 competition by leaking and corrupting a security sensitive memory region [31]. More alarmingly, bad-casting is not only security-critical but is also common in applications. For example, 91 bad-casting vulnerabilities have been reported over the last four years in Chrome. Moreover, over 90% of these bad-casting bugs were rated as *security-high*, meaning that the bug can be directly exploited or indirectly used to mount arbitrary code execution attacks.

To avoid bad-casting issues, several C++ projects employ custom RTTI, which embeds code to manually keep type information at runtime and verify the type conversion safety of `static_cast`. However, only a few C++ programs are designed with custom RTTI, and supporting custom RTTI in existing programs requires heavy manual code modifications.

Another approach, as recently implemented by Google in the Undefined Behavior Sanitizer (UBSAN) [42], optimizes the performance of `dynamic_cast` and replaces all `static_cast` with `dynamic_cast`. However, this approach is limited because `dynamic_cast` only supports polymorphic classes, whereas `static_cast` is used for both polymorphic and non-polymorphic classes. Thus, this simple replacement approach changes the program semantics and results in runtime crashes when `dynamic_cast` is applied to non-polymorphic classes. It is difficult to identify whether a `static_cast` operation will be used for polymorphic or non-polymorphic classes without runtime information. For this reason, tools following this direction have to rely on manual *blacklists* (i.e., opt-out and do not check all non-polymorphic classes) to avoid runtime crashes. For example, UBSAN has to blacklist 250 classes, ten functions, and eight whole source files used for the Chromium browser [9], which is manually created by repeated trial-and-error processes. Considering the amount of code in popular C++ projects, creating such a blacklist would require massive manual engineering efforts.

In this paper, we present CAVER, a runtime bad-casting detection tool that can be seamlessly integrated with large-scale applications such as commodity browsers. It takes a program's source code as input and automatically instruments the program to verify type castings at runtime. We designed a new metadata, the Type Hierarchy Table (`THTable`) to efficiently keep track of rich type information. Unlike RTTI, `THTable` uses a disjoint metadata scheme (i.e., the reference to an object's `THTable` is stored outside the object). This allows CAVER to overcome all limitations of previous bad-casting detection techniques: it not only supports both polymorphic classes and non-polymorphic classes, but also preserves the C++ ABI and works seamlessly with legacy code. More specifically, CAVER achieves three goals:

- **Easy-to-deploy.** CAVER can be easily adopted to existing C++ programs without any manual effort. Unlike current state-of-the-art tools like UBSAN, it does not rely on manual blacklists, which are required to avoid program corruption. To demonstrate, we have integrated CAVER into two popular web browsers, Chromium and Firefox, by only modifying its build configurations.
- **Coverage.** CAVER can protect all type castings of both polymorphic and non-polymorphic classes. Compared to UBSAN, CAVER covers 241% and 199% more classes and their castings, respectively.
- **Performance.** CAVER also employs optimization techniques to further reduce runtime overheads (e.g., type-based casting analysis). Our evaluation shows that CAVER imposes up to 7.6% and 64.6% overheads for performance-intensive benchmarks on the Chromium and Firefox browsers, respectively. On the contrary, UBSAN is 13.8% slower than CAVER on the Chromium browser, and it cannot run the Firefox browser due to a runtime crash.

To summarize, we make three major contribution as follows:

- **Security analysis of bad-casting.** We analyzed the bad-casting problem and its security implications in detail, thus providing security researchers and practitioners a better understanding of this emerging attack vector.
- **Bad-casting detection tool.** We designed and implemented CAVER, a general, automated, and easy-to-deploy tool that can be applied to any C++ application to detect (and mitigate) bad-casting vulnerabilities. We have shared CAVER with the Firefox team [1] and made our source code publicly available.
- **New vulnerabilities.** While evaluating CAVER, we discovered *eleven* previously unknown bad-casting vulnerabilities in two mature and widely-used open source projects, GNU `libstdc++` and Firefox. All vulnerabilities have been reported and fixed in these projects' latest releases. We expect that integration with unit tests and fuzzing infrastructure will allow CAVER to discover more bad-casting vulnerabilities in the future.

This paper is organized as follows. §2 explains bad-casting issues and their security implications. §3 illustrates high-level ideas and usages of CAVER, §4 describes the design of CAVER. §5 describes the implementation details of CAVER, §6 evaluates various aspects of

[1] The Firefox team at Mozilla asked us to share CAVER for regression testing on bad-casting vulnerabilities.

CaVer. §7 further discusses applications and limitations of CaVer, §8 describes related work. Finally, §9 concludes the paper.

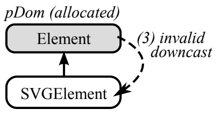
2 C++ Bad-casting Demystified

Type castings in C++. Type casting in C++ allows an object of one type to be converted to another so that the program can use different features of the class hierarchy. C++ provides four explicit casting operations: `static`, `dynamic`, `const`, and `reinterpret`. In this paper, we focus on the first two types — `static_cast` and `dynamic_cast` (5.2.9 and 5.2.7 in ISO/IEC N3690 [26]) — because they can perform downcasting and result in bad-casting. `static_cast` and `dynamic_cast` have a variety of different usages and subtle issues, but for the purpose of this paper, the following two distinctive properties are the most important: (1) time of verification: as the name of each casting operation implies, the correctness of a type conversion is checked (statically) at compile time for `static_cast`, and (dynamically) at runtime for `dynamic_cast`; (2) runtime support (RTTI): to verify type checking at runtime, `dynamic_cast` requires runtime support, called RTTI, that provides type information of the polymorphic objects.

Example 1 illustrates typical usage of both casting operations and their correctness and safety: (1) casting from a derived class (`pCanvas` of `SVGElement`) to a parent class (`pEle` of `Element`) is valid *upcasting*; (2) casting from the parent class (`pEle` of `Element`) to the original allocated class (`pCanvasAgain` of `SVGElement`) is valid *downcasting*; (3) on the other hand, the casting from an object allocated as a base class (`pDom` of `Element`) to a derived class (`p` of `SVGElement`) is invalid *downcasting* (i.e., a bad-casting); (4) memory access via the invalid pointer (`p->m_className`) can cause memory corruption, and more critically, compilers cannot guarantee any correctness of program execution after this incorrect conversion, resulting in *undefined behavior*; and (5) by using `dynamic_cast`, programmers can check the correctness of type casting at runtime, that is, since an object allocated as a base class (`pDom` of `Element`) cannot be converted to its derived class (`SVGElement`), `dynamic_cast` will return a `NULL` pointer and the error-checking code (line 18) can catch this bug, thus avoiding memory corruption.

Type castings in practice. Although `dynamic_cast` can guarantee the correctness of type casting, it is an expensive operation because parsing RTTI involves recursive data structure traversal and linear string comparison. From our preliminary evaluation, `dynamic_cast` is, on average, 90 times slower than `static_cast` on average. For large applications such as the Chrome browser, such performance overhead is not acceptable: simply launching Chrome incurs over 150,000 casts. Therefore, despite its security benefit, the use of `dynamic_cast` is strictly forbidden in Chrome development.

```
1  class SVGElement: public Element { ... };
2
3  Element *pDom = new Element();
4  SVGElement *pCanvas = new SVGElement();
5
6  // (1) valid upcast from pCanvas to pEle
7  Element *pEle = static_cast<Element*>(pCanvas);
8  // (2) valid downcast from pEle to pCanvasAgain (== pCanvas)
9  SVGElement *pCanvasAgain = static_cast<SVGElement*>(pEle);
10
11 // (3) invalid downcast (-> undefined behavior)
12 SVGElement *p = static_cast<SVGElement*>(pDom);
13 // (4) leads to memory corruption
14 p->m_className = "my-canvas";
15
16 // (5) invalid downcast with dynamic_cast, but no corruption
17 SVGElement *p = dynamic_cast<SVGElement*>(pDom);
18 if (p) {
19     p->m_className = "my-canvas";
20  }
```



Example 1: Code example using `static_cast` to convert types of object pointers (e.g., `Element` ↔ `SVGElement` classes). (1) is valid *upcast* and (2) is valid *downcast*. (3) is an invalid *downcast*. (4) Memory access via the invalid pointer result in memory corruption; more critically, compilers cannot guarantee the correctness of program execution after this incorrect conversion, resulting in *undefined behavior*. (5) Using `dynamic_cast`, on the other hand, the program can check the correctness of *downcast* by checking the returned pointer.

A typical workaround is to implement custom RTTI support. For example, most classes in WebKit-based browsers have an `isType()` method (e.g., `isSVGElement()`), which indicates the true allocated type of an object. Having this support, programmers can decouple a `dynamic_cast` into an explicit type check, followed by `static_cast`. For example, to prevent the bad-casting (line 12) in Example 1, the program could invoke the `isSVGElement()` method to check the validity of casting. However, this sort of type tracking and verification has to be manually implemented, and thus supporting custom RTTI in existing complex programs is a challenging problem. Moreover, due to the error-prone nature of manual modifications (e.g., incorrectly marking the object identity flag, forgetting to check the identity using `isType()` function, etc.), bad-casting bugs still occur despite custom RTTI [41].

Security implications of bad-casting. The C++ standard (5.2.9/11 [26]) clearly specifies that the behavior of an application becomes *undefined* after an incorrect `static_cast`. Because undefined behavior is an enigmatic issue, understanding the security implications and exploitability of bad-casting requires deep knowledge of common compiler implementations.

Generally, bad-casting vulnerabilities can be exploited via several means. An incorrectly casted pointer will either have wider code-wise visibility (e.g., allowing out-of-bound memory accesses), or become incorrectly adjusted (e.g., corrupting memory semantics because of misalignment). For example, when bad-casting occurs in proximity to a virtual function table pointer (vptr), an attacker can directly control input to the member variable (e.g., by employing abusive memory allocation techniques such as heap-spray techniques [16, 38]), overwrite the vptr and hijack the control flow. Similarly, an attacker can also exploit bad-casting vulnerabilities to launch non-control-data attacks [7].

The exploitability of a bad-casting bug depends on whether it allows attackers to perform out-of-bound memory access or manipulate memory semantics. This in turn relies on the details of object data layout as specified by the C++ application binary interface (ABI). Because the C++ ABI varies depending on the platform (e.g., Itanium C++ ABI [12] for Linux-based platforms and Microsoft C++ ABI [11] for Windows platforms), security implications for the same bad-casting bug can be different. For example, bad-casting may not crash, corrupt, or alter the behavior of an application built against the Itanium C++ ABI because the base pointer of both the base class and derived class always point to the same location of the object under this ABI. However, the same bad-casting bug can have severe security implications for other ABI implementations that locate a base pointer of a derived class differently from that of a base class, such as HP and legacy g++ C++ ABI [13]. In short, given the number of different compilers and the various architectures supported today, we want to highlight that bad-casting should be considered as a serious security issue. This argument is also validated from recent correspondence with the Firefox security team: after we reported two new bad-casting vulnerabilities in Firefox [4], they also pointed out the C++ ABI compatibility issue and rated the vulnerability as security-high.

Running example: CVE-2013-0912. Our illustrative Example 1 is extracted from a real-world bad-casting vulnerability—CVE-2013-0912, which was used to exploit the Chrome web browser in the Pwn2Own 2013 competition. However, the complete vulnerability is more complicated as it involves a multitude of castings (between siblings and parents).

In HTML5, an SVG image can be embedded directly into an HTML page using the <svg> tag. This tag is implemented using the SVGElement class, which inherits from the Element class. At the same time, if a web page happens to contain unknown tags (any tags other than standard), an object of the HTMLUnknownElement class will be created to represent this unknown tag. Since both tags are valid HTML elements, objects of these types can be safely casted to the Element class. Bad-casting occurs when the browser needs to render an SVG image. Given an Element object, it tries to downcast the object to SVGElement so the caller function can invoke member functions of the SVGElement class. Unfortunately, since not all Element objects are initially allocated as SVGElement objects, this

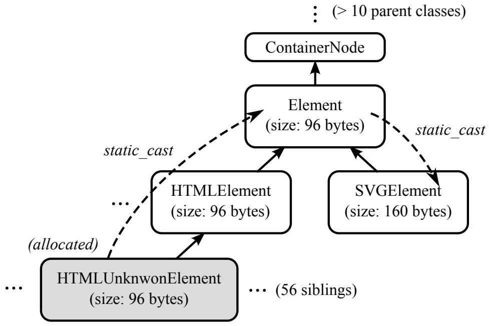


Figure 1: Inheritance hierarchy of classes involved in the CVE-2013-0912 vulnerability. MWR Labs exploited this vulnerability to hijack the Chrome browser in the Pwn2Own 2013 competition [31]. The object is allocated as HTMLUnknownElement and eventually converted (static_cast) to SVGElement. After this incorrect type casting, accessing member variables via this object pointer will cause memory corruption.

static_cast is not always valid. In the exploit demonstrated in the Pwn2Own 2013 competition [31], attackers used an object allocated as HTMLUnknownElement. As the size of an SVGElement object (160 bytes) is much larger than an HTMLUnknownElement object (96 bytes), this incorrectly casted object pointer allowed the attackers to access memory beyond the real boundary of the allocated HTMLUnknownElement object. They then used this capability to corrupt the vtable pointer of the object adjacent to the HTMLUnknownElement object, ultimately leading to a control-flow hijack of the Chrome browser. This example also demonstrates why identifying bad-casting vulnerabilities is not trivial for real-world applications. As shown in Figure 1, the HTMLUnknownElement class has more than 56 siblings and the Element class has more than 10 parent classes in WebKit. Furthermore, allocation and casting locations are far apart within the source code. Such complicated class hierarchies and disconnections between allocation and casting sites make it difficult for developers and static analysis techniques to reason about the true allocation types (i.e., alias analysis).

3 CAVER Overview

In this paper, we focus on the correctness and effectiveness of CAVER against bad-casting bugs, and our main application scenario is as a back-end testing tool for detecting bad-casting bugs. CAVER's workflow (Figure 2) is as simple as compiling a program with one extra compile and link flag (i.e., -fcaver for both). The produced binary becomes capable of verifying the correctness of every type conversion at runtime. When CAVER detects an incorrect type cast, it provides detailed information of the bad-cast: the source class, the destination class, the truly allocated class, and call stacks at the time the bad-cast is captured. Figure 3 shows a snippet of the actual report of CVE-2013-0912. Our bug report experience showed that the report generated by CAVER helped upstream main-

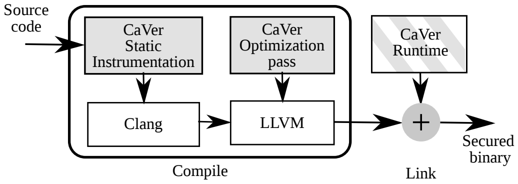


Figure 2: Overview of CAVER's design and workflow. Given the source code of a program, CAVER instruments possible castings at compile time, and injects CAVER's runtime to verify castings when they are performed.

```
== CaVer : (Stopped) A bad-casting detected
@SVGViewSpec.cpp:87:12
  Casting an object of 'blink::HTMLUnknownElement'
    from 'blink::Element
    to   'blink::SVGElement'
         Pointer         0x60c000008280
         Alloc base      0x60c000008280
         Offset          0x000000000000
         THTable         0x7f7963aa20d0

#1 0x7f795d76f1a4 in viewTarget SVGViewSpec.cpp:87
#2 0x7f795d939d1c in viewTargetAttribute V8SVGViewSpec.cpp:56
...
```

Figure 3: A report that CAVER generated on CVE-2013-0912.

tainers easily understand, confirm, and fix eleven newly discovered vulnerabilities without further examination.

4 Design

In this section, we introduce the design of CAVER. We first describe how the THTable is designed to generally represent the type information for both polymorphic and non-polymorphic classes (§4.1), and then explain how CAVER associates the THTable with runtime objects (§4.2). Next, we describe how CAVER verifies the correctness of type castings (§4.3). At the end of this section, we present optimization techniques used to reduce the runtime overhead of CAVER (§4.4).

4.1 Type Hierarchy Table

To keep track of the type information required for validating type casting, CAVER incorporates a new metadata structure, called the *Type Hierarchy Table* (THTable). Given a pointer to an object allocated as type T, the THTable contains the set of all possible types to which T can be casted. In C++, these possible types are a product of two kinds of class relationships: *is-a* and *has-a*. The *is-a* relationship between two objects is implemented as class inheritance, the *has-a* relationship is implemented as class composition (i.e., having a member variable in a class). Thus, for each class in a C++ program, CAVER creates a corresponding THTable that includes information about both relationships.

To represent class inheritance, the THTable employs two unique design decisions. First, information on inherited classes (i.e., base classes) is unrolled and serialized. This allows CAVER to efficiently scan through a set of base classes at runtime while standard RTTI requires recursive traversal. Second, unlike RTTI, which stores a mangled class name, the THTable stores the hash value of a class name. This allows CAVER to avoid expensive string equality comparisons. Note, since all class names are available to CAVER at compile time, all possible hash collisions can be detected and resolved to avoid false negatives during runtime. Moreover, because casting is only allowed between classes within the same inheritance chain, we only need to guarantee the uniqueness of hash values within a set of those classes, as opposed to guaranteeing global uniqueness.

The THTable also includes information of whether a base class is a phantom class, which cannot be represented based on RTTI and causes many false alarms in RTTI-based type verification solutions [9]. We say a class P is a phantom class of a class Q if two conditions are met: (1) Q is directly or indirectly derived from P; and (2) compared to P, Q does not have additional member variables or different virtual functions. In other words, they have the same data layout. Strictly speaking, allocating an object as P and downcasting it to Q is considered bad-casting as Q is not a base class of P. However, such bad-castings are harmless from a security standpoint, as the pointer semantic after downcasting is the same. More importantly, phantom classes are often used in practice to implement object relationships with empty inheritances. For these reasons, CAVER deliberately allows bad-castings caused by phantom classes. This is done by reserving a one bit space in the THTable for each base class, and marking if the base class is a phantom class. We will describe more details on how the phantom class information is actually leveraged in §4.3.

In addition, the THTable contains information on composited class(es) to generally represent the type information for both polymorphic and non-polymorphic classes and overcome the limitation of RTTI-based type verification solutions. RTTI-based solutions locate a RTTI reference via the virtual function table (VTable). However, since only polymorphic classes have VTable, these solutions can cause runtime crashes when they try to locate the VTable for non-polymorphic classes. Unlike RTTI, CAVER binds THTable references to the allocated object with external metadata (refer §4.2 for details). Therefore, CAVER not only supports non-polymorphic objects, but it also does not break the C++ ABI. However, composited class(es) now share the same THTable with their container class. Since a composited class can also have its own inheritances and compositions, we do not unroll information about composited class(es); instead, CAVER provides a reference to the composited class's THTable. The THTable also stores the layout information (offset and size) of each composited class to determine whether the given pointer points to a certain composited class.

Other than inheritance and composition information as described above, the THTable contains basic information on the corresponding type itself: a type size to represent object ranges; and a type name to generate user-friendly bad-casting reports.

4.2 Object Type Binding

To verify the correctness of type casting, CAVER needs to know the actual allocated type of the object to be casted. In CAVER, we encoded this type information in the THTable. In this subsection, we describe how CAVER binds the THTable to each allocated object. To overcome the limitations of RTTI-based solutions, CAVER uses a disjoint metadata scheme (i.e., the reference to an object's THTable is stored outside the object). With this unique metadata management scheme, CAVER not only supports both polymorphic classes and non-polymorphic classes, but also preserves the C++ ABI and works seamlessly with legacy code. Overall, type binding is done in two steps. First, CAVER instruments each allocation site of an application to pass the allocation metadata to its runtime library. Second, CAVER's runtime library maintains the allocation metadata and supports efficient lookup operations.

Instrumentation. The goal of the instrumentation is to pass all information of an allocated object to the runtime library. To bind a THTable to an object, the runtime library needs two pieces of information: a reference to the THTable and the base address of the allocated object.

In C++, objects can be allocated in three ways: in heap, on stack, or as global objects. In all three cases, the type information of the allocated object can be determined statically at compile time. This is possible because C++ requires programmers to specify the object's type at its allocation site, so the corresponding constructor can be invoked to initialize memory. For global and stack objects, types are specified before variable names; and for heap objects, types are specified after the new operator. Therefore, CAVER can obtain type information by statically inspecting the allocation site at compile time. Specifically, CAVER generates the THTable (or reuses the corresponding THTable if already generated) and passes the reference of the THTable to the runtime library. An example on how CAVER instruments a program is shown in Example 2.

For heap objects, CAVER inserts one extra function invocation (trace_heap() in Example 2) to the runtime library after each new operator, and passes the information of the object allocated by new; a reference to the THTable and the base address of an object. A special case for the new operator is an array allocation, where a set of objects of the same type are allocated. To handle this case, we add an extra parameter to inform the runtime library on how many objects are allocated together at the base address.

Unlike heap objects, stack objects are implicitly allocated and freed. To soundly trace them, CAVER inserts two function calls for each stack object at the function prologue and epilogue (trace_stack_begin() and trace_stack_end() in Example 2), and passes the same information of the object as is done for heap objects. A particular challenge is that, besides function returns, a stack unwinding can also happen due to exceptions and setjmp/longjmp. To handle these cases, CAVER leverages existing compiler functionality (e.g., EHScopeStack::Cleanup in clang) to guarantee that the runtime library is always invoked once the execution context leaves the given function scope.

```
  // Heap objects (dynamically allocated)
  void func_heap_ex() {
    C *p_heap_var = new C;
    C *p_heap_array = new C[num_heap_array];
+   trace_heap(&THTable(C), p_heap_var, 1);
+   trace_heap(&THTable(C), p_heap_array, num_heap_array);
    ...
  }

  // Stack objects
  void func_stack_ex() {
    C stack_var;
+   trace_stack_begin(&THTable(C), &stack_var, 1);
    ...
+   trace_stack_end(&stack_var);
  }

  // Global objects
  C global_var;

  // @.ctors: (invoked at the program's initialization)
  //   trace_global_helper_1() and trace_global_helper_2()
+ void trace_global_helper_1() {
+   trace_global(&THTable(C), &global_var, 1);
+ }

   // Verifying the correctness of a static casting
  void func_verify_ex() {
    B *afterAddr = static_cast<A>(beforeAddr);
+   verify_cast(beforeAddr, afterAddr, type_hash(A));
  }
```

Example 2: An example of how CAVER instruments a program. Lines marked with + represent code introduced by CAVER, and &THTable(T) denotes the reference to the THTable of class T. In this example, we assume that the THTable of each allocated class has already been generated by CAVER.

To pass information of global objects to the runtime library, we leverage existing program initialization procedures. In ELF file format files [46], there is a special section called .ctors, which holds constructors that must be invoked during an early initialization of a program. Thus, for each global object, CAVER creates a helper function (trace_global_helper_1() in Example 2) that invokes the runtime library with static metadata (the reference to the THTable) and dynamic metadata (the base address and the number of array elements). Then, CAVER adds the pointer to this helper function to the .ctors section so that the metadata can be conveyed to the runtime library[2].

[2] Although the design detail involving .ctors section is platform dependent, the idea of registering the helper function into the initialization

Runtime library. The runtime library of CAVER maintains all the metadata (`THTable` and base address of an object) passed from tracing functions during the course of an application execution. Overall, we consider two primary requirements when organizing the metadata. First, the data structure must support range queries (i.e., given a pointer pointing to an address within an object ([base, base+size)) CAVER should be able to find the corresponding `THTable` of the object). This is necessary because the object pointer does not always point to the allocation base. For example, the pointer to be casted can point to a composited object. In case of multi-inheritance, the pointer can also point to one of the parent classes. Second, the data structure must support efficient store and retrieve operations. CAVER needs to store the metadata for every allocation and retrieve the metadata for each casting verification. As the number of object allocations and type conversions can be huge (see §6), these operations can easily become the performance bottleneck.

We tackle these challenges using a hybrid solution (see Appendix 2 for the algorithm on runtime library functions). We use red-black trees to trace global and stack objects and an alignment-based direct mapping scheme to trace heap objects[3].

We chose red-black trees for stack and global objects for two reasons. First, tree-like data structures are well known for supporting efficient range queries. Unlike hash-table-based data structures, tree-based data structures arrange nodes according to the order of their keys, whose values can be numerical ranges. Since nodes are already sorted, a balanced tree structure can guarantee $O(logN)$ complexity for range queries while hash-table-based data structure requires $O(N)$ complexity. Second, we specifically chose red-black trees because there are significantly more search operations than update operations (i.e., more type conversion operations than allocations, see §6), thus red-black trees can excel in performance due to self-balancing.

In CAVER, each node of a red-black tree holds the following metadata: the base address and the allocation size as the key of the node, and the `THTable` reference as the value of the node.

For global object allocations, metadata is inserted into the global red-black tree when the object is allocated at runtime, with the key as the base address and the allocation size[4], and the value as the address of the `THTable`. We maintain a per-process global red-black tree without locking mechanisms because there are no data races on the global red-black tree in CAVER. All updates on the global red-black tree occur during early process start-up (i.e., before executing any user-written code) and update orders are well serialized as listed in the .ctors section.

For stack object allocations, metadata is inserted to the stack red-black tree similar to the global object case. Unlike a global object, we maintain a *per-thread* red-black tree for stack objects to avoid data races in multi-threaded applications. Because a stack region (and all operations onto this region) are exclusive to the corresponding thread's execution context, this per-thread data structure is sufficient to avoid data races without locks.

For heap objects, we found that red-black trees are not a good design choice, especially for multi-threaded programs. Different threads in the target programs can update the tree simultaneously, and using locks to avoid data races resulted in high performance overhead, as data contention occured too frequently. Per-thread red-black trees used for stack objects are not appropriate either, because heap objects can be shared by multiple threads. Therefore, we chose to use a custom memory allocator that can support alignment-based direct mapping schemes [3, 22]. In this scheme, the metadata can be maintained for a particular object, and can be retrieved with $O(1)$ complexity on the pointer pointing to anywhere within the object's range.

4.3 Casting Safety Verification

This subsection describes how CAVER uses traced information to verify the safety of type casting. We first describe how the instrumentation is done at compile time, and then describe how the runtime library eventually verifies castings during runtime.

Instrumentation. CAVER instruments `static_cast` to invoke a runtime library function, `verify_cast()`, to verify the casting. Here, CAVER analyzes a type hierarchy involving source and destination types in `static_cast` and only instruments for downcast cases. When invoking `verify_cast()`, CAVER passes the following three pieces of information: `beforeAddr`, the pointer address before the casting; `afterAddr`, the pointer address after the casting; and `TargetTypeHash`, the hash value of the destination class to be casted to (denoted as `type_hash(A)` in Example 2).

Runtime library. The casting verification (Appendix 1) is done in two steps: (1) locating the corresponding `THTable` associated with the object pointed to by `beforeAddr`; and (2) verifying the casting operation by checking whether `TargetTypeHash` is a valid type where `afterAddr` points.

To locate the corresponding `THTable`, we first check the data storage membership because we do not know how the object `beforeAddr` points to is allocated. Checks

function list can be generalized for other platforms as others also support .ctors-like features

[3] The alignment-based direct mapping scheme can be applied for global and stack objects as well, but this is not implemented in the current version. More details can be found in §7.

[4] The allocation size is computed by multiplying the type size represented in `THTable` and the number of array elements passed during runtime.

are ordered by their expense, and the order is critical for good performance. First, a stack object membership is checked by determinig whether the beforeAddr is in the range between the stack top and bottom; then, a heap object membership is checked by whether the beforeAddr is in the range of pre-mapped address spaces reserved for the custom allocator; finally a global object membership is checked with a bit vector array for each loaded binary module. After identifying the data storage membership, CAVER retrieves the metadata containing the allocation base and the reference to the THTable. For stack and global objects, the corresponding red-black tree is searched. For heap objects, the metadata is retrieved from the custom heap.

Next, CAVER verifies the casting operation. Because the THTable includes all possible types that the given object can be casted to (i.e., all types from both inheritances and compositions), CAVER exhaustively matches whether TargetTypeHash is a valid type where afterAddr points. To be more precise, the afterAddr value is adjusted for each matching type. Moreover, to avoid false positives due to a phantom class, CAVER tries to match all phantom classes of the class to be casted to.

4.4 Optimization

Since performance overhead is an important factor for adoption, CAVER applies several optimization techniques. These techniques are applied in two stages, as shown in Figure 2. First, offline optimizations are applied to remove redundant instrumentations. After that, additional runtime optimizations are applied to further reduce the performance overhead.

Safe-allocations. Clearly, not all allocated objects will be involved in type casting. This implicates that CAVER does not need to trace type information for objects that would never be casted. In general, soundly and accurately determining whether objects allocated at a given allocation site will be casted is a challenging problem because it requires sophisticated static points-to analysis. Instead, CAVER takes a simple, yet effective, optimization approach inspired from C type safety checks in CCured [33]. The key idea is that the following two properties always hold for downcasting operations: (1) bad-casting may happen only if an object is allocated as a child of the source type or the source type itself; and (2) bad-casting never happens if an object is allocated as the destination type itself or a child of the destination type. This is because static_cast guarantees that the corresponding object must be a derived type of the source type. Since CAVER can observe all allocation sites and downcasting operations during compilation, it can recursively apply the above properties to identify *safe-allocation* sites, i.e., the allocated objects will never cause bad-casting.

Caching verification results. Because casting verification involves loops (over the number of compositions and the number of bases) and recursive checks (in a composition case), it can be a performance bottleneck. A key observation here is that the verification result is always the same for the same allocation type and the same target type (i.e., when the type of object pointed by afterAddr and TargetTypeHash are the same). Thus, in order to alleviate this potential bottleneck, we maintain a cache for verification results, which is inspired by UBSAN [42]. First, a verification result is represented as a concatenation of the address of a corresponding THTable, the offset of the afterAddr within the object, and the hash value of target type to be casted into (i.e., &THTable || offset || TargetTypeHash). Next, this concatenated value is checked for existence in the cache before verify_cast() actually performs verification. If it does, verify_cast() can conclude that this casting is correct. Otherwise, verify_cast() performs actual verification using the THTable, and updates the cache only if the casting is verified to be correct.

5 Implementation

We implemented CAVER based on the LLVM Compiler project [43] (revision 212782, version 3.5.0). The static instrumentation module is implemented in Clang's CodeGen module and LLVM's Instrumentation module. The runtime library is implemented using the compiler-rt module based on LLVM's Sanitizer code base. In total, CAVER is implemented in 3,540 lines of C++ code (excluding empty lines and comments).

CAVER is currently implemented for the Linux x86 platform, and there are a few platform-dependent mechanisms. For example, the type and tracing functions for global objects are placed in the .ctors section of ELF. As these platform-dependent features can also be found in other platforms, we believe CAVER can be ported to other platforms as well. CAVER interposes threading functions to maintain thread contexts and hold a per-thread red-black tree for stack objects. CAVER also maintains the top and bottom addresses of stack segments to efficiently check pointer membership on the stack. We also modified the front-end drivers of Clang so that users of CAVER can easily build and secure their target applications with one extra compilation flag and linker flag, respectively.

6 Evaluation

We evaluated CAVER with two popular web browsers, Chromium [40] (revision 295873) and Firefox [44] (revision 213433), and two benchmarks from SPEC CPU2006 [39][5]. Our evaluation aims to answer the following questions:

[5] Although CAVER was able to correctly run all C++ benchmarks in SPEC CPU2006, only 483.xalancbmk and 450.soplex have downcast operations.

- How easy is it to deploy CAVER to applications? (§6.1)
- What are the new vulnerabilities CAVER found? (§6.2)
- How precise is CAVER's approach in detecting bad-casting vulnerabilities? (§6.3)
- How good is CAVER's protection coverage? (§6.4)
- What are the instrumentation overheads that CAVER imposes and how many type castings are verified by CAVER? (§6.5)
- What are the runtime performance overheads that CAVER imposes? (§6.6)

Comparison methods. We used UBSAN, the state-of-art tool for detecting bad-casting bugs, as our comparison target of CAVER. Also, We used CAVER-NAIVE, which disabled the two optimization techniques described in §4.4, to show their effectiveness on runtime performance optimization.

Experimental setup. All experiments were run on Ubuntu 13.10 (Linux Kernel 3.11) with a quad-core 3.40 GHz CPU (Intel Xeon E3-1245), 16 GB RAM, and 1 TB SSD-based storage.

6.1 Deployments

As the main design goal for CAVER is automatic deployments, we describe our experience of applying CAVER to tested programs including SPEC CPU 2006 benchmarks, the Chromium browser, and the Firefox browser. CAVER was able to successfully build and run these programs without any program-specific understanding of the code base. In particular, we added one line to the build configuration file to build SPEC CPU 2006, 21 lines to the `.gyp` build configuration to build the Chromium browser, and 10 lines to the `.mozconfig` build configuration file to build the Firefox browser. Most of these build configuration changes were related to replacing `gcc` with `clang`.

On the contrary, UBSAN crashed while running xalancbmk in SPEC CPU 2006 and while running the Firefox browser due to checks on non-polymorphic classes. UBSAN also crashed the Chromium browser without blacklists, but was able to run once we applied the blacklists provided by the Chromium project [9]. In particular, to run Chromium, the blacklist has 32 different rules that account for 250 classes, ten functions, and eight whole source files. Moreover, this blacklist has to be maintained constantly as newly introduced code causes new crashes in UBSAN [10]. This is a practical obstacle for adopting UBSAN in other C++ projects—although UBSAN has been open sourced for some time, Chromium remains the only major project that uses UBSAN, because there is a dedicated team to maintain its blacklist.

6.2 Newly Discovered Bad-casting Vulnerabilities

To evaluate CAVER's capability of detecting bad-casting bugs, we ran CAVER-hardened Chromium and Firefox with their regression tests (mostly checking functional correctness). During this evaluation, CAVER found eleven previously unknown bad-casting vulnerabilities in GNU `libstdc++` while evaluating Chromium and Firefox. Table 1 summarizes these vulnerabilities including related class information: allocated type, source, and destination types in each bad-casting. In addition, we further analyzed their security impacts: potential compatibility problems due to the C++ ABI (see §2) or direct memory corruption, along with security ratings provided by Mozilla for Firefox.

CAVER found two vulnerabilities in the Firefox browser. The Firefox team at Mozilla confirmed and fixed these, and rated both as *security-high*, meaning that the vulnerability can be abused to trigger memory corruption issues. These two bugs were casting the pointer into a class which is not a base class of the originally allocated type. More alarmingly, there were type semantic mismatches after the bad-castings—subsequent code could dereference the incorrectly casted pointer. Thus the C++ ABI and Memory columns are checked for these two cases.

CAVER also found nine bugs in GNU `libstdc++` while running the Chromium browser. We reported these bugs to the upstream maintainers, and they have been confirmed and fixed. Most of these bugs were triggered when `libstdc++` converted the type of an object pointing to its composite objects (e.g., `Base_Ptr` in `libstdc++`) into a more derived class (`Rb_Tree_node` in `libstdc++`), but these derived classes were not base classes of what was originally allocated (e.g., `EncodedDescriptorDatabase` in Chromium). Since these are generic bugs, meaning that benign C++ applications will encounter these issues even if they correctly use `libstdc++` or related libraries, it is difficult to directly evaluate their security impacts without further evaluating the applications themselves.

These vulnerabilities were identified with legitimate functional test cases. Thus, we believe CAVER has great potential to find more vulnerabilities once it is utilized for more applications and test cases, as well as integrated with fuzzing infrastructures like ClusterFuzz [2] for Chromium.

6.3 Effectiveness of Bad-casting Detection

To evaluate the correctness of detecting bad-casting vulnerabilities, we tested five bad-casting exploits of Chromium on the CAVER-hardened Chromium binary (see Table 2). We backported five bad-casting vulnerabilities as unit tests while preserving important features that may affect CAVER's detection algorithm, such as class inheritances and their compositions, and allocation

Product	Bug ID	Vulnerable	Types	Security Implication		
		Function	Allocation / Source / Destination	ABI	Mem	Rating
Firefox	1074280 [4]	PaintLayer()	BasicThebesLayer / Layer / BasicContainerLayer	✓	✓	High (CVE-2014-1594)
Firefox	1089438 [5]	EvictContent()	PRCListStr / PRCList / nsSHistory	✓	✓	High
`libstdc++`	63345 [30]	_M_const_cast()	EncodedDescriptorDatabase / Base_Ptr / Rb_Tree_node	✓	-	†
`libstdc++`	63345 [30]	_M_end()	EnumValueOptions / Rb_tree_node_base / Link_type	✓	-	†
`libstdc++`	63345 [30]	_M_end() const	GeneratorContextImpl / Rb_tree_node_base / Link_type_const	✓	-	†
`libstdc++`	63345 [30]	_M_insert_unique()	WaitableEventKernel / Base_ptr / List_type	✓	-	†
`libstdc++`	63345 [30]	operator*()	BucketRanges / List_node_base / Node	✓	-	†
`libstdc++`	63345 [30]	begin()	FileOptions / Link_type / Rb_Tree_node	✓	-	†
`libstdc++`	63345 [30]	begin() const	std::map / Link_type / Rb_Tree_node	✓	-	†
`libstdc++`	63345 [30]	end()	MessageOptions / Link_type / Rb_Tree_node	✓	-	†
`libstdc++`	63345 [30]	end() const	Importer / Link_type / Rb_Tree_node	✓	-	†

Table 1: A list of vulnerabilities newly discovered by CaVer. All security vulnerabilities listed here are confirmed, and already fixed by the corresponding development teams. Columns under Types represent classes causing bad-castings: allocation, source and destination classes. Columns under Security Implication represents the security impacts of each vulnerability: whether the vulnerability has C++ ABI incompatibility issues (ABI); whether the vulnerability triggers memory corruption (Mem); and the actual security assessment ratings assigned by the vendor (Rating). †: The GNU `libstdc++` members did not provide security ratings.

size. This backporting was due to the limited support for the `LLVM/clang` compiler by older Chromium (other than CVE-2013-0912). Table 2 shows our testing results on these five known bad-casting vulnerabilities. CaVer successfully detected all vulnerabilities.

In addition to real vulnerabilities, we thoroughly evaluated CaVer with test cases that we designed based on all possible combinations of bad-casting vulnerabilities: (1) whether an object is polymorphic or non-polymorphic; and (2) the three object types: allocation, source, and destination.

$$|\{\text{Poly, non-Poly}\}|^{|\{\text{Alloc, From, To}\}|} = 8$$

Eight different unit tests were developed and evaluated as shown in Table 3. Since CaVer's design generally handles both polymorphic and non-polymorphic classes, CaVer successfully detected all cases. For comparison, UBSan failed six cases mainly due to its dependency on RTTI. More severely, among the failed cases, UBSan crashed for two cases when it tried to parse RTTI non-polymorphic class objects, showing it is difficult to use without manual blacklists. Considering Firefox contains greater than 60,000 downcasts, (see Table 4), creating such a blacklist for Firefox would require massive manual engineering efforts.

6.4 Protection Coverage

Table 4 summarizes our evaluation of CaVer's protection coverage during instrumentation, including the number of protected types/classes (the left half), and the number of protected type castings (the right half). In our evaluation with C++ applications in SPEC CPU 2006, Firefox, and Chromium, CaVer covers 241% more types than UBSan; and protects 199% more type castings.

6.5 Instrumentation Overheads

There are several sources that increase a program's binary size (see Table 5), including (1) the inserted functions for tracing objects' type and verifying type castings, (2) the `THTable` of each class, and (3) CaVer's runtime library. Although CaVer did not perform much instrumentation for most SPEC CPU 2006 applications, the file size increase still was noticeable. This increase was caused by the statically linked runtime library (245 KB). The CaVer-hardened Chromium requires 6× more storage compared to Firefox because the Chromium code bases contains more classes than Firefox. The additional `THTable` overhead is the dominant source of file size increases. (see Table 4). For comparison, UBSan increased the file size by 64% and 49% for Chromium and Firefox, respectively; which indicates that `THTable` is an efficient representation of type information compared to RTTI.

Name	# of tables		# of verified cast	
	RTTI	`THTable`	UBSan	CaVer
483.xalancbmk	881	3,402	1,378	1,967
450.soplex	39	227	0	2
Chromium	24,929	94,386	11,531	15,611
Firefox	9,907	30,303	11,596	71,930

Table 4: Comparisons of protection coverage between UBSan and CaVer. In the # of tables column, `VTable` shows the number of virtual function tables and `THTable` shows the number of type hierarchy tables, each of which is generated to build the program. # of verified cast shows the number `static_cast` instrumented in UBSan and CaVer, respectively. Overall, CaVer covers 241% and 199% more classes and their castings, respectively, compared to UBSan.

Name	File Size (KB)				
	Orig.	UBSan		CaVer	
483.xalancbmk	6,111	6,674	9%	7,169	17%
450.soplex	466	817	75%	861	84%
Chromium	249,790	612,321	145%	453,449	81%
Firefox	242,704	395,311	62%	274,254	13%

Table 5: The file size increase of instrumented binaries: CaVer incurs 64% and 49% less storage overheads in Chromium and Firefox browsers, compared to UBSan.

CVE #	Bug ID	Type Names			Security Rating	Mitigated
		Allocation	Source	Destination		by CaVer
CVE-2013-0912	180763	HTMLUnknownElement	Element	SVGElement	High	✓
CVE-2013-2931	302810	MessageEvent	Event	LocatedEvent	High	✓
CVE-2014-1731	349903	RenderListBox	RenderBlockFlow	RenderMeter	High	✓
CVE-2014-3175	387016	SpeechSynthesis	EventTarget	SpeechSynthesisUtterance	High	✓
CVE-2014-3175	387371	ThrobAnimation	Animation	MultiAnimation	Medium	✓

Table 2: Security evaluations of CaVer with known vulnerabilities of the Chromium browser. We first picked five known bad-casting bugs and wrote test cases for each vulnerability, retaining features that may affect CaVer's detection algorithm, including class hierarchy and their compositions, and related classes including allocation, source, and destination types). CaVer correctly detected all vulnerabilities.

(a) CaVer, P Alloc

From \ To	P	Non-P
P	✓	✓
Non-P	✓	✓

(b) CaVer, Non-P Alloc

From \ To	P	Non-P
P	✓	✓
Non-P	✓	✓

(c) UBSan, P Alloc

From \ To	P	Non-P
P	✓	X
Non-P	✓	X

(d) UBSan, Non-P Alloc

From \ To	P	Non-P
P	Crash	X
Non-P	Crash	X

Table 3: Evaluation of protection coverage against all possible combinations of bad-castings. P and Non-P mean polymorphic and non-polymorphic classes, respectively. In each cell, ✓marks a successful detection, X marks a failure, and Crash marks the program crashed. (a) and (b) show the results of CaVer with polymorphic class allocations and non-polymorphic class allocations, respectively, and (c) and (d) show the cases of UBSan. CaVer correctly detected all cases, while UBSan failed for 6 cases including 2 crashes.

6.6 Runtime Performance Overheads

In this subsection, we measured the runtime overheads of CaVer by using SPEC CPU 2006's C++ benchmarks and various browser benchmarks for Chromium and Firefox. For comparison, we measured runtime overheads of the original, non-instrumented version (compiled with `clang`), and the UBSan-hardened version.

Microbenchmarks. To understand the performance characteristics of CaVer-hardened applications, we first profiled micro-scaled runtime behaviors related to CaVer's operations (Table 6). For workloads, we used the built-in input for the two C++ applications of SPEC CPU 2006, and loaded the default start page of the Chromium and Firefox browsers. Overall, CaVer traced considerable number of objects, especially for the browsers: 783k in Chromium, and 15,506k in Firefox.

We counted the number of *verified castings* (see Table 6), and the kinds of allocations (i.e., global, stack, or heap). In our experiment, Firefox performed 710% more castings than Chromium, which implies that the total number of verified castings and the corresponding performance overheads highly depends on the way the application is written and its usage patterns.

SPEC CPU 2006. With these application characteristics in mind, we first measured runtime performance impacts of CaVer on two SPEC CPU 2006 programs, xalancbmk and soplex. CaVer slowed down the execution of xalancbmk and soplex by 29.6% and 20.0%, respectively. CaVer-naive (before applying the optimization techniques described in §4.4) slowed down xalancbmk and soplex by 32.7% and 20.8% respectively.

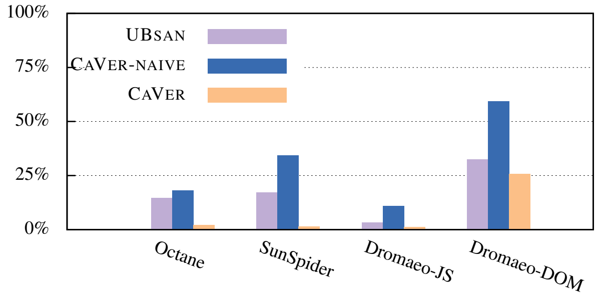


Figure 4: Browser benchmark results for the Chromium browser. On average, while CaVer-naive incurs 30.7% overhead, CaVer showed 7.6% runtime overhead after the optimization. UBSan exhibits 16.9% overhead on average.

For UBSan, xalancbmk crashed because of RTTI limitations in handling non-polymorphic types, and soplex becomes 21.1% slower. Note, the runtime overheads of CaVer is highly dependent on the application characteristics (e.g., the number of downcasts performed in runtime). Thus, we measured overhead with more realistic workloads on two popular browsers, Chromium and Firefox.

Browser benchmarks (Chromium). To understand the end-to-end performance of CaVer, we measured the performance overhead of web benchmarks. We tested four browser benchmarks: Octane [21], SunSpider [47], Dromaeo-JS [29], and Dromaeo-DOM [29], each of which evaluate either the performance of the JavaScript engine or page rendering.

Figure 4 shows the benchmark results of the Chromium browser. On average, CaVer showed 7.6% overhead while CaVer-naive showed 30.7%, which implies the

Name	Object Tracing					Verified Castings			
	Global	Stack		Heap		Global	Stack	Heap	Total
	Total	Peak	Total	Peak	Total				
483.xalancbmk	165	32	190k	8k	88k	0	104	24k	24k
450.soplex	36	1	364	141	658	0	0	0	0
Chromium	3k	274	350k	79k	453k	963	338	150k	151k
Firefox	24k	38k	14,821k	213k	685k	41k	524k	511k	1,077k

Table 6: The number of traced objects and type castings verified by CAVER in our benchmarks. Under the *Object Tracing* column, *Peak* and *Total* denote the maximum number of traced objects during program execution, and the total number of traced objects until its termination, respectively. *Global*, *Stack*, and *Heap* under the *Verified Casts* represent object's original types (allocation) involved in castings. Note that Firefox heavily allocates objects on stack, compared to Chromium. Firefox allocated 4,134% more stack objects, and performs 1,550% more type castings than Chromium.

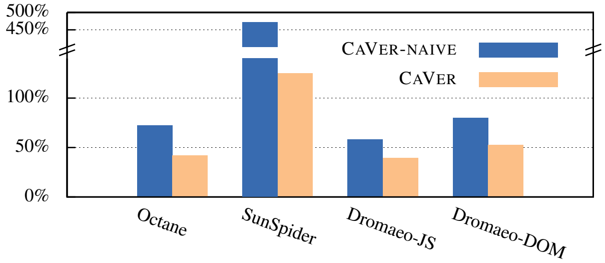


Figure 5: Browser benchmark results for the Firefox browser. On average, CAVER and CAVER-NAIVE showed 64.6% and 170.3% overhead, respectively.

Name	Original		UBSAN		CAVER	
	Peak	Avg	Peak	Avg	Peak	Avg
483.xalancbmk	9	8	crash	crash	14	12
450.soplex	2	2	2	2	5	5
Chromium	376	311	952	804	878	738
Firefox	165	121	crash	crash	208	157

Table 7: Runtime memory impacts (in KB) while running target programs. UBSAN crashed while running xalancbmk and Firefox due to the non-polymorhpic typed classes. *Peak* and *Avg* columns denote the maximum and average memory usages, respectively, while running the program. CAVER used 137% more memory on Chromium, and 23% more memory on Firefox. UBSAN used 158% more memory on Chromium.

optimization techniques in §4.4 provided a 23.1% performance improvement. This performance improvement is mostly due to the safe-allocation optimization, which identified 76,381 safe-allocation types (81% of all types used for Chromium) and opted-out to instrument allocation sites on such types. Compared to UBSAN, CAVER is 13.8% faster even though it offers more wide detection coverage on type casting. Thus, we believe this result shows that CAVER's THTable design and optimization techniques are efficient in terms of runtime performances.

Browser benchmarks (Firefox). We applied CAVER to the Firefox browser and measured the performance overhead for the web benchmarks used in evaluating the Chromium browser. On average, CAVER imposed 64.6% overhead while CAVER-NAIVE imposed 170.3% overhead (Figure 5). Similar to the Chromium case, most of performance improvements are from safe-allocation optimization, which identified 21,829 safe-allocation types (72% of all used types for Firefox). UBSAN was unable to run Firefox because it crashed due to the inability of its RTTI to handle non-polymorphic types, so we do not present the comparison number. Compared to CAVER's results on Chromium, the CAVER-enhanced Firefox showed worse performance, mainly due to the enormous amount of stack objects allocated by Firefox (Table 6). In order words, the potential performance impacts rely on the usage pattern of target applications, rather than the inherent overheads of CAVER's approaches.

Memory overheads. UBSAN and CAVER achieve fast lookup of the metadata of a given object by using a custom memory allocator that is highly optimized for this purpose, at the cost of unnecessary memory fragmentation. In our benchmark (Table 7), UBSAN used 2.5× more memory at peak and average; and CAVER used 2.3× more memory at peak and average, which is an 8% improvement over UBSAN. Considering CAVER's main purpose is a diagnosis tool and the amount of required memory is not large (< 1 GB), we believe that these memory overheads are acceptable cost in practice for the protection gained.

7 Discussion

Integration with fuzzing tools. During our evaluations, we relied on the built-in test inputs distributed with the target programs, and did not specifically attempt to improve code coverage. Yet CAVER is capable of discovering dozens of previously unknown bad-casting bugs. In the future, we plan to integrate CAVER with fuzzing tools like the ClusterFuzz [2] infrastructure for Chromium to improve code coverage. By doing so, we expect to discover more bad-casting vulnerabilities.

Optimization. In this paper, we focused on the correctness, effectiveness, and usability of CAVER. Although we developed several techniques to improve performance, optimization is not our main focus. With more powerful optimization techniques, we believe CAVER can also be used for runtime bad-casting mitigation.

For example, one direction we are pursuing is to use static analysis to prove whether a type casting is always safe. By doing so, we can remove redundant cast verification.

Another direction is to apply alignment-based direct mapping scheme for global and stack objects as well. Please recall that red-black trees used for global and stack objects show $O(logN)$ complexity, while alignment-based direct mapping scheme guarantees $O(1)$ complexity. In order to apply alignment-based direct mapping scheme for global and stack objects together, there has to be radical semantic changes in allocating stack and global objects. This is because alignment-based direct mapping scheme requires that all objects have to be strictly aligned. This may not be difficult for global objects, but designing and implementing for stack objects would be non-trivial for the following reasons: (1) essentially this may involve a knapsack problem (i.e., given different sized stack objects in each stack frame, what are the optimal packing strategies to reduce memory uses while keeping a certain alignment rule); (2) an alignment base address for each stack frame has to be independently maintained during runtime; (3) supporting variable length arrays (allowed in ISO C99 [18]) in stack would be problematic as the packing strategy can be only determined at runtime in this case.

Furthermore, it is also possible to try even more extreme approaches to apply alignment-based direct mapping scheme—simply migrating all stack objects to be allocated in heap. However, this may result in another potential side effects in overhead.

8 Related work

Bad-casting detection. The virtual function table checking in Undefined Behavior Sanitizer (UBSAN-vptr) [42] is the closest related work to CAVER. Similar to CAVER, UBSAN instruments `static_cast` at compile time, and verifies the casting at runtime. The primary difference is that UBSan relies on RTTI to retrieve the type information of an object. Thus, as we have described in §4, UBSAN suffers from several limitations of RTTI . (1) Coverage: UBSAN cannot handle non-polymorphic classes as there is no RTTI for these classes; (2) Ease-of-deployments: hardening large scale software products with UBSAN is non-trivial due to the coverage problem and phantom classes. As a result, UBSAN has to rely on blacklisting [9] to avoid crashes.

RTTI alternatives. Noticing the difficulties in handling complex C++ class hierarchies in large-scale software, several open source projects use a custom form of RTTI. For example, the LLVM project devised a custom RTTI [27]. LLVM-style RTTI requires all classes to mark its identity once it is created (i.e., in C++ constructors) and further implement a static member function to retrieve its identity. Then, all type conversions can be done with templates that leverage this static member function implemented in every class. Because the static member function can tell the true identity of an object, theoretically, all type conversions are always correct and have no bad-casting issues. Compared to CAVER, the drawback of this approach is that it requires manual source code modification. Thus, it would be non-trivial to modify large projects like browsers to switch to this style. More alarmingly, since it relies on developers' manual modification, if developers make mistakes in implementations, bad-casting can still happen [41].

Runtime type tracing. Tracing runtime type information offers several benefits, especially for debugging and profiling. [37] used RTTI to avoid complicated parsing supports in profiling parallel and scientific C++ applications. Instead of relying on RTTI, [15, 28] instruments memory allocation functions to measure complete heap memory profiles. CAVER is inspired by these runtime type tracing techniques, but it introduced the `THTable`, a unique data structure to support efficient verification of complicated type conversion.

Memory corruption prevention. As described in §2, bad-casting can provide attackers access to memory beyond the boundary of the casted object. In this case, there will be a particular violation (e.g., memory corruptions) once it is abused to mount an attack. Such violations can be detected with existing software hardening techniques, which prevents memory corruption attacks and thus potentially stop attacks abusing bad-casting. In particular, Memcheck (Valgrind) [34] and Purify [23] are popularly used solutions to detect memory errors. AddressSanitizer [36] is another popular tool developed recently by optimizing the way to represent and probe the status of allocated memory. However, it cannot detect if the attacker accesses beyond red-zones or stressing memory allocators to abuse a quarantine zone [8]. Another direction is to enforce spatial memory safety [14, 25, 32, 33, 48], but this has drawbacks when handling bad-casting issues. For example, Cyclone [25] requires extensive code modifications; CCured [33] modifies the memory allocation model; and SVA [14] depends on a new virtual execution environment. More fundamentally, most only support C programs.

Overall, compared to these solutions, we believe CAVER makes a valuable contribution because it detects the root cause of one important vulnerability type: bad-casting. CAVER can provide detailed information on how a bad-casting happens. More importantly, depending on certain test cases or workloads, many tools cannot detect bad-casting if a bad-casted pointer is not actually used to violate memory safety. However, CAVER can immediately detect such latent cases if any bad-casting occurs.

Control Flow Integrity (CFI). Similar to memory corruption prevention techniques, supporting CFI [1, 49–51] may prevent attacks abusing bad-casting as many exploits hijack control flows to mount an attack. Furthermore, specific to C++ domain, SafeDispatch [24] and VTV [45] guarantee the integrity of virtual function calls to prevent hijacks over virtual function calls. First of all, soundly implementing CFI itself is challenging. Recent research papers identified security holes in most of CFI implementations [6, 17, 19, 20]. More importantly, all of these solutions are designed to only protect control-data, and thus it cannot detect any non-control data attacks [7]. For example, the recent vulnerability exploit against `glibc` [35] was able to completely subvert the victim's system by merely overwriting non-control data—`EXIM`'s runtime configuration. However, because CAVER is not relying on such post-behaviors originating from bad-casting, it is agnostic to specific exploit methods.

9 Conclusion

The bad-casting problem in C++ programs, which occurs when the type of an object pointer is converted to another that is incorrect and unsafe, has serious security implications. We have developed CAVER, a runtime bad-casting detection tool. It uses a new runtime type tracing mechanism, the Type Hierarchy Table, to efficiently verify type casting dynamically. CAVER provides broader coverage than existing approaches with smaller or comparable performance overhead. We have implemented CAVER and have applied it to large-scale software including the Chromium and Firefox browsers. To date, CAVER has found eleven previously unknown vulnerabilities, which have been reported and subsequently fixed by the corresponding open-source communities.

Acknowledgment

The authors would like to thank the anonymous reviewers for their helpful feedback, as well as our operations staff for their proofreading efforts. We also would like to thank Abhishek Arya, Kostya Serebryany, Alexey Samsonov, Richard Smith, and Parisa Tabriz for their helpful feedback on the paper. This material is based upon work supported in part by the National Science Foundation under Grants No. CNS-1017265, CNS-0831300, CNS-1149051 and DGE 1500084, by the Office of Naval Research under Grant No. N000140911042, No. N000141512162, by the Department of Homeland Security under contract No. N66001-12-C-0133, by the United States Air Force under Contract No. FA8650-10-C-7025, and by ETRI MSIP/IITP[B0101-15-0644]. 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, the Office of Naval Research, the Department of Homeland Security, or the United States Air Force.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[2] C. N. Abhishek Arya. Fuzzing for Security (The Chromium Blog). http://blog.chromium.org/2012/04/fuzzing-for-security.html, 2012.

[3] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. In *USENIX Security Symposium (Security)*, 2009.

[4] Bad casting: From BasicThebesLayer to BasicContainerLayer. Mozilla Bugzilla - Bug 1074280. https://bugzilla.mozilla.org/show_bug.cgi?id=1074280, Nov 2014.

[5] Bad-casting from PRCListStr to nsSHistory. Mozilla Bugzilla - Bug 1089438. https://bugzilla.mozilla.org/show_bug.cgi?id=1089438, Nov 2014.

[6] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *USENIX Security Symposium (Security)*, 2014.

[7] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data Attacks Are Realistic Threats. In *USENIX Security Symposium (Security)*, 2005.

[8] Chris Evans. Using ASAN as a protection. http://scarybeastsecurity.blogspot.com/2014/09/using-asan-as-protection.html, Nov 2014.

[9] Chromium. Chromium Revision 285353 - Blacklist for UBsan's vptr. https://src.chromium.org/viewvc/chrome?view=revision&revision=285353, Jul 2014.

[10] Chromium. Chromium project: Log of /trunk/src/tools/ubsan_-vptr/blacklist.txt. https://src.chromium.org/viewvc/chrome/trunk/src/tools/ubsan_vptr/blacklist.txt?view=log, Nov 2014.

[11] Clang Documentation. MSVC Compatibility. http://clang.llvm.org/docs/MSVCCompatibility.html, Nov 2014.

[12] CodeSourcery, Compaq, EDG, HP, IBM, Intel, R. Hat, and SGI. Itanium C++ ABI (Revision: 1.83). http://mentorembedded.github.io/cxx-abi/abi.html, 2005.

[13] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI. C++ ABI Closed Issues. www.codesourcery.com/public/cxx-abi/cxx-closed.html, Nov 2014.

[14] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

[15] Dai Mikurube. C++ Object Type Identifier for Heap Profiling. http://www.chromium.org/developers/deep-memory-profiler/cpp-object-type-identifier, Nov 2014.

[16] M. Daniel, J. Honoroff, and C. Miller. Engineering Heap Overflow Exploits with JavaScript. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2008.

[17] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *USENIX Security Symposium (Security)*, 2014.

[18] GCC. Arrays of Variable Length. https://gcc.gnu.org/onlinedocs/gcc/Variable-Length.html, Jun 2015.

[19] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming Control-Flow Integrity. In *IEEE Symposium on Security and Privacy (SP)*, 2014.

[20] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *USENIX Security Symposium (Security)*, 2014.

[21] Google. Octane Benchmark. https://code.google.com/p/octane-benchmark, Aug 2014.

[22] Google. Specialized memory allocator for ThreadSanitizer, MemorySanitizer, etc. http://llvm.org/klaus/compiler-rt/blob/7385f8b8b8723064910cf9737dc929e90aeac548/lib/sanitizer_common/sanitizer_allocator.h, Nov 2014.

[23] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Winter 1992 USENIX Conference*, 1991.

[24] D. Jang, Z. Tatlock, and S. Lerner. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[25] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference (ATC)*, 2002.

[26] I. JTC1/SC22/WG21. ISO/IEC 14882:2013 Programming Language C++ (N3690). https://isocpp.org/files/papers/N3690.pdf, 2013.

[27] LLVM Project. How to set up LLVM-style RTTI for your class hierarchy. http://llvm.org/docs/HowToSetUpLLVMStyleRTTI.html, Nov 2014.

[28] J. Mihalicza, Z. Porkoláb, and A. Gabor. Type-preserving heap profiler for c++. In *IEEE International Conference on Software Maintenance (ICSM)*, 2011.

[29] Mozilla. DROMAEO, JavaScript Performance Testing. http://dromaeo.com, Aug 2014.

[30] Multiple undefined behaviors (static_cast<>) in libstdc++-v3/include/bits. GCC Bugzilla - Bug 63345. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=63345, Nov 2014.

[31] MWR Labs. MWR Labs Pwn2Own 2013 Write-up: Webkit Exploit. https://labs.mwrinfosecurity.com/blog/2013/04/19/mwr-labs-pwn2own-2013-write-up---webkit-exploit/, 2013.

[32] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[33] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2005.

[34] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[35] Qualys Security Advisory. Qualys Security Advisory CVE-2015-0235 - GHOST: glibc gethostbyname buffer overflow. http://www.openwall.com/lists/oss-security/2015/01/27/9, Jun 2015.

[36] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference (ATC)*, 2012.

[37] S. Shende, A. D. Malony, J. Cuny, P. Beckman, S. Karmesin, and K. Lindlan. Portable profiling and tracing for parallel, scientific applications using c++. In *ACM SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT)*, 1998.

[38] A. Sotirov. Heap Feng Shui in JavaScript. *Black Hat Europe*, 2007.

[39] Standard Performance Evaluation Corporation. SPEC CPU 2006. http://www.spec.org/cpu2006, Aug 2014.

[40] The Chromium Project. http://www.chromium.org/Home, Aug 2014.

[41] The Chromium Project. Chromium Issues - Bug 387016. http://code.google.com/p/chromium/issues/detail?id=387016, Nov 2014.

[42] The Chromium Projects. Undefined Behavior Sanitizer for Chromium. http://www.chromium.org/developers/testing/undefinedbehaviorsanitizer, Nov 2014.

[43] The LLVM Compiler Infrastructure. http://llvm.org, Aug 2014.

[44] The Mozilla Foundation. Firefox Web Browser. https://www.mozilla.org/firefox, Nov 2014.

[45] C. Tice. Improving Function Pointer Security for Virtual Method Dispatches. In *GNU Tools Cauldron Workshop*, 2012.

[46] TIS Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2. *TIS Committee*, 1995.

[47] WebKit. SunSpider 1.0.2 JavaScript Benchmark. https://www.webkit.org/perf/sunspider/sunspider.html, Aug 2014.

[48] W. Xu, D. C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of c programs. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2004.

[49] B. Zeng, G. Tan, and G. Morrisett. Combining Control-flow Integrity and Static Analysis for Efficient and Validated Data Sandboxing. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[50] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical Control Flow Integrity and Randomization for Binary Executables. In *IEEE Symposium on Security and Privacy (SP)*, 2013.

[51] M. Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *USENIX Security Symposium (Security)*, 2013.

Appendix

```
def getTHTableByAddr(addr):
    if isObjectInStack(addr):
        # addr points to stack objects.
        stack_rbtree = getThreadLocalStackRbtree()
        allocBaseAddr, pTHTable = \
                           stack_rbtree.rangedSearch(addr):
        return (allocBaseAddr, pTHTable)

    if isObjectInHeap(addr):
        # addr points to heap objects.
        MetaData = getMetaDataStorage(addr):
        return (MetaData.allocBaseAddr, MetaData.pTHTable)

    if isObjectInGlobal(addr):
        # addr points to global objects.
        allocBaseAddr, pTHTable = \
                           global_rbtree.rangedSearch(addr):
        return (allocBaseAddr, pTHTable)

    # addr points to unknown area.
    return ERROR

# Return True if there exists a good-casting.
#         False if there is no good-castings.
def isGoodCast(addr, allocBaseAddr, THTable, TargetTypeHash):
    # Handle compositions recursively.
    for i in range(THTable.num_composites):
        comp = THTable.comps[i]
        if addr >= allocBaseAddr + comp.offset
          and addr < allocBaseAddr + comp.offset + comp.size:
            if isGoodCast(addr, allocBaseAddr + comp.offset,
                            comp.thtable, TargetTypeHash):
                return True

    # Check bases.
    for i in range(THTable.num_bases):
        base = THTable.bases[i]
        if addr == allocBaseAddr + base.offset
          and base.hashValue == TargetTypeHash:
            return True

    # Check phantom.
    TargetTHTable = getTHTableByHash(TargetTypeHash)
    for i in range(TargetTHTable.num_bases):
        base = TargetTHTable.bases[i]
        if addr == allocBaseAddr + base.offset
          and base.hashValue == THTable.type_hash
          and base.isPhantom:
            return True

    return False

def verify_cast(beforeAddr, afterAddr, TargetTypeHash):
    (allocBaseAddr, pTHTable) = getTHTableByAddr(beforeAddr)
    if pTHTable == ERROR:
        return

    if isGoodCast(afterAddr, allocBaseAddr, \
                  THTable, TargetTypeHash):
        # This is a good casting.
        return

    # Reaching here means a bad-casting attempt is detected.
    # Below may report the bug, halt the program, or nullify
    # the pointer according to the user’s configuration.
    HandleBadCastingAttempt()
```

Appendix 1: Algorithm for verifying type conversions based on the tracked type information.

```
# global_rbtree is initialized per process.
def trace_global(pTHTable, baseAddr, numArrayElements):
    allocSize = pTHTable.type_size * numArrayElements
    global_rbtree.insert((baseAddr, allocSize), pTHTable)
    return

# stack_rbtree is initialized per thread.
def trace_stack_begin(pTHTable, baseAddr, numArrayElements):
    stack_rbtree = getThreadLocalStackRbtree()
    allocSize = pTHTable.type_size * numArrayElements
    stack_rbtree.insert((baseAddr, allocSize), pTHTable)
    return

def trace_stack_end(baseAddr):
    stack_rbtree = getThreadLocalStackRbtree()
    stack_rbtree.remove(baseAddr)
    return

# Meta-data storage for dynamic objects are reserved
# for each object allocation.
def trace_heap(pTHTable, baseAddr, numArrayElements):
    MetaData = getMetaDataStorage(baseAddr)
    MetaData.baseAddr = baseAddr
    MetaData.allocSize = pTHTable.type_size * numArrayElements
    MetaData.pTHTable = pTHTable
    return
```

Appendix 2: Algorithm for tracking type information on objects in runtime.

All Your Biases Belong To Us: Breaking RC4 in WPA-TKIP and TLS

Mathy Vanhoef
KU Leuven
Mathy.Vanhoef@cs.kuleuven.be

Frank Piessens
KU Leuven
Frank.Piessens@cs.kuleuven.be

Abstract

We present new biases in RC4, break the Wi-Fi Protected Access Temporal Key Integrity Protocol (WPA-TKIP), and design a practical plaintext recovery attack against the Transport Layer Security (TLS) protocol. To empirically find new biases in the RC4 keystream we use statistical hypothesis tests. This reveals many new biases in the initial keystream bytes, as well as several new long-term biases. Our fixed-plaintext recovery algorithms are capable of using multiple types of biases, and return a list of plaintext candidates in decreasing likelihood.

To break WPA-TKIP we introduce a method to generate a large number of identical packets. This packet is decrypted by generating its plaintext candidate list, and using redundant packet structure to prune bad candidates. From the decrypted packet we derive the TKIP MIC key, which can be used to inject and decrypt packets. In practice the attack can be executed within an hour. We also attack TLS as used by HTTPS, where we show how to decrypt a secure cookie with a success rate of 94% using $9 \cdot 2^{27}$ ciphertexts. This is done by injecting known data around the cookie, abusing this using Mantin's *ABSAB* bias, and brute-forcing the cookie by traversing the plaintext candidates. Using our traffic generation technique, we are able to execute the attack in merely 75 hours.

1 Introduction

RC4 is (still) one of the most widely used stream ciphers. Arguably its most well known usage is in SSL and WEP, and in their successors TLS [8] and WPA-TKIP [19]. In particular it was heavily used after attacks against CBC-mode encryption schemes in TLS were published, such as BEAST [9], Lucky 13 [1], and the padding oracle attack [7]. As a mitigation RC4 was recommended. Hence, at one point around 50% of all TLS connections were using RC4 [2], with the current estimate around 30% [18]. This motivated the search for new attacks, relevant examples being [2, 20, 31, 15, 30]. Of special interest is the attack proposed by AlFardan et al., where roughly $13 \cdot 2^{30}$ ciphertexts are required to decrypt a cookie sent over HTTPS [2]. This corresponds to about 2000 hours of data in their setup, hence the attack is considered close to being practical. Our goal is to see how far these attacks can be pushed by exploring three areas. First, we search for new biases in the keystream. Second, we improve fixed-plaintext recovery algorithms. Third, we demonstrate techniques to perform our attacks in practice.

First we empirically search for biases in the keystream. This is done by generating a large amount of keystream, and storing statistics about them in several datasets. The resulting datasets are then analysed using statistical hypothesis tests. Our null hypothesis is that a keystream byte is uniformly distributed, or that two bytes are independent. Rejecting the null hypothesis is equivalent to detecting a bias. Compared to manually inspecting graphs, this allows for a more large-scale analysis. With this approach we found many new biases in the initial keystream bytes, as well as several new long-term biases.

We break WPA-TKIP by decrypting a complete packet using RC4 biases and deriving the TKIP MIC key. This key can be used to inject and decrypt packets [48]. In particular we modify the plaintext recovery attack of Paterson et al. [31, 30] to return a list of candidates in decreasing likelihood. Bad candidates are detected and pruned based on the (decrypted) CRC of the packet. This increases the success rate of simultaneously decrypting all unknown bytes. We achieve practicality using a novel method to rapidly inject identical packets into a network. In practice the attack can be executed within an hour.

We also attack RC4 as used in TLS and HTTPS, where we decrypt a secure cookie in realistic conditions. This is done by combining the *ABSAB* and Fluhrer-McGrew biases using variants of the of Isobe et al. and AlFardan et al. attack [20, 2]. Our technique can easily be extended to include other biases as well. To abuse Mantin's *ABSAB* bias we inject known plaintext around the cookie, and exploit this to calculate Bayesian plaintext likelihoods over the unknown cookie. We then generate a list of (cookie) candidates in decreasing likelihood, and use this to brute-force the cookie in negligible time. The algorithm to generate candidates differs from the WPA-TKIP one due to the reliance on double-byte instead of single-byte likelihoods. All combined, we need $9 \cdot 2^{27}$ encryptions of a cookie to decrypt it with a success rate of 94%. Finally we show how to make a victim generate this amount within only 75 hours, and execute the attack in practice.

To summarize, our main contributions are:

- We use statistical tests to empirically detect biases in the keystream, revealing large sets of new biases.
- We design plaintext recovery algorithms capable of using multiple types of biases, which return a list of plaintext candidates in decreasing likelihood.
- We demonstrate practical exploitation techniques to break RC4 in both WPA-TKIP and TLS.

The remainder of this paper is organized as follows. Section 2 gives a background on RC4, TKIP, and TLS. In Sect. 3 we introduce hypothesis tests and report new biases. Plaintext recovery techniques are given in Sect. 4. Practical attacks on TKIP and TLS are presented in Sect. 5 and Sect. 6, respectively. Finally, we summarize related work in Sect. 7 and conclude in Sect. 8.

2 Background

We introduce RC4 and its usage in TLS and WPA-TKIP.

2.1 The RC4 Algorithm

The RC4 algorithm is intriguingly short and known to be very fast in software. It consists of a Key Scheduling Algorithm (KSA) and a Pseudo Random Generation Algorithm (PRGA), which are both shown in Fig. 1. The state consists of a permutation $\mathcal{S}$ of the set $\{0,\ldots,255\}$, a public counter i, and a private index j. The KSA takes as input a variable-length key and initializes $\mathcal{S}$. At each round $r = 1,2,\ldots$ of the PRGA, the yield statement outputs a keystream byte Z_r. All additions are performed modulo 256. A plaintext byte P_r is encrypted to ciphertext byte C_r using $C_r = P_r \oplus Z_r$.

2.1.1 Short-Term Biases

Several biases have been found in the initial RC4 keystream bytes. We call these short-term biases. The most significant one was found by Mantin and Shamir. They showed that the second keystream byte is twice as likely to be zero compared to uniform [25]. Or more formally that $\Pr[Z_2 = 0] \approx 2 \cdot 2^{-8}$, where the probability is over the random choice of the key. Because zero occurs more often than expected, we call this a positive bias. Similarly, a value occurring less often than expected is called a negative bias. This result was extended by Maitra et al. [23] and further refined by Sen Gupta et al. [38] to show that there is a bias towards zero for most initial keystream bytes. Sen Gupta et al. also found key-length dependent biases: if ℓ is the key length, keystream byte Z_ℓ has a positive bias towards $256-\ell$ [38]. AlFardan et al. showed that all initial 256 keystream bytes are biased by empirically estimating their probabilities when 16-byte keys are used [2]. While doing this they found additional strong biases, an example being the bias towards value r for all positions $1 \leq r \leq 256$. This bias was also independently discovered by Isobe et al. [20].

Listing (1) RC4 Key Scheduling (KSA).

```
j, S = 0, range(256)
for i in range(256):
  j += S[i] + key[i % len(key)]
  swap(S[i], S[j])
return S
```

Listing (2) RC4 Keystream Generation (PRGA).

```
S, i, j = KSA(key), 0, 0
while True:
  i += 1
  j += S[i]
  swap(S[i], S[j])
  yield S[S[i] + S[j]]
```

Figure 1: Implementation of RC4 in Python-like pseudo-code. All additions are performed modulo 256.

The bias $\Pr[Z_1 = Z_2] = 2^{-8}(1-2^{-8})$ was found by Paul and Preneel [33]. Isobe et al. refined this result for the value zero to $\Pr[Z_1 = Z_2 = 0] \approx 3 \cdot 2^{-16}$ [20]. In [20] the authors searched for biases of similar strength between initial bytes, but did not find additional ones. However, we did manage to find new ones (see Sect. 3.3).

2.1.2 Long-Term Biases

In contrast to short-term biases, which occur only in the initial keystream bytes, there are also biases that keep occurring throughout the whole keystream. We call these long-term biases. For example, Fluhrer and McGrew (FM) found that the probability of certain digraphs, i.e., consecutive keystream bytes (Z_r, Z_{r+1}), deviate from uniform throughout the whole keystream [13]. These biases depend on the public counter i of the PRGA, and are listed in Table 1 (ignoring the condition on r for now). In their analysis, Fluhrer and McGrew assumed that the internal state of the RC4 algorithm was uniformly random.

Digraph	Condition	Probability
(0,0)	$i = 1$	$2^{-16}(1+2^{-7})$
(0,0)	$i \neq 1, 255$	$2^{-16}(1+2^{-8})$
(0,1)	$i \neq 0, 1$	$2^{-16}(1+2^{-8})$
(0,$i+1$)	$i \neq 0, 255$	$2^{-16}(1-2^{-8})$
($i+1$,255)	$i \neq 254 \wedge r \neq 1$	$2^{-16}(1+2^{-8})$
(129,129)	$i = 2, r \neq 2$	$2^{-16}(1+2^{-8})$
(255,$i+1$)	$i \neq 1, 254$	$2^{-16}(1+2^{-8})$
(255,$i+2$)	$i \in [1, 252] \wedge r \neq 2$	$2^{-16}(1+2^{-8})$
(255,0)	$i = 254$	$2^{-16}(1+2^{-8})$
(255,1)	$i = 255$	$2^{-16}(1+2^{-8})$
(255,2)	$i = 0, 1$	$2^{-16}(1+2^{-8})$
(255,255)	$i \neq 254 \wedge r \neq 5$	$2^{-16}(1-2^{-8})$

Table 1: Generalized Fluhrer-McGrew (FM) biases. Here i is the public counter in the PRGA and r the position of the first byte of the digraph. Probabilities for long-term biases are shown (for short-term biases see Fig. 4).

This assumption is only true after a few rounds of the PRGA [13, 26, 38]. Consequently these biases were generally not expected to be present in the initial keystream bytes. However, in Sect. 3.3.1 we show that most of these biases do occur in the initial keystream bytes, albeit with different probabilities than their long-term variants.

Another long-term bias was found by Mantin [24]. He discovered a bias towards the pattern $ABSAB$, where A and B represent byte values, and $\mathcal{S}$ a short sequence of bytes called the gap. With the length of the gap $\mathcal{S}$ denoted by g, the bias can be written as:

$$\Pr[(Z_r, Z_{r+1}) = (Z_{r+g+2}, Z_{r+g+3})] = 2^{-16}(1+2^{-8}e^{\frac{-4-8g}{256}}) \tag{1}$$

Hence the bigger the gap, the weaker the bias. Finally, Sen Gupta et al. found the long-term bias [38]

$$\Pr[(Z_{w256}, Z_{w256+2}) = (0,0)] = 2^{-16}(1+2^{-8})$$

where $w \geq 1$. We discovered that a bias towards $(128, 0)$ is also present at these positions (see Sect. 3.4).

2.2 TKIP Cryptographic Encapsulation

The design goal of WPA-TKIP was for it to be a temporary replacement of WEP [19, §11.4.2]. While it is being phased out by the WiFi Alliance, a recent study shows its usage is still widespread [48]. Out of 6803 networks, they found that 71% of protected networks still allow TKIP, with 19% exclusively supporting TKIP.

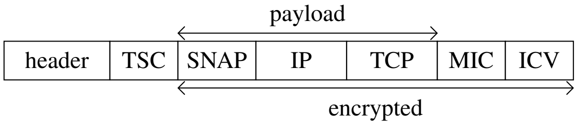


Figure 2: Simplified TKIP frame with a TCP payload.

Our attack on TKIP relies on two elements of the protocol: its weak Message Integrity Check (MIC) [44, 48], and its faulty per-packet key construction [2, 15, 31, 30]. We briefly introduce both aspects, assuming a 512-bit Pairwise Transient Key (PTK) has already been negotiated between the Access Point (AP) and client. From this PTK a 128-bit temporal encryption key (TK) and two 64-bit Message Integrity Check (MIC) keys are derived. The first MIC key is used for AP-to-client communication, and the second for the reverse direction. Some works claim that the PTK, and its derived keys, are renewed after a user-defined interval, commonly set to 1 hour [44, 48]. However, we found that generally only the Groupwise Transient Key (GTK) is periodically renewed. Interestingly, our attack can be executed within an hour, so even networks which renew the PTK every hour can be attacked.

When the client wants to transmit a payload, it first calculates a MIC value using the appropriate MIC key and the Micheal algorithm (see Fig. Figure 2). Unfortunately Micheal is straightforward to invert: given plaintext data and its MIC value, we can efficiently derive the MIC key [44]. After appending the MIC value, a CRC checksum called the Integrity Check Value (ICV) is also appended. The resulting packet, including MAC header and example TCP payload, is shown in Figure 2. The payload, MIC, and ICV are encrypted using RC4 with a per-packet key. This key is calculated by a mixing function that takes as input the TK, the TKIP sequence counter (TSC), and the transmitter MAC address (TA). We write this as $\mathrm{K} = \mathrm{KM}(\mathrm{TA}, \mathrm{TK}, \mathrm{TSC})$. The TSC is a 6-byte counter that is incremented after transmitting a packet, and is included unencrypted in the MAC header. In practice the output of KM can be modelled as uniformly random [2, 31]. In an attempt to avoid weak-key attacks that broke WEP [12], the first three bytes of K are set to [19, §11.4.2.1.1]:

$$\mathrm{K}_0 = \mathrm{TSC}_1 \quad \mathrm{K}_1 = (\mathrm{TSC}_1 \mid 0\mathrm{x}20) \,\&\, 0\mathrm{x}7\mathrm{f} \quad \mathrm{K}_2 = \mathrm{TSC}_0$$

Here, TSC_0 and TSC_1 are the two least significant bytes of the TSC. Since the TSC is public, so are the first three bytes of K. Both formally and using simulations, it has been shown this actually weakens security [2, 15, 31, 30].

2.3 The TLS Record Protocol

We focus on the TLS record protocol when RC4 is selected as the symmetric cipher [8]. In particular we assume the handshake phase is completed, and a 48-byte TLS master secret has been negotiated.

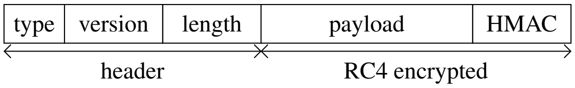


Figure 3: TLS Record structure when using RC4.

To send an encrypted payload, a TLS record of type application data is created. It contains the protocol version, length of the encrypted content, the payload itself, and finally an HMAC. The resulting layout is shown in Fig. 3. The HMAC is computed over the header, a sequence number incremented for each transmitted record, and the plaintext payload. Both the payload and HMAC are encrypted. At the start of a connection, RC4 is initialized with a key derived from the TLS master secret. This key can be modelled as being uniformly random [2]. None of the initial keystream bytes are discarded.

In the context of HTTPS, one TLS connection can be used to handle multiple HTTP requests. This is called a persistent connection. Slightly simplified, a server indicates support for this by setting the HTTP `Connection` header to `keep-alive`. This implies RC4 is initialized only once to send all HTTP requests, allowing the usage of long-term biases in attacks. Finally, cookies can be marked as being `secure`, assuring they are transmitted only over a TLS connection.

3 Empirically Finding New Biases

In this section we explain how to empirically yet soundly detect biases. While we discovered many biases, we will not use them in our attacks. This simplifies the description of the attacks. And, while using the new biases may improve our attacks, using existing ones already sufficed to significantly improve upon existing attacks. Hence our focus will mainly be on the most intriguing new biases.

3.1 Soundly Detecting Biases

In order to empirically detect new biases, we rely on hypothesis tests. That is, we generate keystream statistics over random RC4 keys, and use statistical tests to uncover deviations from uniform. This allows for a large-scale and automated analysis. To detect single-byte biases, our null hypothesis is that the keystream byte values are uniformly distributed. To detect biases between two bytes, one may be tempted to use as null hypothesis that the pair is uniformly distributed. However, this falls short if there are already single-byte biases present. In this case single-byte biases imply that the pair is also biased, while both bytes may in fact be independent. Hence, to detect double-byte biases, our null hypothesis is that they are independent. With this test, we even detected pairs that are actually more uniform than expected. Rejecting the null hypothesis is now the same as detecting a bias.

To test whether values are uniformly distributed, we use a chi-squared goodness-of-fit test. A naive approach to test whether two bytes are independent, is using a chi-squared independence test. Although this would work, it is not ideal when only a few biases (outliers) are present. Moreover, based on previous work we expect that only a few values between keystream bytes show a clear dependency on each other [13, 24, 20, 38, 4]. Taking the Fluhrer-McGrew biases as an example, at any position at most 8 out of a total 65536 value pairs show a clear bias [13]. When expecting only a few outliers, the M-test of Fuchs and Kenett can be asymptotically more powerful than the chi-squared test [14]. Hence we used the M-test to detect dependencies between keystream bytes. To determine which values are biased between dependent bytes, we perform proportion tests over all value pairs.

We reject the null hypothesis only if the p-value is lower than 10^{-4}. Holm's method is used to control the family-wise error rate when performing multiple hypothesis tests. This controls the probability of even a single false positive over all hypothesis tests. We always use the two-sided variant of an hypothesis test, since a bias can be either positive or negative.

Simply giving or plotting the probability of two dependent bytes is not ideal. After all, this probability includes the single-byte biases, while we only want to report the strength of the dependency between both bytes. To solve this, we report the absolute relative bias compared to the expected single-byte based probability. More precisely, say that by multiplying the two single-byte probabilities of a pair, we would expect it to occur with probability p. Given that this pair actually occurs with probability s, we then plot the value $|q|$ from the formula $s = p \cdot (1+q)$. In a sense the relative bias indicates how much information is gained by not just considering the single-byte biases, but using the real byte-pair probability.

3.2 Generating Datasets

In order to generate detailed statistics of keystream bytes, we created a distributed setup. We used roughly 80 standard desktop computers and three powerful servers as workers. The generation of the statistics is done in C. Python was used to manage the generated datasets and control all workers. On start-up each worker generates a cryptographically random AES key. Random 128-bit RC4 keys are derived from this key using AES in counter mode. Finally, we used R for all statistical analysis [34].

Our main results are based on two datasets, called `first16` and `consec512`. The `first16` dataset estimates $\Pr[Z_a = x \wedge Z_b = y]$ for $1 \leq a \leq 16$, $1 \leq b \leq 256$, and $0 \leq x, y < 256$ using 2^{44} keys. Its generation took

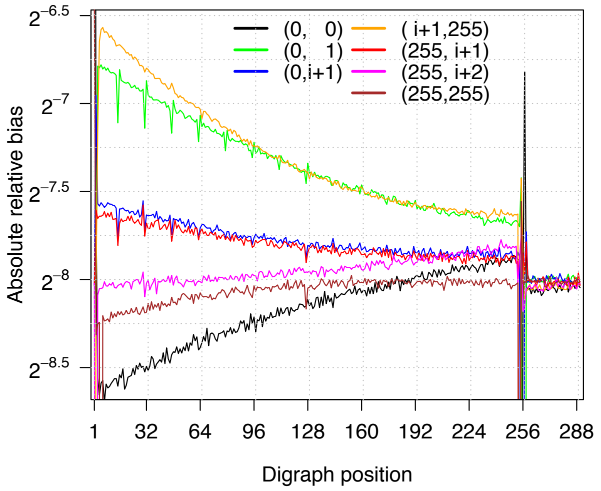


Figure 4: Absolute relative bias of several Fluhrer-McGrew digraphs in the initial keystream bytes, compared to their expected single-byte based probability.

roughly 9 CPU years. This allows detecting biases between the first 16 bytes and the other initial 256 bytes. The `consec512` dataset estimates $\Pr[Z_r = x \wedge Z_{r+1} = y]$ for $1 \leq r \leq 512$ and $0 \leq x, y < 256$ using 2^{45} keys, which took 16 CPU years to generate. It allows a detailed study of consecutive keystream bytes up to position 512.

We optimized the generation of both datasets. The first optimization is that one run of a worker generates at most 2^{30} keystreams. This allows usage of 16-bit integers for all counters collecting the statistics, even in the presence of significant biases. Only when combining the results of workers are larger integers required. This lowers memory usage, reducing cache misses. To further reduce cache misses we generate several keystreams before updating the counters. In independent work, Paterson et al. used similar optimizations [30]. For the `first16` dataset we used an additional optimization. Here we first generate several keystreams, and then update the counters in a sorted manner based on the value of Z_a. This optimization caused the most significant speed-up for the `first16` dataset.

3.3 New Short-Term Biases

By analysing the generated datasets we discovered many new short-term biases. We classify them into several sets.

3.3.1 Biases in (Non-)Consecutive Bytes

By analysing the `consec512` dataset we discovered numerous biases between consecutive keystream bytes. Our first observation is that the Fluhrer-McGrew biases are also present in the initial keystream bytes. Exceptions occur at positions 1, 2 and 5, and are listed in Table 1 (note the extra conditions on the position r). This is surprising, as the Fluhrer-McGrew biases were generally not expected to be present in the initial keystream bytes [13]. However, these biases are present, albeit with different probabilities. Figure 4 shows the absolute relative bias of most Fluhrer-McGrew digraphs, compared to their expected single-byte based probability (recall Sect. 3.1). For all digraphs, the sign of the relative bias q is the same as its long-term variant as listed in Table 1. We observe that the relative biases converge to their long-term values, especially after position 257. The vertical lines around position 1 and 256 are caused by digraphs which do not hold (or hold more strongly) around these positions.

First byte	Second byte	Probability
Consecutive biases:		
$Z_{15} = 240$	$Z_{16} = 240$	$2^{-15.94786}(1 - 2^{-4.894})$
$Z_{31} = 224$	$Z_{32} = 224$	$2^{-15.96486}(1 - 2^{-5.427})$
$Z_{47} = 208$	$Z_{48} = 208$	$2^{-15.97595}(1 - 2^{-5.963})$
$Z_{63} = 192$	$Z_{64} = 192$	$2^{-15.98363}(1 - 2^{-6.469})$
$Z_{79} = 176$	$Z_{80} = 176$	$2^{-15.99020}(1 - 2^{-7.150})$
$Z_{95} = 160$	$Z_{96} = 160$	$2^{-15.99405}(1 - 2^{-7.740})$
$Z_{111} = 144$	$Z_{112} = 144$	$2^{-15.99668}(1 - 2^{-8.331})$
Non-consecutive biases:		
$Z_3 = 4$	$Z_5 = 4$	$2^{-16.00243}(1 + 2^{-7.912})$
$Z_3 = 131$	$Z_{131} = 3$	$2^{-15.99543}(1 + 2^{-8.700})$
$Z_3 = 131$	$Z_{131} = 131$	$2^{-15.99347}(1 - 2^{-9.511})$
$Z_4 = 5$	$Z_6 = 255$	$2^{-15.99918}(1 + 2^{-8.208})$
$Z_{14} = 0$	$Z_{16} = 14$	$2^{-15.99349}(1 + 2^{-9.941})$
$Z_{15} = 47$	$Z_{17} = 16$	$2^{-16.00191}(1 + 2^{-11.279})$
$Z_{15} = 112$	$Z_{32} = 224$	$2^{-15.96637}(1 - 2^{-10.904})$
$Z_{15} = 159$	$Z_{32} = 224$	$2^{-15.96574}(1 + 2^{-9.493})$
$Z_{16} = 240$	$Z_{31} = 63$	$2^{-15.95021}(1 + 2^{-8.996})$
$Z_{16} = 240$	$Z_{32} = 16$	$2^{-15.94976}(1 + 2^{-9.261})$
$Z_{16} = 240$	$Z_{33} = 16$	$2^{-15.94960}(1 + 2^{-10.516})$
$Z_{16} = 240$	$Z_{40} = 32$	$2^{-15.94976}(1 + 2^{-10.933})$
$Z_{16} = 240$	$Z_{48} = 16$	$2^{-15.94989}(1 + 2^{-10.832})$
$Z_{16} = 240$	$Z_{48} = 208$	$2^{-15.92619}(1 - 2^{-10.965})$
$Z_{16} = 240$	$Z_{64} = 192$	$2^{-15.93357}(1 - 2^{-11.229})$

Table 2: Biases between (non-consecutive) bytes.

A second set of strong biases have the form:

$$\Pr[Z_{w16-1} = Z_{w16} = 256 - w16] \qquad (2)$$

with $1 \leq w \leq 7$. In Table 2 we list their probabilities. Since 16 equals our key length, these are likely key-length dependent biases.

Another set of biases have the form $\Pr[Z_r = Z_{r+1} = x]$. Depending on the value x, these biases are either negative or positive. Hence summing over all x and calculating $\Pr[Z_r = Z_{r+1}]$ would lose some statistical informa-

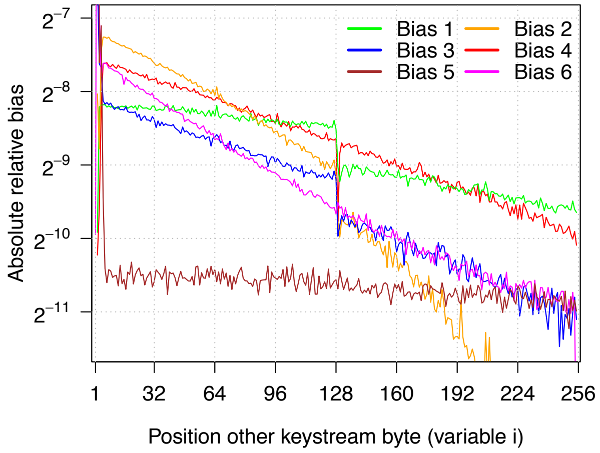


Figure 5: Biases induced by the first two bytes. The number of the biases correspond to those in Sect. 3.3.2.

tion. In principle, these biases also include the Fluhrer-McGrew pairs $(0,0)$ and $(255,255)$. However, as the bias for both these pairs is much higher than for other values, we don't include them here. Our new bias, in the form of $\Pr[Z_r = Z_{r+1}]$, was detected up to position 512.

We also detected biases between non-consecutive bytes that do not fall in any obvious categories. An overview of these is given in Table 2. We remark that the biases induced by $Z_{16} = 240$ generally have a position, or value, that is a multiple of 16. This is an indication that these are likely key-length dependent biases.

3.3.2 Influence of Z_1 and Z_2

Arguably our most intriguing finding is the amount of information the first two keystream bytes leak. In particular, Z_1 and Z_2 influence all initial 256 keystream bytes. We detected the following six sets of biases:

1) $Z_1 = 257 - i \wedge Z_i = 0$ 4) $Z_1 = i - 1 \wedge Z_i = 1$
2) $Z_1 = 257 - i \wedge Z_i = i$ 5) $Z_2 = 0 \wedge Z_i = 0$
3) $Z_1 = 257 - i \wedge Z_i = 257 - i$ 6) $Z_2 = 0 \wedge Z_i = i$

Their absolute relative bias, compared to the single-byte biases, is shown in Fig. 5. The relative bias of pairs 5 and 6, i.e., those involving Z_2, are generally negative. Pairs involving Z_1 are generally positive, except pair 3, which always has a negative relative bias. We also detected dependencies between Z_1 and Z_2 other than the $\Pr[Z_1 = Z_2]$ bias of Paul and Preneel [33]. That is, the following pairs are strongly biased:

A) $Z_1 = 0 \wedge Z_2 = x$ C) $Z_1 = x \wedge Z_2 = 0$
B) $Z_1 = x \wedge Z_2 = 258 - x$ D) $Z_1 = x \wedge Z_2 = 1$

Bias A and C are negative for all $x \neq 0$, and both appear to be mainly caused by the strong positive bias

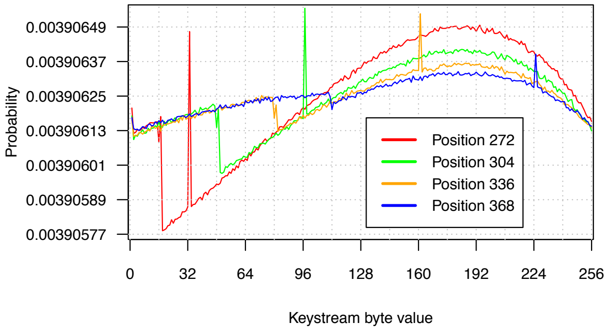


Figure 6: Single-byte biases beyond position 256.

$\Pr[Z_1 = Z_2 = 0]$ found by Isobe et al. Bias B and D are positive. We also discovered the following three biases:

$$\Pr[Z_1 = Z_3] = 2^{-8}(1 - 2^{-9.617}) \tag{3}$$
$$\Pr[Z_1 = Z_4] = 2^{-8}(1 + 2^{-8.590}) \tag{4}$$
$$\Pr[Z_2 = Z_4] = 2^{-8}(1 - 2^{-9.622}) \tag{5}$$

Note that all either involve an equality with Z_1 or Z_2.

3.3.3 Single-Byte Biases

We analysed single-byte biases by aggregating the `consec512` dataset, and by generating additional statistics specifically from single-byte probabilities. The aggregation corresponds to calculating

$$\Pr[Z_r = k] = \sum_{y=0}^{255} \Pr[Z_r = k \wedge Z_{r+1} = y] \tag{6}$$

We ended up with 2^{47} keys used to estimate single-byte probabilities. For all initial 513 bytes we could reject the hypothesis that they are uniformly distributed. In other words, all initial 513 bytes are biased. Figure 6 shows the probability distribution for some positions. Manual inspection of the distributions revealed a significant bias towards $Z_{256+k \cdot 16} = k \cdot 32$ for $1 \leq k \leq 7$. These are likely key-length dependent biases. Following [26] we conjecture there are single-byte biases even beyond these positions, albeit less strong.

3.4 New Long-Term Biases

To search for new long-term biases we created a variant of the `first16` dataset. It estimates

$$\Pr[Z_{256w+a} = x \wedge Z_{256w+b} = y] \tag{7}$$

for $0 \leq a \leq 16$, $0 \leq b < 256$, $0 \leq x, y < 256$, and $w \geq 4$. It is generated using 2^{12} RC4 keys, where each key was used to generate 2^{40} keystream bytes. This took roughly 8 CPU years. The condition on w means we always dropped the initial 1023 keystream bytes. Using this dataset we can detect biases whose periodicity is a proper divisor of 256 (e.g., it detected all Fluhrer-McGrew biases). Our new short-term biases were not present in this dataset, indicating they indeed only occur in the initial keystream bytes, at least with the probabilities we listed. We did find the new long-term bias

$$\Pr[(Z_{w256}, Z_{w256+2}) = (128, 0)] = 2^{-16}(1 + 2^{-8}) \quad (8)$$

for $w \geq 1$. Surprisingly this was not discovered earlier, since a bias towards $(0,0)$ at these positions was already known [38]. We also specifically searched for biases of the form $\Pr[Z_r = Z_{r'}]$ by aggregating our dataset. This revealed that many bytes are dependent on each other. That is, we detected several long-term biases of the form

$$\Pr[Z_{256w+a} = Z_{256w+b}] \approx 2^{-8}(2 \pm 2^{-16}) \quad (9)$$

Due to the small relative bias of 2^{-16}, these are difficult to reliably detect. That is, the pattern where these biases occur, and when their relative bias is positive or negative, is not yet clear. We consider it an interesting future research direction to (precisely and reliably) detect all keystream bytes which are dependent in this manner.

4 Plaintext Recovery

We will design plaintext recovery techniques for usage in two areas: decrypting TKIP packets and HTTPS cookies. In other scenarios, variants of our methods can be used.

4.1 Calculating Likelihood Estimates

Our goal is to convert a sequence of ciphertexts $\mathcal{C}$ into predictions about the plaintext. This is done by exploiting biases in the keystream distributions $p_k = \Pr[Z_r = k]$. These can be obtained by following the steps in Sect. 3.2. All biases in p_k are used to calculate the likelihood that a plaintext byte equals a certain value μ. To accomplish this, we rely on the likelihood calculations of AlFardan et al. [2]. Their idea is to calculate, for each plaintext value μ, the (induced) keystream distributions required to witness the captured ciphertexts. The closer this matches the real keystream distributions p_k, the more likely we have the correct plaintext byte. Assuming a fixed position r for simplicity, the induced keystream distributions are defined by the vector $N^\mu = (N_0^\mu, \ldots, N_{255}^\mu)$. Each N_k^μ represents the number of times the keystream byte was equal to k, assuming the plaintext byte was μ:

$$N_k^\mu = |\{C \in \mathcal{C} \mid C = k \oplus \mu\}| \quad (10)$$

Note that the vectors N^μ and $N^{\mu'}$ are permutations of each other. Based on the real keystream probabilities p_k we calculate the likelihood that this induced distribution would occur in practice. This is modelled using a multinomial distribution with the number of trails equal to $|\mathcal{C}|$, and the categories being the 256 possible keystream byte values. Since we want the probability of this *sequence* of keystream bytes we get [30]:

$$\Pr[\mathcal{C} \mid P = \mu] = \prod_{k \in \{0,\ldots,255\}} (p_k)^{N_k^\mu} \quad (11)$$

Using Bayes' theorem we can convert this into the likelihood λ_μ that the plaintext byte is μ:

$$\lambda_\mu = \Pr[P = \mu \mid \mathcal{C}] \sim \Pr[\mathcal{C} \mid P = \mu] \quad (12)$$

For our purposes we can treat this as an equality [2]. The most likely plaintext byte μ is the one that maximises λ_μ. This was extended to a pair of dependent keystream bytes in the obvious way:

$$\lambda_{\mu_1,\mu_2} = \prod_{k_1,k_2 \in \{0,\ldots,255\}} (p_{k_1,k_2})^{N_{k_1,k_2}^{\mu_1,\mu_2}} \quad (13)$$

We found this formula can be optimized if most keystream values k_1 and k_2 are independent and uniform. More precisely, let us assume that all keystream value pairs in the set $\mathcal{I}$ are independent and uniform:

$$\forall (k_1,k_2) \in \mathcal{I} \colon\ p_{k_1,k_2} = p_{k_1} \cdot p_{k_2} = u \quad (14)$$

where u represents the probability of an unbiased double-byte keystream value. Then we rewrite formula 13 to:

$$\lambda_{\mu_1,\mu_2} = (u)^{M^{\mu_1,\mu_2}} \cdot \prod_{k_1,k_2 \in \mathcal{I}^c} (p_{k_1,k_2})^{N_{k_1,k_2}^{\mu_1,\mu_2}} \quad (15)$$

where

$$M^{\mu_1,\mu_2} = \sum_{k_1,k_2 \in \mathcal{I}} N_{k_1,k_2}^{\mu_1,\mu_2} = |\mathcal{C}| - \sum_{k_1,k_2 \in \mathcal{I}^c} N_{k_1,k_2}^{\mu_1,\mu_2} \quad (16)$$

and with $\mathcal{I}^c$ the set of dependent keystream values. If the set $\mathcal{I}^c$ is small, this results in a lower time-complexity. For example, when applied to the long-term keystream setting over Fluhrer-McGrew biases, roughly 2^{19} operations are required to calculate all likelihood estimates, instead of 2^{32}. A similar (though less drastic) optimization can also be made when single-byte biases are present.

4.2 Likelihoods From Mantin's Bias

We now show how to compute a double-byte plaintext likelihood using Mantin's *ABSAB* bias. More formally, we want to compute the likelihood λ_{μ_1,μ_2} that the plaintext bytes at fixed positions r and $r+1$ are μ_1 and μ_2, respectively. To accomplish this we abuse surrounding known plaintext. Our main idea is to first calculate the likelihood of the *differential* between the known and unknown plaintext. We define the differential $\widehat{Z}_r^g$ as:

$$\widehat{Z}_r^g = (Z_r \oplus Z_{r+2+g}, Z_{r+1} \oplus Z_{r+3+g}) \tag{17}$$

Similarly we use $\widehat{C}_r^g$ and $\widehat{P}_r^g$ to denote the differential over ciphertext and plaintext bytes, respectively. The *ABSAB* bias can then be written as:

$$\Pr[\widehat{Z}_r^g = (0,0)] = 2^{-16}(1 + 2^{-8} \mathrm{e}^{\frac{-4-8g}{256}}) = \alpha(g) \tag{18}$$

When XORing both sides of $\widehat{Z}_r^g = (0,0)$ with $\widehat{P}_r^g$ we get

$$\Pr[\widehat{C}_r^g = \widehat{P}_r^g] = \alpha(g) \tag{19}$$

Hence Mantin's bias implies that the ciphertext differential is biased towards the plaintext differential. We use this to calculate the likelihood $\lambda_{\widehat{\mu}}$ of a differential $\widehat{\mu}$. For ease of notation we assume a fixed position r and a fixed *ABSAB* gap of g. Let $\widehat{\mathcal{C}}$ be the sequence of captured ciphertext differentials, and μ_1' and μ_2' the known plaintext bytes at positions $r+2+g$ and $r+3+g$, respectively. Similar to our previous likelihood estimates, we calculate the probability of witnessing the ciphertext differentials $\widehat{\mathcal{C}}$ assuming the plaintext differential is $\widehat{\mu}$:

$$\Pr[\widehat{\mathcal{C}} \mid \widehat{P} = \widehat{\mu}] = \prod_{\widehat{k} \in \{0,\ldots,255\}^2} \Pr[\widehat{Z} = \widehat{k}]^{N_{\widehat{k}}^{\widehat{\mu}}} \tag{20}$$

where

$$N_{\widehat{k}}^{\widehat{\mu}} = \left| \left\{ \widehat{C} \in \widehat{\mathcal{C}} \mid \widehat{C} = \widehat{k} \oplus \widehat{\mu} \right\} \right| \tag{21}$$

Using this notation we see that this is indeed identical to an ordinary likelihood estimation. Using Bayes' theorem we get $\lambda_{\widehat{\mu}} = \Pr[\widehat{\mathcal{C}} \mid \widehat{P} = \widehat{\mu}]$. Since only one differential pair is biased, we can apply and simplify formula 15:

$$\lambda_{\widehat{\mu}} = (1 - \alpha(g))^{|\mathcal{C}| - |\widehat{\mu}|} \cdot \alpha(g)^{|\widehat{\mu}|} \tag{22}$$

where we slightly abuse notation by defining $|\widehat{\mu}|$ as

$$|\widehat{\mu}| = \left| \left\{ \widehat{C} \in \widehat{\mathcal{C}} \mid \widehat{C} = \widehat{\mu} \right\} \right| \tag{23}$$

Finally we apply our knowledge of the known plaintext bytes to get our desired likelihood estimate:

$$\lambda_{\mu_1,\mu_2} = \lambda_{\widehat{\mu} \oplus (\mu_1', \mu_2')} \tag{24}$$

To estimate at which gap size the *ABSAB* bias is still detectable, we generated 2^{48} blocks of 512 keystream bytes. Based on this we empirically confirmed Mantin's *ABSAB* bias up to gap sizes of at least 135 bytes. The theoretical estimate in formula 1 slightly underestimates the true empirical bias. In our attacks we use a maximum gap size of 128.

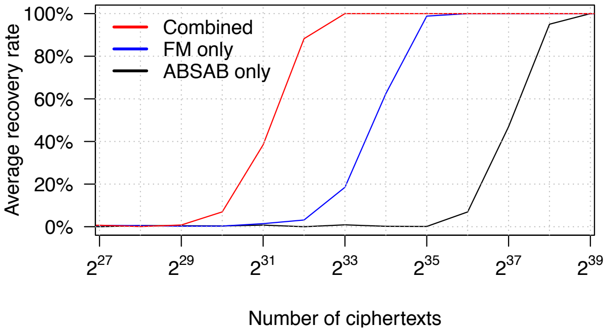


Figure 7: Average success rate of decrypting two bytes using: (1) one *ABSAB* bias; (2) Fluhrer-McGrew (FM) biases; and (3) combination of FM biases with 258 *ABSAB* biases. Results based on 2048 simulations each.

4.3 Combining Likelihood Estimates

Our goal is to combine multiple types of biases in a likelihood calculation. Unfortunately, if the biases cover overlapping positions, it quickly becomes infeasible to perform a single likelihood estimation over all bytes. In the worst case, the calculation cannot be optimized by relying on independent biases. Hence, a likelihood estimate over n keystream positions would have a time complexity of $\mathcal{O}(2^{2 \cdot 8 \cdot n})$. To overcome this problem, we perform and combine multiple separate likelihood estimates.

We will combine multiple types of biases by multiplying their individual likelihood estimates. For example, let λ'_{μ_1,μ_2} be the likelihood of plaintext bytes μ_1 and μ_2 based on the Fluhrer-McGrew biases. Similarly, let λ'_{g,μ_1,μ_2} be likelihoods derived from *ABSAB* biases of gap g. Then their combination is straightforward:

$$\lambda_{\mu_1,\mu_2} = \lambda'_{\mu_1,\mu_2} \cdot \prod_g \lambda'_{g,\mu_1,\mu_2} \tag{25}$$

While this method may not be optimal when combining likelihoods of dependent bytes, it does appear to be a general and powerful method. An open problem is determining which biases can be combined under a single likelihood calculation, while keeping computational requirements acceptable. Likelihoods based on other biases, e.g., Sen Gupta's and our new long-term biases, can be added as another factor (though some care is needed so positions properly overlap).

To verify the effectiveness of this approach, we performed simulations where we attempt to decrypt two bytes using one double-byte likelihood estimate. First this is done using only the Fluhrer-McGrew biases, and using only one *ABSAB* bias. Then we combine $2 \cdot 129$ *ABSAB* biases and the Fluhrer-McGrew biases, where we use the method from Sect. 4.2 to derive likelihoods from *ABSAB* biases. We assume the unknown bytes are surrounded at both sides by known plaintext, and use a maximum *ABSAB* gap of 128 bytes. Figure 7 shows the results of this experiment. Notice that a single *ABSAB* bias is weaker than using all Fluhrer-McGrew biases at a specific position. However, combining several *ABSAB* biases clearly results in a major improvement. We conclude that our approach to combine biases significantly reduces the required number of ciphertexts.

4.4 List of Plaintext Candidates

In practice it is useful to have a list of plaintext candidates in decreasing likelihood. For example, by traversing this list we could attempt to brute-force keys, passwords, cookies, etc. (see Sect. 6). In other situations the plaintext may have a rigid structure allowing the removal of candidates (see Sect. 5). We will generate a list of plaintext candidates in decreasing likelihood, when given either single-byte or double-byte likelihood estimates.

First we show how to construct a candidate list when given single-byte plaintext likelihoods. While it is trivial to generate the two most likely candidates, beyond this point the computation becomes more tedious. Our solution is to incrementally compute the N most likely candidates based on their length. That is, we first compute the N most likely candidates of length 1, then of length 2, and so on. Algorithm 1 gives a high-level implementation of this idea. Variable $P_r[i]$ denotes the i-th most likely plaintext of length r, having a likelihood of $E_r[i]$. The two `min` operations are needed because in the initial loops we are not yet be able to generate N candidates, i.e., there only exist 256^r plaintexts of length r. Picking the μ' which maximizes $pr(\mu')$ can be done efficiently using a priority queue. In practice, only the latest two versions of lists E and P need to be stored. To better maintain numeric stability, and to make the computation more efficient, we perform calculations using the logarithm of the likelihoods. We implemented Algorithm 1 and report on its performance in Sect. 5, where we use it to attack a wireless network protected by WPA-TKIP.

To generate a list of candidates from double-byte likelihoods, we first show how to model the likelihoods as a hidden Markov model (HMM). This allows us to present a more intuitive version of our algorithm, and refer to the extensive research in this area if more efficient implementations are needed. The algorithm we present can be seen as a combination of the classical Viterbi algorithm, and Algorithm 1. Even though it is not the most optimal one, it still proved sufficient to significantly improve plaintext recovery (see Sect. 6). For an introduction to HMMs we refer the reader to [35]. Essentially an HMM models a system where the internal states are not observable, and after each state transition, output is (probabilistically) produced dependent on its new state.

Algorithm 1: Generate plaintext candidates in decreasing likelihood using single-byte estimates.

```
Input: $L$ : Length of the unknown plaintext
       $\lambda_{1 \leq r \leq L,\, 0 \leq \mu \leq 255}$: single-byte likelihoods
       $N$: Number of candidates to generate
Returns: List of candidates in decreasing likelihood
$P_0[1] \leftarrow \varepsilon$
$E_0[1] \leftarrow 0$
for $r = 1$ to $L$ do
    for $\mu = 0$ to $255$ do
        $pos(\mu) \leftarrow 1$
        $pr(\mu) \leftarrow E_{r-1}[1] + \log(\lambda_{r,\mu})$
    for $i = 1$ to $\min(N, 256^r)$ do
        $\mu \leftarrow \mu'$ which maximizes $pr(\mu')$
        $P_r[i] \leftarrow P_{r-1}[pos(\mu)] \parallel \mu$
        $E_r[i] \leftarrow E_{r-1}[pos(\mu)] + \log(\lambda_{r,\mu})$
        $pos(\mu) \leftarrow pos(\mu) + 1$
        $pr(\mu) \leftarrow E_{r-1}[pos(\mu)] + \log(\lambda_{r,\mu})$
        if $pos(\mu) > \min(N, 256^{r-1})$ then
            $pr(\mu) \leftarrow -\infty$
return $P_N$
```

We model the plaintext likelihood estimates as a first-order time-inhomogeneous HMM. The state space S of the HMM is defined by the set of possible plaintext values $\{0, \ldots, 255\}$. The byte positions are modelled using the time-dependent (i.e., inhomogeneous) state transition probabilities. Intuitively, the "current time" in the HMM corresponds to the current plaintext position. This means the transition probabilities for moving from one state to another, which normally depend on the current time, will now depend on the position of the byte. More formally:

$$\Pr[S_{t+1} = \mu_2 \mid S_t = \mu_1] \sim \lambda_{t,\mu_1,\mu_2} \qquad (26)$$

where t represents the time. For our purposes we can treat this as an equality. In an HMM it is assumed that its current state is not observable. This corresponds to the fact that we do not know the value of any plaintext bytes. In an HMM there is also some form of output which depends on the current state. In our setting a particular plaintext value leaks no observable (side-channel) information. This is modelled by always letting every state produce the same null output with probability one.

Using the above HMM model, finding the most likely plaintext reduces to finding the most likely state sequence. This is solved using the well-known Viterbi algorithm. Indeed, the algorithm presented by AlFardan et al. closely resembles the Viterbi algorithm [2]. Similarly, finding the N most likely plaintexts is the same as finding the N most likely state sequences. Hence any N-best variant of the Viterbi algorithm (also called list Viterbi

Algorithm 2: Generate plaintext candidates in decreasing likelihood using double-byte estimates.

```
Input: $L$ : Length of the unknown plaintext plus two
       $m_1$ and $m_L$: known first and last byte
       $\lambda_{1 \leq r < L,\, 0 \leq \mu_1, \mu_2 \leq 255}$: double-byte likelihoods
       $N$: Number of candidates to generate
Returns: List of candidates in decreasing likelihood

for $\mu_2 = 0$ to 255 do
    $E_2[\mu_2, 1] \leftarrow \log(\lambda_{1,m_1,\mu_2})$
    $P_2[\mu_2, 1] \leftarrow m_1 \parallel \mu_2$

for $r = 3$ to $L$ do
    for $\mu_2 = 0$ to 255 do
        for $\mu_1 = 0$ to 255 do
            $pos(\mu_1) \leftarrow 1$
            $pr(\mu_1) \leftarrow E_{r-1}[\mu_1, 1] + \log(\lambda_{r,\mu_1,\mu_2})$

        for $i = 1$ to $\min(N, 256^{r-1})$ do
            $\mu_1 \leftarrow \mu$ which maximizes $pr(\mu)$
            $P_r[\mu_2, i] \leftarrow P_{r-1}[\mu_1, pos(\mu_1)] \parallel \mu_2$
            $E_r[\mu_2, i] \leftarrow E_{r-1}[\mu_1, pos(\mu_1)] + \log(\lambda_{r,\mu_1,\mu_2})$

            $pos(\mu_1) \leftarrow pos(\mu_1) + 1$
            $pr(\mu_1) \leftarrow E_{r-1}[\mu_1, pos(\mu_1)] + \log(\lambda_{r,\mu_1,\mu_2})$

            if $pos(\mu_1) > \min(N, 256^{r-2})$ then
                $pr(\mu_1) \leftarrow -\infty$

return $P_N[m_L, :]$
```

algorithm) can be used, examples being [42, 36, 40, 28]. The simplest form of such an algorithm keeps track of the N best candidates ending in a particular value μ, and is shown in Algorithm 2. Similar to [2, 30] we assume the first byte m_1 and last byte m_L of the plaintext are known. During the last round of the outer for-loop, the loop over μ_2 has to be executed only for the value m_L. In Sect. 6 we use this algorithm to generate a list of cookies.

Algorithm 2 uses considerably more memory than Algorithm 1. This is because it has to store the N most likely candidates for each possible ending value μ. We remind the reader that our goal is not to present the most optimal algorithm. Instead, by showing how to model the problem as an HMM, we can rely on related work in this area for more efficient algorithms [42, 36, 40, 28]. Since an HMM can be modelled as a graph, all k-shortest path algorithms are also applicable [10]. Finally, we remark that even our simple variant sufficed to significantly improve plaintext recovery rates (see Sect. 6).

5 Attacking WPA-TKIP

We use our plaintext recovery techniques to decrypt a full packet. From this decrypted packet the MIC key can be derived, allowing an attacker to inject and decrypt packets. The attack takes only an hour to execute in practice.

5.1 Calculating Plaintext Likelihoods

We rely on the attack of Paterson et al. to compute plaintext likelihood estimates [31, 30]. They noticed that the first three bytes of the per-packet RC4 key are public. As explained in Sect. 2.2, the first three bytes are fully determined by the TKIP Sequence Counter (TSC). It was observed that this dependency causes strong TSC-dependent biases in the keystream [31, 15, 30], which can be used to improve the plaintext likelihood estimates. For each TSC value they calculated plaintext likelihoods based on empirical, per-TSC, keystream distributions. The resulting 256^2 likelihoods are combined by multiplying them over all TSC pairs. In a sense this is similar to combining multiple types of biases as done in Sect. 4.3, though here the different types of biases are known to be independent. We use the single-byte variant of the attack [30, §4.1] to obtain likelihoods $\lambda_{r,\mu}$ for every unknown byte r.

The downside of this attack is that it requires detailed per-TSC keystream statistics. Paterson at al. generated statistics for the first 512 bytes, which took 30 CPU years [30]. However, in our attack we only need these statistics for the first few keystream bytes. We used 2^{32} keys per TSC value to estimate the keystream distribution for the first 128 bytes. Using our distributed setup the generation of these statistics took 10 CPU years.

With our per-TSC keystream distributions we obtained similar results to that of Paterson et al. [31, 30]. By running simulations we confirmed that the odd byte positions [30], instead of the even ones [31], can be recovered with a higher probability than others. Similarly, the bytes at positions 49-51 and 63-67 are generally recovered with higher probability as well. Both observations will be used to optimize the attack in practice.

5.2 Injecting Identical Packets

We show how to fulfil the first requirement of a successful attack: the generation of identical packets. If the IP of the victim is know, and incoming connections towards it are not blocked, we can simply send identical packets to the victim. Otherwise we induce the victim into opening a TCP connection to an attacker-controlled server. This connection is then used to transmit identical packets to the victim. A straightforward way to accomplish this is by social engineering the victim into visiting a website hosted by the attacker. The browser will open a TCP connection with the server in order to load the website. However, we can also employ more sophisticated methods that have a broader target range. One such method is abusing the inclusion of (insecure) third-party resources on popular websites [27]. For example, an attacker can register a mistyped domain, accidentally used in a resource address (e.g., an image URL) on a popular website. Whenever the victim visits this website and loads the resource, a TCP connection is made to the server of the attacker. In [27] these types of vulnerabilities were found to be present on several popular websites. Additionally, any type of web vulnerability that can be abused to make a victim execute JavaScript can be utilised. In this sense, our requirements are more relaxed than those of the recent attacks on SSL and TLS, which *require* the ability to run JavaScript code in the victim's browser [9, 1, 2]. Another method is to hijack an existing TCP connection of the victim, which under certain conditions is possible without a man-in-the-middle position [17]. We conclude that, while there is no universal method to accomplish this, we can assume control over a TCP connection with the victim. Using this connection we inject identical packets by repeatedly retransmitting identical TCP packets, even if the victim is behind a firewall. Since retransmissions are valid TCP behaviour, this will work even if the victim is behind a firewall.

We now determine the optimal structure of the injected packet. A naive approach would be to use the shortest possible packet, meaning no TCP payload is included. Since the total size of the LLC/SNAP, IP, and TCP header is 48 bytes, the MIC and ICV would be located at position 49 up to and including 60 (see Fig. 2). At these locations 7 bytes are strongly biased. In contrast, if we use a TCP payload of 7 bytes, the MIC and ICV are located at position 56 up to and including 60. In this range 8 bytes are strongly biased, resulting in better plaintext likelihood estimates. Through simulations we confirmed that using a 7 byte payload increases the probability of successfully decrypting the MIC and ICV. In practice, adding 7 bytes of payload also makes the length of our injected packet unique. As a result we can easily identify and capture such packets. Given both these advantages, we use a TCP data packet containing 7 bytes of payload.

5.3 Decrypting a Complete Packet

Our goal is to decrypt the injected TCP packet, including its MIC and ICV fields. Note that all these TCP packets will be encrypted with a different RC4 key. For now we assume all fields in the IP and TCP packet are known, and will later show why we can safely make this assumption. Hence, only the 8-byte MIC and 4-byte ICV of the packet remain unknown. We use the per-TSC keystream statistics to compute single-byte plaintext likelihoods for all 12 bytes. However, this alone would give a very low success probability of simultaneously (correctly) decrypting all bytes. We solve this by realising that the TKIP ICV is a simple CRC checksum which we can easily verify ourselves. Hence we can detect bad candidates by inspecting their CRC checksum. We now generate a plaintext candidate list, and traverse it until we find a packet having a correct CRC. This drastically improves the probability of simultaneously decrypting all bytes. From the decrypted packet we can derive the TKIP MIC key [44], which can then be used to inject and decrypt arbitrary packets [48].

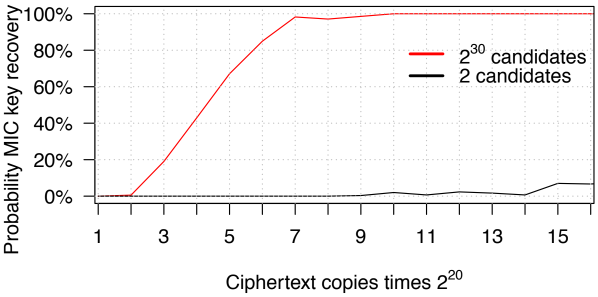


Figure 8: Success rate of obtaining the TKIP MIC key using nearly 2^{30} candidates, and using only the two best candidates. Results are based on 256 simulations each.

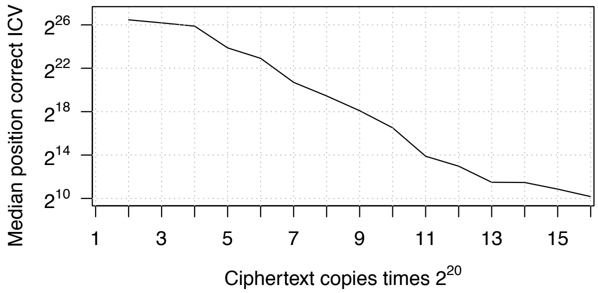


Figure 9: Median position of a candidate with a correct ICV with nearly 2^{30} candidates. Results are based on 256 simulations each.

Figure 8 shows the success rate of finding a packet with a good ICV and deriving the correct MIC key. For comparison, it also includes the success rates had we only used the two most likely candidates. Figure 9 shows the median position of the first candidate with a correct ICV. We plot the median instead of average to lower influence of outliers, i.e., at times the correct candidate was unexpectedly far (or early) in the candidate list.

The question that remains how to determine the contents of the unknown fields in the IP and TCP packet. More precisely, the unknown fields are the internal IP and port of the client, and the IP time-to-live (TTL) field. One observation makes this clear: both the IP and TCP header contain checksums. Therefore, we can apply exactly the same technique (i.e., candidate generation and pruning) to derive the values of these fields with high success rates. This can be done independently of each other, and independently of decrypting the MIC and ICV.

Another method to obtain the internal IP is to rely on HTML5 features. If the initial TCP connection is created by a browser, we can first send JavaScript code to obtain the internal IP of the victim using WebRTC [37]. We also noticed that our NAT gateway generally did not modify the source port used by the victim. Consequently we can simply read this value at the server. The TTL field can also be determined without relying on the IP checksum. Using a `traceroute` command we count the number of hops between the server and the client, allowing us to derive the TTL value at the victim.

5.4 Empirical Evaluation

To test the plaintext recovery phase of our attack we created a tool that parses a raw `pcap` file containing the captured Wi-Fi packets. It searches for the injected packets, extracts the ciphertext statistics, calculates plaintext likelihoods, and searches for a candidate with a correct ICV. From this candidate, i.e., decrypted injected packet, we derive the MIC key.

For the ciphertext generation phase we used an OpenVZ VPS as malicious server. The incoming TCP connection from the victim is handled using a custom tool written in Scapy. It relies on a patched version of Tcpreplay to rapidly inject the identical TCP packets. The victim machine is a Latitude E6500 and is connected to an Asus RT-N10 router running Tomato 1.28. The victim opens a TCP connection to the malicious server by visiting a website hosted on it. For the attacker we used a Compaq 8510p with an AWUS036nha to capture the wireless traffic. Under this setup we were able to generate roughly 2500 packets per second. This number was reached even when the victim was actively browsing YouTube videos. Thanks to the 7-byte payload, we uniquely detected the injected packet in all experiments without any false positives.

We ran several test where we generated and captured traffic for (slightly more) than one hour. This amounted to, on average, capturing $9.5 \cdot 2^{20}$ different encryptions of the packet being injected. Retransmissions were filtered based on the TSC of the packet. In nearly all cases we successfully decrypted the packet and derived the MIC key. Recall from Sect. 2.2 that this MIC key is valid as long as the victim does not renew its PTK, and that it can be used to inject and decrypt packets from the AP to the victim. For one capture our tool found a packet with a correct ICV, but this candidate did not correspond to the actual plaintext. While our current evaluation is limited in the number of captures performed, it shows the attack is practically feasible, with overall success probabilities appearing to agree with the simulated results of Fig. 8.

Listing 3: Manipulated HTTP request, with known plaintext surrounding the cookie at both sides.

```
GET / HTTP/1.1
Host: site.com
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:32.0)
     Gecko/20100101 Firefox/32.0
Accept: text/html,application/xhtml+xml,application/
     xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Cookie: auth=XXXXXXXXXXXXXXXX; injected1=known1;
     injected2=knownplaintext2; ...
```

6 Decrypting HTTPS Cookies

We inject known data around a cookie, enabling use of the *ABSAB* biases. We then show that a HTTPS cookie can be brute-forced using only 75 hours of ciphertext.

6.1 Injecting Known Plaintext

We want to be able to predict the position of the targeted cookie in the encrypted HTTP requests, and surround it with known plaintext. To fix ideas, we do this for the secure `auth` cookie sent to `https://site.com`. Similar to previous attacks on SSL and TLS, we assume the attacker is able to execute JavaScript code in the victim's browser [9, 1, 2]. In our case, this means an active man-in-the-middle (MiTM) position is used, where plaintext HTTP channels can be manipulated. Our first realisation is that an attacker can predict the length and content of HTTP headers preceding the `Cookie` field. By monitoring plaintext HTTP requests, these headers can be sniffed. If the targeted `auth` cookie is the first value in the `Cookie` header, this implies we know its position in the HTTP request. Hence, our goal is to have a layout as shown in Listing 3. Here the targeted cookie is the first value in the `Cookie` header, preceded by known headers, and followed by attacker injected cookies.

To obtain the layout in Listing 3 we use our MiTM position to redirect the victim to `http://site.com`, i.e., to the target website over an insecure HTTP channel. If the target website uses HTTP Strict Transport Security (HSTS), but does not use the `includeSubDomains` attribute, this is still possible by redirecting the victim to a (fake) subdomain [6]. Since few websites use HSTS, and even fewer use it properly [47], this redirection will likely succeed. Against old browsers HSTS can even be bypassed completely [6, 5, 41]. Since secure cookies guarantee only confidentiality but not integrity, the insecure HTTP channel can be used to overwrite, remove, or inject secure cookies [3, 4.1.2.5]. This allows us to remove all cookies except the `auth` cookie, pushing it to the front of the list. After this we can inject cookies that will be included after the `auth` cookie. An example of a HTTP(S) request manipulated in this manner is shown in Listing 3. Here the secure `auth` cookie is surrounded by known plaintext at both sides. This allows us to use Mantin's *ABSAB* bias when calculating plaintext likelihoods.

6.2 Brute-Forcing The Cookie

In contrast to passwords, many websites do not protect against brute-forcing cookies. One reason for this is that the password of an average user has a much lower entropy than a random cookie. Hence it makes sense to brute-force a password, but not a cookie: the chance of successfully brute-forcing a (properly generated) cookie is close to zero. However, if RC4 can be used to connect to the web server, our candidate generation algorithm voids this assumption. We can traverse the plaintext candidate list in an attempt to brute-force the cookie.

Since we are targeting a cookie, we can exclude certain plaintext values. As RFC 6265 states, a cookie value can consists of at most 90 unique characters [3, §4.1.1]. A similar though less general observation was already made by AlFardan et al. [2]. Our observation allows us to give a tighter bound on the required number of ciphertexts to decrypt a cookie, even in the general case. In practice, executing the attack with a reduced character set is done by modifying Algorithm 2 so the for-loops over μ_1 and μ_2 only loop over allowed characters.

Figure 10 shows the success rate of brute-forcing a 16-character cookie using at most 2^{23} attempts. For comparison, we also include the probability of decrypting the cookie if only the most likely plaintext was used. This also allows for an easier comparison with the work for AlFardan et al. [2]. Note that they only use the Fluhrer-McGrew biases, whereas we combine serveral *ABSAB* biases together with the Fluhrer-McGrew biases. We conclude that our brute-force approach, as well as the inclusion of the *ABSAB* biases, significantly improves success rates. Even when using only 2^{23} brute-force attempts, success rates of more than 94% are obtained once $9 \cdot 2^{27}$ encryptions of the cookie have been captured. We conjecture that generating more candidates will further increase success rates.

6.3 Empirical Evaluation

The main requirement of our attack is being able to collect sufficiently many encryptions of the cookie, i.e., having many ciphertexts. We fulfil this requirement by forcing the victim to generate a large number of HTTPS requests. As in previous attacks on TLS [9, 1, 2], we accomplish this by assuming the attacker is able to execute JavaScript in the browser of the victim. For example, when performing a man-in-the-middle attack, we can inject JavaScript into any plaintext HTTP connection. We then use `XMLHttpRequest` objects to issue Cross-Origin Requests to the targeted website. The browser will automatically add the secure cookie to these (encrypted) requests. Due to the same-origin policy we cannot read the replies, but this poses no problem, we only require that the cookie is included in the request. The requests are sent inside HTML5 WebWorkers. Essentially this means our JavaScript code will run in the background of the browser, and any open page(s) stay responsive. We use GET requests, and carefully craft the values of our injected cookies so the targeted `auth` cookie is always at a fixed position in the keystream (modulo 256). Recall that this alignment is required to make optimal use of the Fluhrer-McGrew biases. An attacker can learn the required amount of padding by first letting the client make a request without padding. Since RC4 is a stream cipher, and no padding is added by the TLS protocol, an attack can easily observe the length of this request. Based on this information it is trivial to derive the required amount of padding.

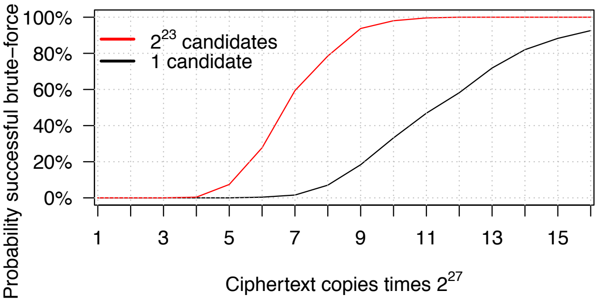


Figure 10: Success rate of brute-forcing a 16-byte cookie using roughly 2^{23} candidates, and only the most likely candidate, dependent on the number of collected ciphertexts. Results based on 256 simulations each.

To test our attack in practice we implemented a tool in C which monitors network traffic and collects the necessary ciphertext statistics. This requires reassembling the TCP and TLS streams, and then detecting the 512-byte (encrypted) HTTP requests. Similar to optimizing the generation of datasets as in Sect. 3.2, we cache several requests before updating the counters. We also created a tool to brute-force the cookie based on the generated candidate list. It uses persistent connections and HTTP pipelining [11, §6.3.2]. That is, it uses one connection to send multiple requests without waiting for each response.

In our experiments the victim uses a 3.1 GHz Intel Core i5-2400 CPU with 8 GB RAM running Windows 7. Internet Explorer 11 is used as the browser. For the server a 3.4 GHz Intel Core i7-3770 CPU with 8 GB RAM is used. We use nginx as the web server, and configured RC4-SHA1 with RSA as the only allowable cipher suite. This assures that RC4 is used in all tests. Both the server and client use an Intel 82579LM network card, with the link speed set to 100 Mbps. With an idle browser this setup resulted in an average of 4450 requests per second. When the victim was actively browsing YouTube videos this decreased to roughly 4100. To achieve such numbers, we found it's essential that the browser uses persistent connections to transmit the HTTP requests. Otherwise a new TCP and TLS handshake must be performed for every request, whose round-trip times would significantly slow down traffic generation. In practice this means the website must allow a `keep-alive` connection. While generating requests the browser remained responsive at all times. Finally, our custom tool was able to test more than 20000 cookies per second. To execute the attack with a success rate of 94% we need roughly $9 \cdot 2^{27}$ ciphertexts. With 4450 requests per seconds, this means we require 75 hours of data. Compared to the (more than) 2000 hours required by AlFardan et al. [2, §5.3.3] this is a significant improvement. We remark that, similar to the attack of AlFardan et al. [2], our attack also tolerates changes of the encryption keys. Hence, since cookies can have a long lifetime, the generation of this traffic can even be spread out over time. With 20000 brute-force attempts per second, all 2^{23} candidates for the cookie can be tested in less than 7 minutes.

We have executed the attack in practice, and successfully decrypted a 16-byte cookie. In our instance, capturing traffic for 52 hours already proved to be sufficient. At this point we collected $6.2 \cdot 2^{27}$ ciphertexts. After processing the ciphertexts, the cookie was found at position 46229 in the candidate list. This serves as a good example that, if the attacker has some luck, less ciphertexts are needed than our $9 \cdot 2^{27}$ estimate. These results push the attack from being on the verge of practicality, to feasible, though admittedly somewhat time-consuming.

7 Related Work

Due to its popularity, RC4 has undergone wide cryptanalysis. Particularly well known are the key recovery attacks that broke WEP [12, 50, 45, 44, 43]. Several other key-related biases and improvements of the original WEP attack have also been studied [21, 39, 32, 22].

We refer to Sect. 2.1 for an overview of various biases discovered in the keystream [25, 23, 38, 2, 20, 33, 13, 24, 38, 15, 31, 30]. In addition to these, the long-term bias $\Pr[Z_r = Z_{r+1} \mid 2 \cdot Z_r = i_r] = 2^{-8}(1+2^{-15})$ was discovered by Basu et al. [4]. While this resembles our new short-term bias $\Pr[Z_r = Z_{r+1}]$, in their analysis they assume the internal state $\mathcal{S}$ is a random permutation, which is true only after a few rounds of the PRGA. Isobe et al. searched for dependencies between initial keystream bytes by empirically estimating $\Pr[Z_r = y \wedge Z_{r-a} = x]$ for $0 \leq x, y \leq 255$, $2 \leq r \leq 256$, and $1 \leq a \leq 8$ [20]. They did not discover any new biases using their approach. Mironov modelled RC4 as a Markov chain and recommended to skip the initial $12 \cdot 256$ keystream bytes [26]. Paterson et al. generated keystream statistics over consecutive keystream bytes when using the TKIP key structure [30]. However, they did not report which (new) biases were present. Through empirical analysis, we show that biases between consecutive bytes are present even when using RC4 with random 128 bit keys.

The first practical attack on WPA-TKIP was found by Beck and Tews [44] and was later improved by other researchers [46, 16, 48, 49]. Recently several works studied the per-packet key construction both analytically [15] and through simulations [2, 31, 30]. For our attack we replicated part of the results of Paterson et al. [31, 30], and are the first to demonstrate this type of attack in practice. In [2] AlFardan et al. ran experiments where the two most likely plaintext candidates were generated using single-byte likelihoods [2]. However, they did not present an algorithm to return arbitrarily many candidates, nor extended this to double-byte likelihoods.

The SSL and TLS protocols have undergone wide scrutiny [9, 41, 7, 1, 2, 6]. Our work is based on the attack of AlFardan et al., who estimated that $13 \cdot 2^{30}$ ciphertexts are needed to recover a 16-byte cookie with high success rates [2]. We reduce this number to $9 \cdot 2^{27}$ using several techniques, the most prominent being usage of likelihoods based on Mantin's *ABSAB* bias [24]. Isobe et al. used Mantin's *ABSAB* bias, in combination with previously decrypted bytes, to decrypt bytes after position 257 [20]. However, they used a counting technique instead of Bayesian likelihoods. In [29] a guess-and-determine algorithm combines *ABSAB* and Fluhrer-McGrew biases, requiring roughly 2^{34} ciphertexts to decrypt an individual byte with high success rates.

8 Conclusion

While previous attacks against RC4 in TLS and WPA-TKIP were on the verge of practicality, our work pushes them towards being practical and feasible. After capturing $9 \cdot 2^{27}$ encryptions of a cookie sent over HTTPS, we can brute-force it with high success rates in negligible time. By running JavaScript code in the browser of the victim, we were able to execute the attack in practice within merely 52 hours. Additionally, by abusing RC4 biases, we successfully attacked a WPA-TKIP network within an hour. We consider it surprising this is possible using only known biases, and expect these types of attacks to further improve in the future. Based on these results, we strongly urge people to stop using RC4.

9 Acknowledgements

We thank Kenny Paterson for providing valuable feedback during the preparation of the camera-ready paper, and Tom Van Goethem for helping with the JavaScript traffic generation code.

This research is partially funded by the Research Fund KU Leuven. Mathy Vanhoef holds a Ph. D. fellowship of the Research Foundation - Flanders (FWO).

References

[1] N. J. Al Fardan and K. G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE Symposium on Security and Privacy*, 2013.

[2] N. J. AlFardan, D. J. Bernstein, K. G. Paterson, B. Poettering, and J. C. N. Schuldt. On the security of RC4 in TLS and WPA. In *USENIX Security Symposium*, 2013.

[3] A. Barth. HTTP state management mechanism. RFC 6265, 2011.

[4] R. Basu, S. Ganguly, S. Maitra, and G. Paul. A complete characterization of the evolution of RC4 pseudo random generation algorithm. *J. Mathematical Cryptology*, 2(3):257–289, 2008.

[5] D. Berbecaru and A. Lioy. On the robustness of applications based on the SSL and TLS security protocols. In *Public Key Infrastructure*, pages 248–264. Springer, 2007.

[6] K. Bhargavan, A. D. Lavaud, C. Fournet, A. Pironti, and P. Y. Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 98–113. IEEE, 2014.

[7] B. Canvel, A. P. Hiltgen, S. Vaudenay, and M. Vuagnoux. Password interception in a SSL/TLS channel. In *Advances in Cryptology (CRYPTO)*, 2003.

[8] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.2. RFC 5246, 2008.

[9] T. Duong and J. Rizzo. Here come the xor ninjas. In *Ekoparty Security Conference*, 2011.

[10] D. Eppstein. k-best enumeration. *arXiv preprint arXiv:1412.5075*, 2014.

[11] R. Fielding and J. Reschke. Hypertext transfer protocol (HTTP/1.1): Message syntax and routing. RFC 7230, 2014.

[12] S. Fluhrer, I. Mantin, and A. Shamir. Weaknesses in the key scheduling algorithm of RC4. In *Selected areas in cryptography*. Springer, 2001.

[13] S. R. Fluhrer and D. A. McGrew. Statistical analysis of the alleged RC4 keystream generator. In *FSE*, 2000.

[14] C. Fuchs and R. Kenett. A test for detecting outlying cells in the multinomial distribution and two-way contingency tables. *J. Am. Stat. Assoc.*, 75:395–398, 1980.

[15] S. S. Gupta, S. Maitra, W. Meier, G. Paul, and S. Sarkar. Dependence in IV-related bytes of RC4 key enhances vulnerabilities in WPA. Cryptology ePrint Archive, Report 2013/476, 2013. `http://eprint.iacr.org/`.

[16] F. M. Halvorsen, O. Haugen, M. Eian, and S. F. Mjølsnes. An improved attack on TKIP. In *14th Nordic Conference on Secure IT Systems*, NordSec '09, 2009.

[17] B. Harris and R. Hunt. Review: TCP/IP security threats and attack methods. *Computer Communications*, 22(10):885–897, 1999.

[18] ICSI. The ICSI certificate notary. Retrieved 22 Feb. 2015, from `http://notary.icsi.berkeley.edu`.

[19] IEEE Std 802.11-2012. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, 2012.

[20] T. Isobe, T. Ohigashi, Y. Watanabe, and M. Morii. Full plaintext recovery attack on broadcast RC4. In *FSE*, 2013.

[21] A. Klein. Attacks on the RC4 stream cipher. *Designs, Codes and Cryptography*, 48(3):269–286, 2008.

[22] S. Maitra and G. Paul. New form of permutation bias and secret key leakage in keystream bytes of RC4. In *Fast Software Encryption*, pages 253–269. Springer, 2008.

[23] S. Maitra, G. Paul, and S. S. Gupta. Attack on broadcast RC4 revisited. In *Fast Software Encryption*, 2011.

[24] I. Mantin. Predicting and distinguishing attacks on RC4 keystream generator. In *EUROCRYPT*, 2005.

[25] I. Mantin and A. Shamir. A practical attack on broadcast RC4. In *FSE*, 2001.

[26] I. Mironov. (Not so) random shuffles of RC4. In *CRYPTO*, 2002.

[27] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: Large-scale evaluation of remote JavaScript inclusions. In *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012.

[28] D. Nilsson and J. Goldberger. Sequentially finding the n-best list in hidden Markov models. In *International Joint Conferences on Artificial Intelligence*, 2001.

[29] T. Ohigashi, T. Isobe, Y. Watanabe, and M. Morii. Full plaintext recovery attacks on RC4 using multiple biases. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, 98(1):81–91, 2015.

[30] K. G. Paterson, B. Poettering, and J. C. Schuldt. Big bias hunting in amazonia: Large-scale computation and exploitation of RC4 biases. In *Advances in Cryptology — ASIACRYPT*, 2014.

[31] K. G. Paterson, J. C. N. Schuldt, and B. Poettering. Plaintext recovery attacks against WPA/TKIP. In *FSE*, 2014.

[32] G. Paul, S. Rathi, and S. Maitra. On non-negligible bias of the first output byte of RC4 towards the first three bytes of the secret key. *Designs, Codes and Cryptography*, 49(1-3):123–134, 2008.

[33] S. Paul and B. Preneel. A new weakness in the RC4 keystream generator and an approach to improve the security of the cipher. In *FSE*, 2004.

[34] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, 2014.

[35] L. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 1989.

[36] M. Roder and R. Hamzaoui. Fast tree-trellis list Viterbi decoding. *Communications, IEEE Transactions on*, 54(3):453–461, 2006.

[37] D. Roesler. STUN IP address requests for WebRTC. Retrieved 17 June 2015, from `https://github.com/diafygi/webrtc-ips`.

[38] S. Sen Gupta, S. Maitra, G. Paul, and S. Sarkar. (Non-)random sequences from (non-)random permutations - analysis of RC4 stream cipher. *Journal of Cryptology*, 27(1):67–108, 2014.

[39] P. Sepehrdad, S. Vaudenay, and M. Vuagnoux. Discovery and exploitation of new biases in RC4. In *Selected Areas in Cryptography*, pages 74–91. Springer, 2011.

[40] N. Seshadri and C.-E. W. Sundberg. List Viterbi decoding algorithms with applications. *IEEE Transactions on Communications*, 42(234):313–323, 1994.

[41] B. Smyth and A. Pironti. Truncating TLS connections to violate beliefs in web applications. In *WOOT'13: 7th USENIX Workshop on Offensive Technologies*, 2013.

[42] F. K. Soong and E.-F. Huang. A tree-trellis based fast search for finding the n-best sentence hypotheses in continuous speech recognition. In *Acoustics, Speech, and Signal Processing, 1991. ICASSP-91., 1991 International Conference on*, pages 705–708. IEEE, 1991.

[43] A. Stubblefield, J. Ioannidis, and A. D. Rubin. A key recovery attack on the 802.11b wired equivalent privacy protocol (WEP). *ACM Trans. Inf. Syst. Secur.*, 7(2), 2004.

[44] E. Tews and M. Beck. Practical attacks against WEP and WPA. In *Proceedings of the second ACM conference on Wireless network security*, WiSec '09, 2009.

[45] E. Tews, R.-P. Weinmann, and A. Pyshkin. Breaking 104 bit WEP in less than 60 seconds. In *Information Security Applications*, pages 188–202. Springer, 2007.

[46] Y. Todo, Y. Ozawa, T. Ohigashi, and M. Morii. Falsification attacks against WPA-TKIP in a realistic environment. *IEICE Transactions*, 95-D(2), 2012.

[47] T. Van Goethem, P. Chen, N. Nikiforakis, L. Desmet, and W. Joosen. Large-scale security analysis of the web: Challenges and findings. In *TRUST*, 2014.

[48] M. Vanhoef and F. Piessens. Practical verification of WPA-TKIP vulnerabilities. In *ASIACCS*, 2013.

[49] M. Vanhoef and F. Piessens. Advanced Wi-Fi attacks using commodity hardware. In *ACSAC*, 2014.

[50] S. Vaudenay and M. Vuagnoux. Passive–only key recovery attacks on RC4. In *Selected Areas in Cryptography*, pages 344–359. Springer, 2007.

Attacks Only Get Better: Password Recovery Attacks Against RC4 in TLS

Christina Garman
Johns Hopkins University
cgarman@cs.jhu.edu

Kenneth G. Paterson
Royal Holloway, University of London
kenny.paterson@rhul.ac.uk

Thyla van der Merwe
Royal Holloway, University of London
thyla.vandermerwe.2012@rhul.ac.uk

Abstract

Despite recent high-profile attacks on the RC4 algorithm in TLS, its usage is still running at about 30% of all TLS traffic. We provide new attacks against RC4 in TLS that are focussed on recovering user passwords, still the pre-eminent means of user authentication on the Internet today. Our new attacks use a generally applicable Bayesian inference approach to transform *a priori* information about passwords in combination with gathered ciphertexts into *a posteriori* likelihoods for passwords. We report on extensive simulations of the attacks. We also report on a "proof of concept" implementation of the attacks for a specific application layer protocol, namely BasicAuth. Our work validates the truism that attacks only get better with time: we obtain good success rates in recovering user passwords with 2^{26} encryptions, whereas the previous generation of attacks required around 2^{34} encryptions to recover an HTTP session cookie.

1 Introduction

TLS in all current versions allows RC4 to be used as its bulk encryption mechanism. Attacks on RC4 in TLS were first presented in 2013 in [2] (see also [13, 16]). Since then, usage of RC4 in TLS has declined, but it still accounted for around 30% of all TLS connections in March 2015.[1] Moreover, the majority of websites still support RC4[2] and a small proportion of websites *only* support RC4.[3]

We describe attacks recovering TLS-protected passwords whose ciphertext requirements are significantly reduced compared to those of [2]. Instead of the 2^{34} ciphertexts that were needed for recovering 16-byte, base64-encoded secure cookies in [2], our attacks now require around 2^{26} ciphertexts. We also describe a proof-of-concept implementation of these attacks against a specific application-layer protocol making use of passwords, namely BasicAuth.

1.1 Our Contributions

We obtain our improved attacks by revisiting the statistical methods of [2], refining, extending and applying them to the specific problem of recovering TLS-protected passwords. Passwords are a good target for our attacks because they are still very widely used on the Internet for providing user authentication in protocols like BasicAuth and IMAP, with TLS being used to prevent them being passively eavesdropped. To build effective attacks, we need to find and exploit systems in which users' passwords are automatically and repeatedly sent under the protection of TLS, so that sufficiently many ciphertexts can be gathered for our statistical analyses.

Bayesian analysis We present a formal Bayesian analysis that combines an *a priori* plaintext distribution with keystream distribution statistics to produce *a posteriori* plaintext likelihoods. This analysis formalises and extends the procedure followed in [2] for single-byte attacks. There, only keystream distribution statistics were used (specifically, biases in the individual bytes in the early portion of the RC4 keystream) and plaintexts were assumed to be uniformly distributed, while here we also exploit (partial) knowledge of the plaintext distribution to produce a more accurate estimate of the *a posteriori* likelihoods. This yields a procedure that is optimal (in the sense of yielding a maximum *a posteriori* estimate for the plaintext) if the plaintext distribution is known exactly.

[1] According to data obtained from the International Computer Science Institute (ICSI) Certificate Notary project, which collects statistics from live upstream SSL/TLS traffic in a passive manner; see http://notary.icsi.berkeley.edu.

[2] According to statistics obtained from SSL Pulse; see https://www.trustworthyinternet.org/ssl-pulse/.

[3] Amounting to 0.79% according to a January 2015 survey of about 400,000 of the Alexa top 1 million sites; see https://securitypitfalls.wordpress.com/2015/02/01/january-2015-scan-results/.

In the context of password recovery, an *estimate* for the *a priori* plaintext distribution can be empirically formed by using data from password breaches or by synthetically constructing password dictionaries. We will demonstrate, via simulations, that this Bayesian approach improves performance (measured in terms of success rate of plaintext recovery for a given number of ciphertexts) compared to the approach in [2].

Our Bayesian analysis concerns vectors of consecutive plaintext bytes, which is appropriate given passwords as the plaintext target. This, however, means that the keystream distribution statistics also need to be for vectors of consecutive keystream bytes. Such statistics do not exist in the prior literature on RC4, except for the Fluher-McGrew biases [10] (which supply the distributions for adjacent byte pairs far down the keystream). Fortunately, in the early bytes of the RC4 keystream, the single-byte biases are dominant enough that a simple product distribution can be used as a reasonable estimate for the distribution on vectors of keystream bytes. We also show how to build a more accurate approximation to the relevant keystream distributions using double-byte distributions. (Obtaining the double-byte distributions to a suitable degree of accuracy consumed roughly 4800 core-days of computation; for details see the full version [12].) This approximation is not only more accurate but also *necessary* when the target plaintext is located further down the stream, where the single-byte biases disappear and where double-byte biases become dominant. Indeed, our double-byte-based approximation to the keystream distribution on vectors can be used to smoothly interpolate between the region where single-byte biases dominate and where the double-byte biases come into play (which is exhibited as a fairly sharp transition around position 256 in the keystream).

In the end, what we obtain is a formal algorithm that estimates the likelihood of each password in a dictionary based on both the *a priori* password distribution and the observed ciphertexts. This formal algorithm is amenable to efficient implementation using either the single-byte based product distribution for keystreams or the double-byte-based approximation to the distribution on keystreams. The dominant terms in the running time for both of the resulting algorithms is $\mathcal{O}(nN)$ where n is the length of the target password and N is the size of the dictionary used in the attack.

An advantage of our new algorithms over the previous work in [2] is that they output a value for the likelihood of each password candidate, enabling these to be ranked and then tried in order of descending likelihood.

Note that our Bayesian approach is quite general and not limited to recovery of passwords, nor to RC4 – it can be applied whenever the plaintext distribution is approximately known, where the same plaintext is repeatedly encrypted, and where the stream cipher used for encryption has known biases in either single bytes or adjacent pairs of bytes.

Evaluation We evaluate and compare our password recovery algorithms through extensive simulations, exploring the relationships between the main parameters of our attack:

- The length n of the target password.
- The number S of available encryptions of the password.
- The starting position r of the password in the plaintext stream.
- The size N of the dictionary used in the attack, and the availability (or not) of an *a priori* password distribution for this dictionary.
- The number of attempts T made (meaning that our algorithm is considered successful if it ranks the correct password amongst the top T passwords, i.e. the T passwords with highest likelihoods as computed by the algorithm).
- Which of our two algorithms is used (the one computing the keystream statistics using the product distribution or the one using a double-byte-based approximation).
- Whether the passwords are Base64 encoded before being transmitted, or are sent as raw ASCII/Unicode.

Given the many possible parameter settings and the cost of performing simulations, we focus on comparing the performance with all but one or two parameters or variables being fixed in each instance.

Proofs of concept Our final contribution is to apply our attacks to specific and widely-deployed applications making use of passwords over TLS: BasicAuth and (in the full version [12]), IMAP. We introduce BasicAuth and describe a proof-of-concept implementation of our attacks against it, giving an indication of the practicality of our attacks. We do the same for IMAP in the full version [12].

For both applications, we have significant success rates with only $S = 2^{26}$ ciphertexts, in contrast to the roughly 2^{34} ciphertexts required in [2]. This is because we are able to force the target passwords into the first 256 bytes of plaintext, where the large single-byte biases in RC4 keystreams come into play. For example, with $S = 2^{26}$ ciphertexts, we would expect to recover a length 6 BasicAuth password with 44.5% success rate after $T = 5$ attempts; the rate rises to 64.4% if $T = 100$ attempts are made. In practice, many sites do not configure any limit on the number of BasicAuth attempts made by a client; moreover a study [5] showed that 84% of websites surveyed allowed for up to 100 password guesses (though these sites were not necessarily using BasicAuth as their authentication mechanism). As we will show, our result compares very favourably to the previous attacks and to random guessing of passwords without any reference to the ciphertexts.

However, there is a downside too: to make use of the early, single-byte biases in RC4 keystreams, we have to repeatedly cause TLS connections to be closed and new ones to be opened. Because of latency in the TLS Handshake Protocol, this leads to a significant slowdown in the wall clock running time of the attack; for $S = 2^{26}$, a latency of 100ms, and exploiting browsers' propensity to open multiple parallel connections, we estimate a running time of around 300 hours for the attack. This is still more than 6 times faster than the 2000 hours estimated in [2]. Furthermore, the attack's running time reduces proportionately to the latency of the TLS Handshake Protocol, so in environments where the client and server are close – for example in a LAN – the execution time could be a few tens of hours.

2 Further Background

2.1 The RC4 algorithm

Originally a proprietary stream cipher designed by Ron Rivest in 1987, RC4 is remarkably fast when implemented in software and has a very simple description. Details of the cipher were leaked in 1994 and the cipher has been subject to public analysis and study ever since.

RC4 allows for variable-length key sizes, anywhere from 40 to 256 bits, and consists of two algorithms, namely, a *key scheduling algorithm* (KSA) and a *pseudo-random generation algorithm* (PRGA). The KSA takes as input an l-byte key and produces the initial internal state $st_0 = (i, j, \mathscr{S})$ for the PRGA; $\mathscr{S}$ is the canonical representation of a permutation of the numbers from 0 to 255 where the permutation is a function of the l-byte key, and i and j are indices for $\mathscr{S}$. The KSA is specified in Algorithm 1 where K represents the l-byte key array and $\mathscr{S}$ the 256-byte state array. Given the internal state st_r, the PRGA will generate a keystream byte Z_{r+1} as specified in Algorithm 2.

2.2 Single-byte biases in the RC4 Keystream

RC4 has several cryptographic weaknesses, notably the existence of various biases in the RC4 keystream, see for example [2, 10, 14, 15, 19]. Large single-byte biases are prominent in the early postions of the RC4 keystream. Mantin and Shamir [15] observed the first of these biases, in Z_2 (the second byte of the RC4 keystream), and showed how to exploit it in what they called a *broadcast attack*, wherein the same plaintext is repeatedly encrypted under different keys. AlFardan *et al.* [2] performed large-scale computations to estimate these early biases, using 2^{45} keystreams to compute the single-byte keystream distributions in the first 256 output positions. They also provided a statistical approach to recovering plaintext bytes in the broadcast attack scenario, and explored its exploitation in TLS. Much of the new bias behaviour they observed was subsequently explained in [18]. Unfortunately, from an attacker's perspective, the single-byte biases die away very quickly beyond position 256 in the RC4 keystream. This means that they can only be used in attacks to extract plaintext bytes which are found close to the start of plaintext streams. This was a significant complicating factor in the attacks of [2], where, because of the behaviour of HTTP in modern browsers, the target HTTP secure cookies were not so located.

Algorithm 1: RC4 key scheduling (KSA)

```
input  : key $K$ of $l$ bytes
output : initial internal state $st_0$
begin
    for $i = 0$ to 255 do
        $\mathscr{S}[i] \leftarrow i$
    $j \leftarrow 0$
    for $i = 0$ to 255 do
        $j \leftarrow j + \mathscr{S}[i] + K[i \bmod l]$
        swap($\mathscr{S}[i], \mathscr{S}[j]$)
    $i, j \leftarrow 0$
    $st_0 \leftarrow (i, j, \mathscr{S})$
    return $st_0$
```

Algorithm 2: RC4 keystream generator (PRGA)

```
input  : internal state $st_r$
output : keystream byte $Z_{r+1}$
         updated internal state $st_{r+1}$
begin
    parse $(i, j, \mathscr{S}) \leftarrow st_r$
    $i \leftarrow i + 1$
    $j \leftarrow j + \mathscr{S}[i]$
    swap($\mathscr{S}[i], \mathscr{S}[j]$)
    $Z_{r+1} \leftarrow \mathscr{S}[\mathscr{S}[i] + \mathscr{S}[j]]$
    $st_{r+1} \leftarrow (i, j, \mathscr{S})$
    return $(Z_{r+1}, st_{r+1})$
```

2.3 Double-byte biases in the RC4 Keystream

Fluhrer and McGrew [10] showed that there are biases in adjacent bytes in RC4 keystreams, and that these so-called double-byte biases are persistent throughout the keystream. The presence of these long-term biases (and the absence of any other similarly-sized double-byte biases) was confirmed computationally in [2]. AlFardan *et al.* [2] also exploited these biases in their double-byte attack to recover HTTP secure cookies.

Because we wish to exploit double-byte biases in early portions of the RC4 keystream and because the analysis of [10] assumes the RC4 permutation $\mathscr{S}$ is uniformly random (which is not the case for early keystream bytes), we carried out extensive computations to estimate the initial double-byte keystream distributions: we used roughly 4800 core-days of computation to generate 2^{44} RC4 keystreams for random 128-bit RC4 keys (as used in TLS); we used these keystreams to estimate the double-byte keystream distributions for RC4 in the first 512 positions.

While the gross behaviour that we observed is dominated by products of the known single-byte biases in the first 256 positions and by the Fluhrer-McGrew biases in the later positions, we did observe some new and interesting double-byte biases. Since these are likely to be of independent interest to researchers working on RC4, we report in more detail on this aspect of our work in the full version [12].

2.4 RC4 and the TLS Record Protocol

We provide an overview of the TLS Record Protocol with RC4 selected as the method for encryption and direct the reader to [2, 6, 7, 8] for further details.

Application data to be protected by TLS, i.e, a sequence of bytes or a record R, is processed as follows: An 8-byte sequence number `SQN`, a 5-byte header `HDR` and R are concatenated to form the input to an HMAC function. We let T denote the resulting output of this function. In the case of RC4 encryption, the plaintext, $P = T||R$, is XORed byte-per-byte with the RC4 keystream. In other words,

$$C_r = P_r \oplus Z_r,$$

for the r^{th} bytes of the ciphertext, plaintext and RC4 keystream respectively (for $r = 1, 2, 3 \ldots$). The data that is transmitted has the form `HDR`$||C$, where C is the concatenation of the individual ciphertext bytes.

The RC4 algorithm is intialized in the standard way at the start of each TLS connection with a 128-bit encryption key. This key, K, is derived from the TLS master secret that is established during the TLS Handshake Protocol; K is either established via the the full TLS Handshake Protocol or TLS session resumption. The first few bytes to be protected by RC4 encryption is a `Finished` message of the TLS Handshake Protocol. We do not target this record in our attacks since this message is not constant over multiple sessions. The exact size of this message is important in dictating how far down the keystream our target plaintext will be located; in turn this determines whether or not it can be recovered using only single-byte biases. A common size is 36 bytes, but the exact size depends on the output size of the TLS PRF used in computing the `Finished` message and of the hash function used in the HMAC algorithm in the record protocol.

Decryption is the reverse of the process described above. As noted in [2], any error in decryption is treated as fatal – an error message is sent to the sender and all cryptographic material, including the RC4 key, is disposed of. This enables an active attacker to force the use of new encryption and MAC keys: the attacker can induce session termination, followed by a new session being established when the next message is sent over TLS, by simply modifying a TLS Record Protocol message. This could be used to ensure that the target plaintext in an attack is repeatedly sent under the protection of a fresh RC4 key. However, this approach is relatively expensive since it involves a rerun of the full TLS Handshake Protocol, involving multiple public key operations and, more importantly, the latency involved in an exchange of 4 messages (2 complete round-trips) on the wire. A better approach is to cause the TCP connection carrying the TLS traffic to close, either by injecting sequences of `FIN` and `ACK` messages in both directions, or by injecting a `RST` message in both directions. This causes the TLS *connection* to be terminated, but not the TLS *session* (assuming the session is marked as "resumable" which is typically the case). This behaviour is codified in [8, Section 7.2.1]. Now when the next message is sent over TLS, a TLS session resumption instance of the Handshake Protocol is executed to establish a fresh key for RC4. This avoids the expensive public key operations and reduces the TLS latency to 1 round-trip before application data can be sent. On large sites, session resumption is usually handled by making use of TLS session tickets [17] on the server-side.

2.5 Passwords

Text-based passwords are arguably the dominant mechanism for authenticating users to web-based services and computer systems. As is to be expected of user-selected secrets, passwords do not follow uniform distributions. Various password breaches of recent years, including the Adobe breach of 150 million records in 2013 and the RockYou leak of 32.6 million passwords in 2009, attest to this with passwords such as `123456` and `password` frequently being counted amongst the most popular.[4] For example, our own analysis of the RockYou password data set confirmed this: the number of unique passwords in the RockYou dataset is 14,344,391, meaning that (on average) each password was repeated 2.2 times, and we indeed found the most common password to be `123456` (accounting for about 0.9% of the entire data set). Our later simulations will make extensive use of the RockYou data set as an attack dictionary. A more-fine grained analysis of it can be found in [20]. We also make use of data from the Singles.org breach for generating our target passwords. Singles.org is a now-defunct Christian dating website that was breached in 2009; religiously-inspired passwords such as `jesus` and `angel` appear with high frequency in its 12,234 distinct entries, making its frequency distribution quite different from that of the RockYou set.

There is extensive literature regarding the reasons for poor password selection and usage, including [1, 9, 21, 22]. In [4], Bonneau formalised a number of different metrics for analysing password distributions and studied a corpus of 70M Yahoo! passwords (collected in a privacy-preserving manner). His work highlights the importance of careful validation of password guessing attacks, in particular, the problem of estimating attack complexities in the face of passwords that occur rarely – perhaps uniquely – in a data set, the so-called *hapax legomena* problem. The approach to validation that we adopt benefits from the analysis of [4], as explained further in Section 4.

3 Plaintext Recovery via Bayesian Analysis

In this section, we present a formal Bayesian analysis of plaintext recovery attacks in the broadcast setting for stream ciphers. We then apply this to the problem of extracting passwords, specialising the formal analysis and making it implementable in practice based only on the single-byte and double-byte keystream distributions.

3.1 Formal Bayesian Analysis

Suppose we have a candidate set of N plaintexts, denoted $\mathscr{X}$, with the *a priori* probability of an element $x \in \mathscr{X}$ being denoted p_x. We assume for simplicity that all the candidates consist of byte strings of the same length n. For example $\mathscr{X}$ might consist of all the passwords of a given length n from some breach data set, and then p_x can be computed as the relative frequency of x in the data set. If the frequency data is not available, then the uniform distribution on $\mathscr{X}$ can be assumed.

Next, suppose that a plaintext from $\mathscr{X}$ is encrypted S times, each time under independent, random keys using a stream cipher such as RC4. Suppose also that the first character of the plaintext always occurs in the same position r in the plaintext stream in each encryption. Let $c = (c_{ij})$ denote the $S \times n$ matrix of bytes in which row i, denoted $c^{(i)}$ for $0 \le i < S$, is a vector of n bytes corresponding to the values in positions $r, \ldots, r+n-1$ in ciphertext i. Let X be the random variable denoting the (unknown) value of the plaintext.

We wish to form a maximum a posteriori (MAP) estimate for X, given the observed data c and the *a priori* probability distribution p_x, that is, we wish to maximise $\Pr(X = x \mid C = c)$ where C is a random variable corresponding to the matrix of ciphertext bytes.

Using Bayes' theorem, we have

$$\Pr(X = x \mid C = c) = \Pr(C = c \mid X = x) \cdot \frac{\Pr(X = x)}{\Pr(C = c)}.$$

Here the term $\Pr(X = x)$ corresponds to the *a priori* distribution p_x on $\mathscr{X}$. The term $\Pr(C = c)$ is independent of the choice of x (as can be seen by writing $\Pr(C = c) = \sum_{x \in \mathscr{X}} \Pr(C = c \mid X = x) \cdot \Pr(X = x)$). Since we are only interested in maximising $\Pr(X = x \mid C = c)$, we ignore this term henceforth.

Now, since ciphertexts are formed by XORing keystreams z and plaintext x, we can write

$$\Pr(C = c \mid X = x) = \Pr(W = w)$$

where w is the $S \times n$ matrix formed by XORing each row of c with the vector x and W is a corresponding random variable. Then to maximise $\Pr(X = x \mid C = c)$, it suffices to maximise the value of

$$\Pr(X = x) \cdot \Pr(W = w)$$

over $x \in \mathscr{X}$. Let $w^{(i)}$ denote the i-th row of the matrix w, so $w^{(i)} = c^{(i)} \oplus x$. Then $w^{(i)}$ can be thought of as a vector of keystream bytes (coming from positions $r, \ldots, r+n-1$) induced by the candidate x, and we can write

$$\Pr(W = w) = \prod_{i=0}^{S-1} \Pr(Z = w^{(i)})$$

where, on the right-hand side of the above equation, Z denotes a random variable corresponding to a vector of bytes of length n starting from position r in the keystream. Writing $\mathscr{B} = \{\texttt{0x00}, \ldots, \texttt{0xFF}\}$ for the set of bytes, we can rewrite this as:

$$\Pr(W = w) = \prod_{z \in \mathscr{B}^n} \Pr(Z = z)^{N_{x,z}}$$

where the product is taken over all possible byte strings of length n and $N_{x,z}$ is defined as:

$$N_{x,z} = |\{i : z = c^{(i)} \oplus x\}_{0 \le i < S}|,$$

[4] A comprehensive list of data breaches, including password breaches, can be found at `http://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/`.

that is, $N_{x,z}$ counts the number of occurrences of vector z in the rows of the matrix formed by XORing each row of c with candidate x. Putting everything together, our objective is to compute for each candidate $x \in \mathscr{X}$ the value:

$$\Pr(X = x) \cdot \prod_{z \in \mathscr{B}^n} \Pr(Z = z)^{N_{x,z}}$$

and then to rank these values in order to determine the most likely candidate(s).

Notice that the expressions here involve terms $\Pr(Z = z)$ which are probabilities of occurrence for n consecutive bytes of keystream. Such estimates are not generally available in the literature, and for the values of n we are interested in (corresponding to putative password lengths), obtaining accurate estimates for them by sampling many keystreams would be computationally prohibitive. Moreover, the product $\prod_{z \in \mathscr{B}^n}$ involves 2^{8n} terms and is not amenable to calculation. Thus we must turn to approximate methods to make further progress.

Note also that taking $n = 1$ in the above analysis, we obtain exactly the same approach as was used in the single-byte attack in [2], except that we include the *a priori* probabilities $\Pr(X = x)$ whereas these were (implicitly) assumed to be uniform in [2].

3.2 Using a Product Distribution

Our task is to derive simplified ways of computing the expression

$$\Pr(X = x) \cdot \prod_{z \in \mathscr{B}^n} \Pr(Z = z)^{N_{x,z}}$$

and then apply these to produce efficient algorithms for computing (approximate) likelihoods of candidates $x \in \mathscr{X}$.

The simplest approach is to assume that the n bytes of the keystreams can be treated independently. For RC4, this is actually a very good approximation in the regime where single-byte biases dominate (that is, in the first 256 positions). Thus, writing $Z = (Z_r, \dots, Z_{r+n-1})$ and $z = (z_r, \dots, z_{r+n-1})$ (with the subscript r denoting the position of the first keystream byte of interest), we have:

$$\Pr(Z = z) \approx \prod_{j=0}^{n-1} \Pr(Z_{r+j} = z_{r+j}) = \prod_{j=0}^{n-1} p_{r+j,z}$$

where now the probabilities appearing on the right-hand side are single-byte keystream probabilities, as reported in [2] for example. Then writing $x = (x_0, \dots, x_{n-1})$ and rearranging terms, we obtain:

$$\prod_{z \in \mathscr{B}^n} \Pr(Z = z)^{N_{x,z}} \approx \prod_{j=0}^{n-1} \prod_{z \in \mathscr{B}} p_{r+j,z}^{N_{x_j,z,j}}$$

where $N_{y,z,j} = |\{i : z = c_{i,j} \oplus y\}_{0 \le i < S}|$ counts (now for single bytes instead of length n vectors of bytes) the number of occurrences of byte z in the column vector formed by XORing column j of c with a candidate byte y.

Notice that, as in [2], the counters $N_{y,z,j}$ for $y \in \mathscr{B}$ can all be computed efficiently by permuting the counters $N_{\texttt{0x00},z,j}$, these being simply counters for the number of occurrences of each byte value z in column j of the ciphertext matrix c.

In practice, it is more convenient to work with logarithms, converting products into sums, so that we evaluate for each candidate $x = (x_0, \dots, x_{n-1})$ an expression of the form

$$\gamma_x := \log(p_x) + \sum_{j=0}^{n-1} \sum_{z \in \mathscr{B}} N_{x_j,z,j} \log(p_{r+j,z}).$$

Given a large set of candidates $\mathscr{X}$, we can streamline the computation by first computing the counters $N_{y,z,j}$, then, for each possible byte value y, the value of the inner sum $\sum_{z \in \mathscr{B}} N_{y,z,j} \log(p_{r+j,z})$, and then reusing these individual values across all the relevant candidates x for which $x_j = y$. This reduces the evaluation of γ_x for a single candidate x to $n+1$ additions of real numbers.

The above procedure, including the various optimizations, is specified as an attack in Algorithm 3. We refer to it as our single-byte attack because of its reliance on the single-byte keystream probabilities $p_{r+j,z}$. It outputs a collection of approximate log likelihoods $\{\gamma_x : x \in \mathscr{X}\}$ for each candidate $x \in \mathscr{X}$. These can be further processed to extract, for example, the candidate with the highest score, or the top T candidates.

3.3 Double-byte-based Approximation

We continue to write $Z = (Z_r, \dots, Z_{r+n-1})$ and $z = (z_r, \dots, z_{r+n-1})$ and aim to find an approximation for $\Pr(Z = z)$ which lends itself to efficient computation of approximate log likelihoods as in our first algorithm. Now we rely on the double-byte keystream distribution, writing

$$p_{s,z_1,z_2} := \Pr((Z_s, Z_{s+1}) = (z_1, z_2)), \quad s \ge 1, k_1, k_2 \in \mathscr{B}$$

for the probabilities of observing bytes (z_1, z_2) in the RC4 keystream in positions $(s, s+1)$. We estimated these probabilities for r in the range $1 \le r \le 511$ using 2^{44} RC4 keystreams – for details, see the full version; for larger r, these are well approximated by the Fluhrer-McGrew biases [10] (as was verified in [2]).

We now make the Markovian assumption that, for each j,

$$\begin{aligned} &\Pr(Z_j = z_j \mid Z_{j-1} = z_{j-1} \wedge \dots \wedge Z_0 = z_0) \\ &\qquad \approx \Pr(Z_j = z_j \mid Z_{j-1} = z_{j-1}), \end{aligned}$$

```
Algorithm 3: Single-byte attack
input  : $c_{i,j} : 0 \le i < S, 0 \le j < n$ – array formed from $S$ independent encryptions of fixed $n$-byte candidate $X$
         $r$ – starting position of $X$ in plaintext stream
         $\mathcal{X}$ – collection of $N$ candidates
         $p_x$ – a priori probability of candidates $x \in \mathcal{X}$
         $p_{r+j,z}$ $(0 \le j < n, z \in \mathcal{B})$ – single-byte keystream distribution
output : $\{\gamma_x : x \in \mathcal{X}\}$ – set of (approximate) log likelihoods for candidates in $\mathcal{X}$
begin
    for $j = 0$ to $n-1$ do
        for $z =$ 0x00 to 0xFF do
            $N'_{z,j} \leftarrow 0$
    for $j = 0$ to $n-1$ do
        for $i = 0$ to $S-1$ do
            $N'_{c_{i,j},j} \leftarrow N'_{c_{i,j},j} + 1$
    for $j = 0$ to $n-1$ do
        for $y =$ 0x00 to 0xFF do
            for $z =$ 0x00 to 0xFF do
                $N_{y,z,j} \leftarrow N'_{z\oplus y,j}$
            $L_{y,j} = \sum_{z\in\mathcal{B}} N_{y,z,j}\log(p_{r+j,z})$,
    for $x = (x_0,\ldots,x_{n-1}) \in \mathcal{X}$ do
        $\gamma_x \leftarrow \log(p_x) + \sum_{j=0}^{n-1} L_{x_j,j}$
    return $\{\gamma_x : x \in \mathcal{X}\}$
```

meaning that byte j in the keystream can be modelled as depending only on the preceding byte and not on earlier bytes. We can write

$$\Pr(Z_j = z_j \mid Z_{j-1} = z_{j-1}) = \frac{\Pr(Z_j = z_j \wedge Z_{j-1} = z_{j-1})}{\Pr(Z_{j-1} = z_{j-1})}$$

where the numerator can then be replaced by p_{j-1,z_{j-1},z_j} and the denominator by $p_{j-1,z_{j-1}}$, a single-byte keystream probability. Then using an inductive argument and our assumption, we easily obtain:

$$\Pr(Z = z) \approx \frac{\prod_{j=0}^{n-2} p_{r+j,z_j,z_{j+1}}}{\prod_{j=1}^{n-2} p_{r+j,z_j}}$$

giving an approximate expression for our desired probability in terms of single-byte and double-byte probabilities. Notice that if we assume that the adjacent byte pairs are independent, then we have $p_{r+j,z_j,z_{j+1}} = p_{r+j,z_j} \cdot p_{r+j+1,z_{j+1}}$ and the above expression collapses down to the one we derived in the previous subsection.

For candidate x, we again write $x = (x_0,\ldots,x_{n-1})$ and rearranging terms, we obtain:

$$\prod_{z\in\mathcal{B}^n} \Pr(Z = z)^{N_{x,z}} \approx \frac{\prod_{j=0}^{n-2}\prod_{z_1\in\mathcal{B}}\prod_{z_2\in\mathcal{B}} p_{r+j,z_1,z_2}^{N_{x_j,x_{j+1},z_1,z_2,j}}}{\prod_{j=1}^{n-2}\prod_{z\in\mathcal{B}} p_{r+j,z}^{N_{x_j,z,r+j}}}$$

where $N_{y_1,y_1,z_1,z_2,j} = |\{i : z_1 = c_{i,j} \oplus y_1 \wedge z_2 = c_{i,j+1} \oplus y_2\}_{0\le i<S}|$ counts (now for consecutive pairs of bytes) the number of occurrences of bytes (z_1,z_2) in the pair of column vectors formed by XORing columns $(j, j+1)$ of c with candidate bytes (y_1,y_2) (and where $N_{x_j,z,r+j}$ is as in our previous algorithm).

Again, the counters $N_{y_1,y_2,z_1,z_2,j}$ for $y_1,y_2 \in \mathcal{B}$ can all be computed efficiently by permuting the counters $N_{\texttt{0x00},\texttt{0x00},z_1,z_2,j}$, these being simply counters for the number of occurrences of pairs of byte values (z_1,z_2) in column j and $j+1$ of the ciphertext matrix c. As before, we work with logarithms, so that we evaluate for each candidate $x = (x_0,\ldots,x_{n-1})$ an expression of the form

$$\gamma_x := \log(p_x) + \sum_{j=0}^{n-2}\sum_{z_1\in\mathcal{B}}\sum_{z_2\in\mathcal{B}} N_{x_j,x_{j+1},z_1,z_2,j}\log(p_{r+j,z_1,z_2}) - \sum_{j=1}^{n-2}\sum_{z\in\mathcal{B}} N_{x_j,z,r+j}\log(p_{r+j,z}).$$

With appropriate pre-computation of the terms $N_{y_1,y_2,z_1,z_2,j}\log(p_{r+j,z_1,z_2})$ and $N_{y,z,r+j}\log(p_{r+j,z})$ for all y_1,y_2 and all y, the computation for each candidate $x \in \mathcal{X}$ can be reduced to roughly $2n$ floating point additions. The pre-computation can be further reduced by computing the terms for only those pairs (y_1,y_2) actually arising in candidates in $\mathcal{X}$ in positions $(j, j+1)$. We use this further optimisation in our implementation.

The above procedure is specified as an attack in Algorithm 4. We refer to it as our double-byte attack because of its reliance on the double-byte keystream probabilities p_{s,z_1,z_2}. It again outputs a collection of approximate log likelihoods $\{\gamma_x : x \in \mathscr{X}\}$ for each candidate $x \in \mathscr{X}$, suitable for further processing. Note that for simplicity of presentation, it involves a quintuply-nested loop to compute the values $N_{y_1,y_2,z_1,z_2,j}$; these values should of course be directly computed from the $(n-1)\cdot 2^{16}$ pre-computed counters $N'_{c_{i,j},c_{i,j+1},j}$ in an in-line fashion using the formula $N_{y_1,y_2,z_1,z_2,j} = N'_{z_1\oplus y_1,z_2\oplus y_2,,j}$.

4 Simulation Results

4.1 Methodology

We performed extensive simulations of both of our attacks, varying the different parameters to evaluate their effects on success rates. We focus on the problem of password recovery, using the RockYou data set as an attack dictionary and the Singles.org data set as the set of target passwords. Except where noted, in each simulation, we performed 256 independent runs of the relevant attack. In each attack in a simulation, we select a password of some fixed length n from the Singles.org password data set according to the known *a priori* probability distribution for that data set, encrypt it S times in different starting positions r using random 128-bit keys for RC4, and then attempt to recover the password from the ciphertexts using the set of all passwords of length n from the entire RockYou data set (14 million passwords) as our candidate set $\mathscr{X}$. We declare success if the target password is found within the top T passwords suggested by the algorithm (according to the approximate likelihood measures γ_x). Our default settings, unless otherwise stated, are $n = 6$ and $T = 5$. Six is the most common password length in the data sets we encountered; $T = 5$ is an arbitrary choice, and we examine the effect of varying T in detail below. We try all values for r between 1 and $256-n+1$, where the single-byte biases dominate the behaviour of the RC4 keystreams. Typical values of S are 2^s where $s \in \{20,22,24,26,28\}$.

Using different data sets for the attack dictionary and the target set from which encrypted passwords are chosen is more realistic than using a single dictionary for both purposes, not least because in a real attack, the exact content and *a priori* distribution of the target set would not be known. This approach also avoids the problem of *hapax legomena* highlighted in [4]. However, this has the effect of limiting the success rates of our attacks to less than 100%, since there are highly likely passwords in the target set (such as `jesus`) that do not occur at all, or only have very low *a priori* probabilities in the attack dictionary, and conversely. Figure 1 compares the use of the RockYou password distribution to attack Singles.org passwords with the less realistic use of the RockYou password distribution to attack RockYou itself. It can be seen that, for the particular choice of attack parameters ($S = 2^{24}$, $n = 6$, $T = 5$, double-byte attack), the effect on success rate is not particularly large. However, for other attack parameters, as we will see below, we observe a maximum success rate of around 80% for our attacks, whereas we would achieve 100% success rates if we used RockYou against itself. The observed maximum success rate could be increased by augmenting the attack dictionary with synthetically generated, site-specific passwords and by removing RockYou-specific passwords from the attack dictionary. We leave the development and evaluation of these improvements to future work.

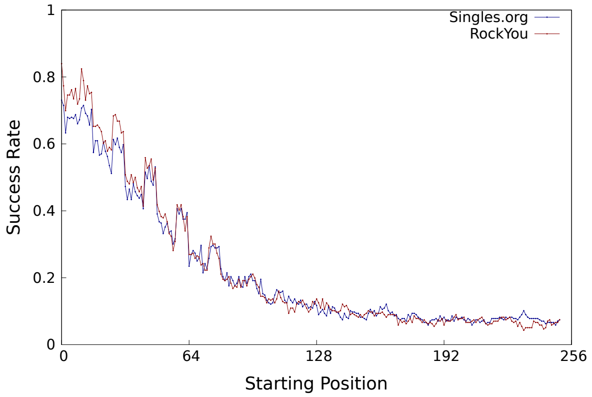


Figure 1: Recovery rate for Singles.org passwords using RockYou data set as dictionary, compared to recovery rate for RockYou passwords using RockYou data set as dictionary ($S = 2^{24}$, $n = 6$, $T = 5$, $1 \leq r \leq 251$, double-byte attack).

Many data sets are available from password breaches. We settled on using RockYou for the attack dictionary because it was one of the biggest data sets in which all passwords and their associated frequencies were available, and because the distribution of passwords, while certainly skewed, was less skewed than for other data sets. We used Singles.org for the target set because the Singles.org breach occurred later than the RockYou breach, so that the former could reasonably used as an attack dictionary for the latter. Moreover, the Singles.org distribution being quite different from that for RockYou makes password recovery against Singles.org using RockYou as a dictionary more challenging for our attacks. A detailed evaluation of the extent to which the success rates of our attacks depend on the choice of attack dictionary and target set is beyond the scope of this current work.

A limitation of our approach as described is that we assume the password length n to be already known. Sev-

Algorithm 4: Double-byte attack

```
input : $c_{i,j} : 0 \leq i < S, 0 \leq j < n$ – array formed from $S$ independent encryptions of fixed $n$-byte candidate $X$
        $r$ – starting position of $X$ in plaintext stream
        $\mathscr{X}$ – collection of $N$ candidates
        $p_x$ – *a priori* probability of candidates $x \in \mathscr{X}$
        $p_{r+j,z}$ $(0 \leq j < n,\ z \in \mathscr{B})$ – single-byte keystream distribution
        $p_{r+j,z_1,z_2}$ $(0 \leq j < n-1,\ z_1, z_2 \in \mathscr{B})$ – double-byte keystream distribution
output: $\{\gamma_x : x \in \mathscr{X}\}$ – set of (approximate) log likelihoods for candidates in $\mathscr{X}$
begin
    for $j = 0$ to $n-2$ do
        for $z_1 = \texttt{0x00}$ to $\texttt{0xFF}$ do
            $N'_{z,j} \leftarrow 0$
            for $z_2 = \texttt{0x00}$ to $\texttt{0xFF}$ do
                $N'_{z_1,z_2,j} \leftarrow 0$
    for $j = 0$ to $n-2$ do
        for $i = 0$ to $S-1$ do
            $N'_{c_{i,j},j} \leftarrow N'_{c_{i,j},j} + 1$
            $N'_{c_{i,j},c_{i,j+1},j} \leftarrow N'_{c_{i,j},c_{i,j+1},j} + 1$
    for $j = 1$ to $n-2$ do
        for $y = \texttt{0x00}$ to $\texttt{0xFF}$ do
            for $z = \texttt{0x00}$ to $\texttt{0xFF}$ do
                $N_{y,z,j} \leftarrow N'_{z \oplus y,j}$
            $L_{y,j} = \sum_{z \in \mathscr{B}} N_{y,z,j} \log(p_{r+j,z})$,
    for $j = 0$ to $n-2$ do
        for $y_1 = \texttt{0x00}$ to $\texttt{0xFF}$ do
            for $y_2 = \texttt{0x00}$ to $\texttt{0xFF}$ do
                for $z_1 = \texttt{0x00}$ to $\texttt{0xFF}$ do
                    for $z_2 = \texttt{0x00}$ to $\texttt{0xFF}$ do
                        $N_{y_1,y_2,z_1,z_2,j} \leftarrow N'_{z_1 \oplus y_1, z_2 \oplus y_2,,j}$
                $L_{y_1,y_2,j} = \sum_{z_1 \in \mathscr{B}} \sum_{z_2 \in \mathscr{B}} N_{y_1,y_2,z_1,z_2,j} \log(p_{r+j,z_1,z_2})$,
    for $x = (x_0, \ldots, x_{n-1}) \in \mathscr{X}$ do
        $\gamma_x \leftarrow \log(p_x) + \sum_{j=0}^{n-2} L_{x_j,x_{j+1},j} - \sum_{j=1}^{n-2} L_{x_j,j}$
    return $\{\gamma_x : x \in \mathscr{X}\}$
```

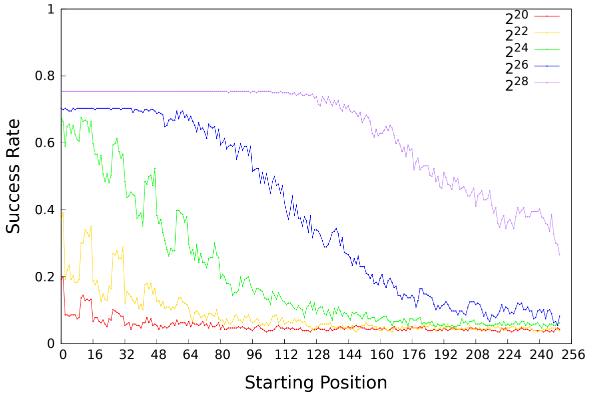


Figure 2: Recovery rates for single-byte algorithm for $S = 2^{20}, \dots, 2^{28}$ ($n = 6$, $T = 5$, $1 \le r \le 251$).

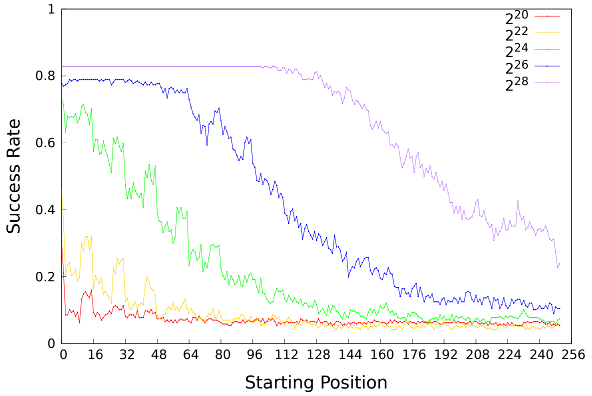


Figure 3: Recovery rates for double-byte algorithm for $S = 2^{20}, \dots, 2^{28}$ ($n = 6$, $T = 5$, $1 \le r \le 251$).

eral solutions to this problem are described in the full version [12].

4.2 Results

Single-Byte Attack We ran the attack described in Algorithm 3 with our default parameters ($n = 6$, $T = 5$, $1 \le r \le 251$) for $S = 2^s$ with $s \in \{20, 22, 24, 26, 28\}$ and evaluated the attack's success rate. We used our default of 256 independent runs per parameter set. The results are shown in Figure 2. We observe that:

- The performance of the attack improves markedly as S, the number of ciphertexts, increases, but the success rate is bounded by 75%. We attribute this to the use of one dictionary (RockYou) to recover passwords from another (Singles.org) – for the same attack parameters, we achieved 100% success rates when using RockYou against RockYou, for example.
- For 2^{24} ciphertexts we see a success rate of greater than 60% for small values of r, the assumed position of the password in the RC4 keystream. We see a drop to below 50% for starting positions greater than 32. We note the effect of the key-length-dependent biases on password recovery; passwords encrypted at starting positions $16\ell - n, 16\ell - n + 1, \dots, 16\ell - 1, 16\ell$, where $\ell = 1, 2, \dots, 6$, have a higher probability of being recovered in comparison to neighbouring starting positions.
- For 2^{28} ciphertexts we observe a success rate of more than 75% for $1 \le r \le 120$.

Double-Byte Attack Analogously, we ran the attack of Algorithm 4 for $S = 2^s$ with $s \in \{20, 22, 24, 26, 28\}$ and our defaults of $n = 6$, $T = 5$. The results for these simulations are shown in Figure 3. Note that:

- Again, at 2^{24} ciphertexts the effect of key-length-dependent biases is visible.
- For 2^{26} ciphertexts we observe a success rate that is greater than 78% for $r \le 48$.

Comparing the Single-Byte Attack with a Naive Algorithm Figure 4 provides a comparison between our single-byte algorithm with $T = 1$ and a naive password recovery attack based on the methods of [2], in which the password bytes are recovered one at a time by selecting the highest likelihood byte value in each position and declaring success if all bytes of the password are recovered correctly. Significant improvement over the naive attack can be observed, particularly for high values of r. For example with $S = 2^{24}$, the naive algorithm essentially has a success rate of zero for every r, whereas our single-byte algorithm has a success rate that exceeds 20% for $1 \le r \le 63$. By way of comparison, an attacker knowing the password length and using the obvious guessing strategy would succeed with probability 4.2% with a single guess, this being the *a priori* probability of the password 123456 amongst all length 6 passwords in the Singles.org dataset (and 123456 being the highest ranked password in the RockYou dictionary, so the first one that an attacker using this strategy with the RockYou dictionary would try). As another example, with $S = 2^{28}$ ciphertexts, a viable recovery rate is observed all the way up to $r = 251$ for our single-byte algorithm, whereas the naive algorithm fails badly beyond $r = 160$ for even this large value of S. Note however that the naive attack can achieve a success rate of 100% for sufficiently large S, whereas our attack cannot. This is because the naive attack directly

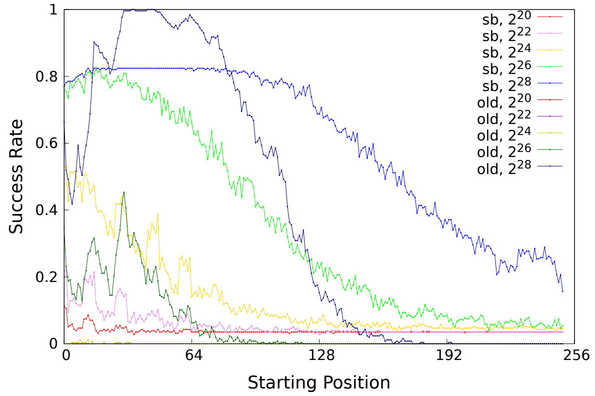


Figure 4: Performance of our single-byte algorithm versus a naive single-byte attack based on the methods of AlFardan *et al.* (labelled "old") ($n = 6$, $T = 1$, $1 \le r \le 251$).

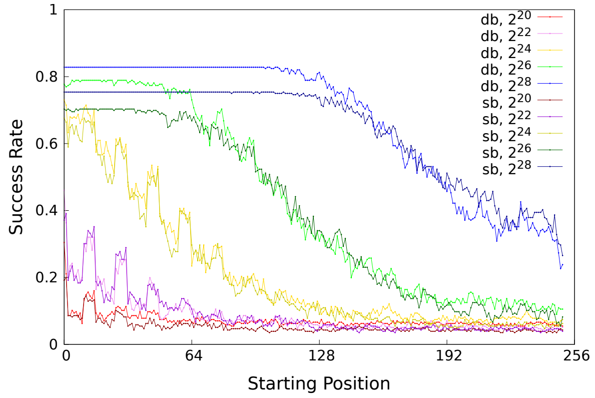


Figure 5: Recovery rate of single-byte versus double-byte algorithm for $S = 2^{20}, \dots, 2^{28}$ ($n = 6$, $T = 5$, $1 \le r \le 251$).

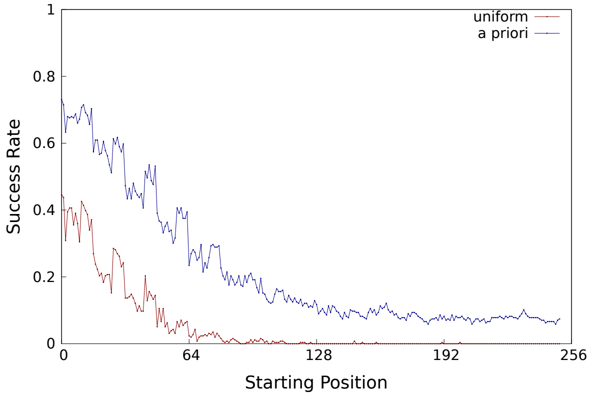


Figure 6: Recovery rate for uniformly distributed passwords versus known *a priori* distribution ($S = 2^{24}$, $n = 6$, $T = 5$, $1 \le r \le 251$, double-byte algorithm).

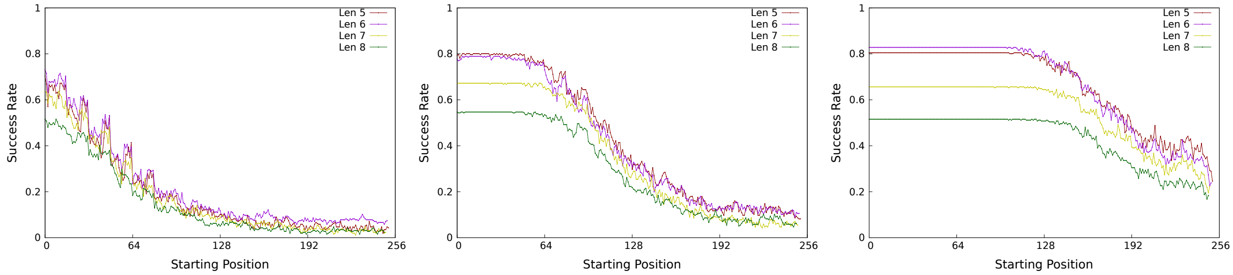
computes a password candidate rather than evaluating the likelihood of candidates from a list which may not contain the target password. On the other hand, our attack trivially supports larger values of T, whereas the naive attack is not so easily modified to enable this feature.

Comparing the Single-Byte and Double-Byte Attacks Figure 5 provides a comparison of our single-byte and double-byte attacks. With all other parameters equal, the success rates are very similar for the initial 256 positions. The reason for this is the absence of many strong double-byte biases that do not arise from the known single-byte biases in the early positions of the RC4 keystream.

Effect of the *a priori* Distribution As a means of testing the extent to which our success rates are influenced by knowledge of the *a priori* probabilities of the candidate passwords, we ran simulations in which we tried to recover passwords sampled correctly from the Singles.org dataset but using a uniform *a priori* distribution for the RockYou-based dictionary used in the attack. Figure 6 shows the results ($S = 2^{24}$, $n = 6$, $T = 5$, double-byte attack) of these simulations, compared to the results we obtain by exploiting the *a priori* probabilities in the attack. It can be seen that a significant gain is made by using the *a priori* probabilities, with the uniform attack's success rate rapidly dropping to zero at around $r = 128$.

Effect of Password Length Figure 7 shows the effect of increasing n, the password length, on recovery rates, with the sub-figures showing the performance of our double-byte attack for different numbers of ciphertexts ($S = 2^s$ with $s \in \{24, 26, 28\}$). Other parameters are set to their default values. As intuition suggests, password recovery becomes more difficult as the length increases. Also notable is that the ceiling on success rate of our attack decreases with increasing n, dropping from more than 80% for $n = 5$ to around 50% for $n = 8$. This is due to the fact that only 48% of the length 8 passwords in the Singles.org data set actually occur in the RockYou attack dictionary: our attack is doing as well as it can in this case, and we would expect stronger performance with an attack dictionary that is better matched to the target site.

Effect of Increasing Try Limit T Recall that the parameter T defines the number of password trials our attacks make. The number of permitted attempts for specific protocols like BasicAuth and IMAP is server-dependent and not mandated in the relevant specifications. Whilst not specific to our chosen protocols, a 2010 study [5] showed that 84% of websites surveyed allowed at least $T = 100$ attempts; many websites appear to actually al-



(a) 2^{24} ciphertexts

(b) 2^{26} ciphertexts

(c) 2^{28} ciphertexts

Figure 7: Effect of password length on recovery rate ($T = 5$, $1 \leq r \leq 251$, double-byte algorithm).

low $T = \infty$. Figure 8 shows the effect of varying T in our double-byte algorithm for different numbers of ciphertexts ($S = 2^s$ with $s \in \{24, 26, 28\}$). Other parameters are set to their default values. It is clear that allowing large values of T boosts the success rate of the attacks.

Note however that a careful comparison must be made between our attack with parameter T and the success rate of the obvious password guessing attack given T attempts. Such a guessing attack does not require any ciphertexts but instead uses the *a priori* distribution on passwords in the attack dictionary (RockYou) to make guesses for the target password in descending order of probability, the success rate being determined by the *a priori* probabilities of the guessed passwords in the target set (Singles.org). Clearly, our attacks are only of value if they significantly out-perform this ciphertext-less attack.

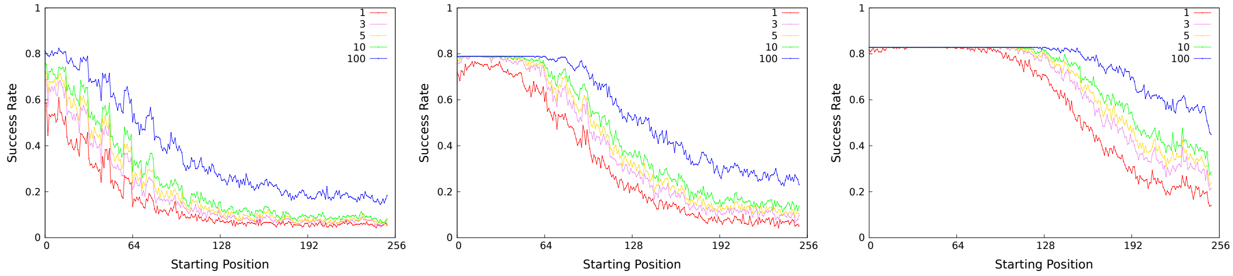
Figure 9 shows the results of plotting $\log_2(T)$ against success rate α for $S = 2^s$ with $s \in \{14, 16, \ldots, 28\}$. The figure then illustrates the value of T necessary in our attack to achieve a given password recovery rate α for different values of S. This measure is related to the α-work-factor metric explored in [4] (though with the added novelty of representing a work factor when one set of passwords is used to recover passwords from a different set). To generate this figure, we used 1024 independent runs rather than the usual 256, but using a fixed set of 1024 passwords sampled according to the *a priori* distribution for Singles.org. This was in an attempt to improve the stability of the results (with small numbers of ciphertexts S, the success rate becomes heavily dependent on the particular set of passwords selected and their *a priori* probabilities, while we wished to have comparability across different values of S).

The success rates shown in Figure 9 are for our double-byte attack with $n = 6$ and $r = 133$, this specific choice of r being motivated by it being the location of passwords for our BasicAuth attack proof-of-concept when the Chrome browser is used (similar results are obtained for other values of r). The graph also shows the corresponding work factor T as a function of α for the guessing attack (labeled “optimal guessing” in the figure).

Figure 9 shows that our attack far outperforms the guessing attack for larger values of S, with a significant advantage accruing for $S = 2^{24}$ and above. However, the advantage over the guessing attack for smaller values of S, namely 2^{20} and below, is not significant. This can be attributed to our attack simply not being able to compute stable enough statistics for these small numbers of ciphertexts. In turn, this is because the expected random fluctuations in the keystream distributions overwhelm the small biases; in short, the signal does not sufficiently exceed the noise for these low values of S.

Effect of Base64 Encoding We investigated the effect of Base64 encoding of passwords on recovery rates, since many application layer protocols use such an encoding. The encoding increases the password length, making recovery harder, but also introduces redundancy, potentially helping the recovery process to succeed. Figure 10 shows our simulation results comparing the performance of our double-byte algorithm acting on 6-character passwords and on Base64 encoded versions of them. It is apparent from the figure that the overall effect of the Base64 encoding is to help our attack to succeed. In practice, the start of the target password may not be well-aligned with the Base64 encoding process (for example, part of the last character of the username and/or a delimiter such as “:” may be jointly encoded with part of the first character of the password). This can be handled by building a special-purpose set of candidates $\mathscr{X}$ for each possibility. Handling this requires some care when mounting a real attack against a specific protocol; a detailed analysis is deferred to future work.

Shifting Attack In certain application protocols and attack environments (such as HTTPS) it is possible for the adversary to incrementally pad the plaintext messages so that the unknown bytes are always aligned with positions having large keystream biases. Our algorithm descriptions



(a) 2^{24} ciphertexts (b) 2^{26} ciphertexts (c) 2^{28} ciphertexts

Figure 8: Effect of try limit T on recovery rate ($n = 6$, $1 \leq r \leq 251$, double-byte algorithm).

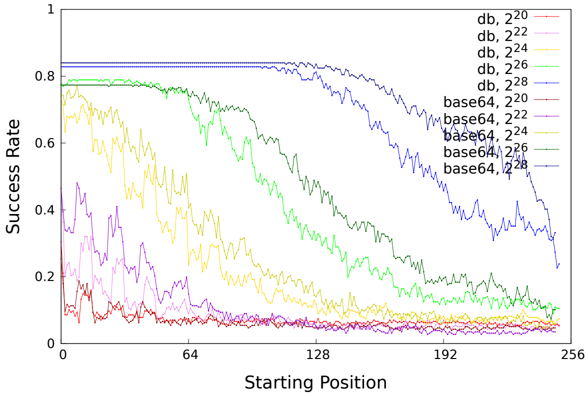


Figure 10: Recovery rate of Base64 encoded password versus a "normal" password for 6-character passwords ($T = 5$, $1 \leq r \leq 251$, double-byte algorithm).

and code are both easily modified to handle this situation, and we have conducted simulations with the resulting shift attack. We report on these simulations in the full version, [12].

5 Practical Validation

In this section we describe proof-of-concept implementations of our attacks against a specific application-layer protocol running over TLS, namely BasicAuth. In the full version [12], we additionally consider the IMAP protocol as a target.

5.1 Introducing BasicAuth

Defined as part of the HTTP/1.0 specification [3] and extended in [11], the Basic Access Authentication scheme (BasicAuth) provides a simple means for controlling access to webpages and other protected resources. In view of its simplicity, the scheme is still very widely used in the enterprise application space. The protocol essentially involves the client sending the server a username and password in Base64 encoded form, and as such, requires the use of a lower-layer secure protocol like TLS to mitigate trivial eavesdropping attacks. Certain web browsers display a login dialog when an initiating challenge message is received from the server and many browsers present users with the option of storing their user credentials in the browser, with the credentials thereafter being automatically presented on behalf of the user.

The client response to the challenge is of the form
`Authorization: Basic Base64(userid:password)`
where `Base64`(·) denotes the Base64 encoding function (which maps 3 characters at a time onto 4 characters of output).

5.2 Attacking BasicAuth

To obtain a working attack against BasicAuth, we need to ensure that two conditions are met:

- The Base64-encoded password included in the BasicAuth client response can be located sufficiently early in the plaintext stream.

- There is a method for forcing a browser to repeatedly send the BasicAuth client response.

We have observed that the first condition is met for particular browsers, including Google Chrome. For example, we inspected HTTPS traffic sent from Chrome to an iChair server.[5] We observed the user's Base64-encoded password being sent with every HTTP(S) request in the same position in the stream, namely position $r = 133$ (this includes 16 bytes consumed by the client's `Finished` message as well as the 20-bytes consumed by the TLS Record Protocol tag). For Mozilla Firefox, the value of r was the less useful 349.

[5] iChair is a popular system for conference reviewing, widely used in the cryptography research community and available from `http://www.baigneres.net/ichair`. It uses BasicAuth as its user authentication mechanism.

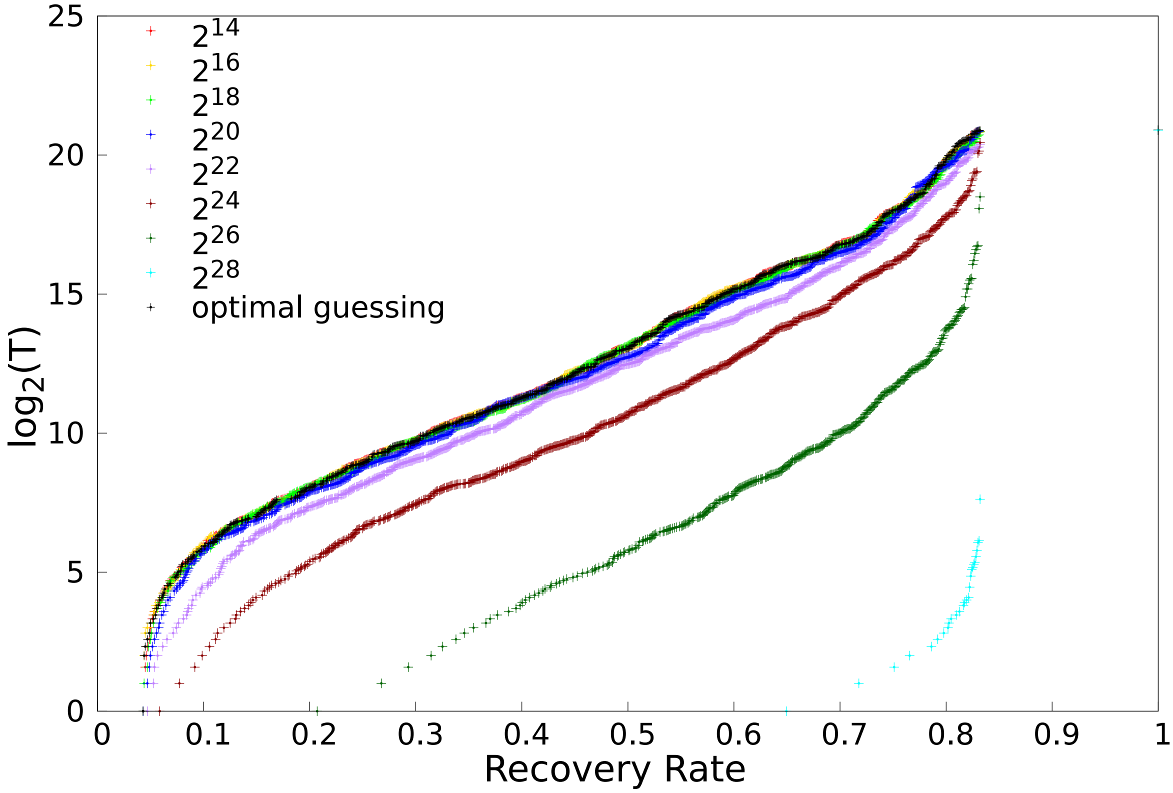


Figure 9: Value of T required to achieve a given password recovery rate α for $S = 2^s$ with $s \in \{14, 16, \ldots, 28\}$ ($n = 6$, $r = 133$, double-byte algorithm).

For the second condition, we adopt the methods used in the BEAST, CRIME and Lucky 13 attacks on TLS, and also used in attacking RC4 in [2]: we assume that the user visits a site `www.evil.com` which loads JavaScript into the user's browser; the JavaScript makes GET or POST requests to the target website at `https://www.good.com` by using `XMLHttpRequest` objects (this is permitted under Cross Origin Resource Sharing (CORS), a mechanism developed to allow JavaScript to make requests to a domain other than the one from which the script originates). The Base64-encoded BasicAuth password is automatically included in each such request. To force the password to be repeatedly encrypted at an early position in the RC4 keystream, we use a MITM attacker to break the TLS connection (by injecting sequences of TCP `FIN` and `ACK` messages into the connection). This requires some careful timing on the part of the JavaScript and the MITM attacker.

We built a proof-of-concept demonstration of these components to illustrate the principles. We set up a virtual network with three virtual machines each running Ubuntu 14.04, kernel version 3.13.0-32. On the first machine, we installed iChair. We configured the iChair web server to use RC4 as its default TLS cipher. The second machine was running the Chrome 38 browser and acted as the client in our attack. We installed the required JavaScript directly on this machine rather than downloading from another site. The third machine acted as the MITM attacker, required to intercept the TLS-protected traffic and to tear-down the TLS connections. We used the Python tool Scapy[6] to run an ARP poisoning attack on the client and server from the MITM so as to be able to intercept packets; with the connection hijacked we were able to force a graceful shutdown of the connection between the client and the server after the password-bearing record had been observed and recorded. We observed that forcing a graceful shutdown of each subsequent connection did allow for TLS resumption (rather than leading to the need for a full TLS Handshake run).

With this setup, the JavaScript running in the client browser sent successive HTTPS GET requests to the

[6] Available at `http://www.secdev.org/projects/scapy/`.

iChair server every 80ms. Our choice of 80ms was motivated by the fact that for our particular configuration, we observed a total time of around 80ms for TLS resumption, delivery of the password-bearing record and the induced shutdown of the TCP connection. This choice enabled us to capture 2^{16} encrypted password-bearing records in 1.6 hours (the somewhat greater than expected time here being due to anomalies in network behaviour). Running at this speed, the attack was stable over a period of hours.

We note that the latency involved in our setup is much lower than would be found in a real network in which the server may be many hops away from the client: between 500ms and 1000ms is typical for establishing an initial TLS connection to a remote site, with the latency being roughly half that for session resumptions. Notably, the cost of public key operations is not the issue, but rather the network latency involved in the round-trips required for TCP connection establishment and then running the TLS Handshake. However, browsers also open up multiple TLS connections in parallel when fetching multiple resources from a site, as a means of reducing the latency perceived by users; the maximum number of concurrent connections per server is 6 for both the Chrome and Firefox browsers (though, we only ever saw roughly half this number in practice, even with low inter-request times). This means that, assuming a TLS resumption latency (including the client's TCP `SYN`, delivery of the password-bearing record and the final, induced TCP `ACK`) of 250ms and the JavaScript is running fast enough to induce the browser to maintain 6 connections in parallel, the amount of time needed to mount an attack with $S = 2^{26}$ would be on the order of 776 hours. If the latency was further reduced to 100ms (because of proximity of the server to the client), the attack execution time would be reduced to 312 hours.

Again setting $n = 6$, $T = 100$, $r = 133$ and using the simulation results displayed in Figure 10, we would expect a success rate of 64.4% for this setup (with $S = 2^{26}$). For $T = 5$, the corresponding success rate would be 44.5%.

We emphasise that we have not executed a complete attack on these scales, but merely demonstrated the feasibility of the attack in our laboratory setup.

6 Conclusion and Open Problems

We have presented plaintext recovery attacks that derive from a formal Bayesian analysis of the problem of estimating plaintext likelihoods given an *a priori* plaintext distribution, suitable keystream distribution information, and a large number of encryptions of a fixed plaintext under independent keys. We applied these ideas to the specific problem of recovering passwords encrypted by the RC4 algorithm with 128-bit keys as used in TLS, though they are of course more generally applicable – to uses of RC4 other than in TLS, and to stream ciphers with non-uniform keystream distributions in general. Using large-scale simulations, we have investigated the performance of these attacks under different settings for the main parameters.

We then studied the applicability of these attacks for a specific application layer protocol, BasicAuth. For certain browsers and clients, the passwords were located at a favourable point in the plaintext stream and we could induce the password to be repeatedly encrypted under fresh, random keys. We built a proof-of-concept implementation of the attack. It was difficult to arrange for the rate of generation of encryptions to be as high as desired for a speedy attack. This was mainly due to the latency associated with TLS connection establishment (even with session resumption) rather than any fundamental barrier.

Good-to-excellent password recovery success rates can be achieved using 2^{24} – 2^{28} ciphertexts in our attacks. We also demonstrated that our single-byte attack for password recovery significantly outperforms a naive password recovery attack based on the ideas of [2]. We observed an improvement over a guessing strategy even for low numbers (2^{22} or 2^{24}) of ciphertexts. By contrast to these numbers, the preferred double-byte attack of [2] required on the order of 2^{34} encryptions to recover a 16-byte cookie, though without incurring the time overheads arising from TLS session resumption that our approach incurs.

Our research has led to the identification of a number of areas for further work:

- Our Bayesian approach can also be applied to the situation where we model the plaintext as a word from a language described as a Markov model with memory. It would be interesting to investigate the extent to which this approach can be applied to either password recovery or more general analysis of, say, typical HTTP traffic.

- We have focussed on the use of the single-byte biases described in [2] and the double-byte biases of Fluhrer and McGrew (and from our own extensive computations for the first 512 keystream positions). Other biases in RC4 keystreams are known, for example [14]. It is a challenge to integrate these in our Bayesian framework, with the aim being to further improve our attacks.

- We identified new double-byte biases early in the RC4 keystream which deserve a theoretical explanation.

- It would be an interesting challenge to develop algorithms for constructing synthetic, site-specific dictionaries along with *a priori* probability distributions. Existing work in this direction includes Marx's WordHound tool.[7]

- We identified several open questions in the discussion of our simulation results, including the effect of the choice of password data sets on success rates, and the evaluation of different methods for recovering the target password's length.

Acknowledgements

We would like to thank Google, Dan Kaminsky at White Ops and Ingo von Maurich at Ruhr Universität Bochum for their generous donation of computing resources. Dan gave us free rein on a 512-core system for the 4800 core-days necessary to perform our double-byte keystream distribution estimates, while resources from Google and Ruhr Universität Bochum were used to conduct our attack simulations. We would also like to thank Alexei Melnikov for acting as our IMAP oracle.

Garman was funded by a generous grant from the Mozilla Foundation and supported by the Office of Naval Research under contract N00014-14-1-0333; Paterson was supported by an EPSRC Leadership Fellowship, EP/H005455/1; van der Merwe was supported by the EPSRC as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London.

References

[1] Adams, A., and Sasse, M. A. Users are not the enemy. *Commun. ACM 42*, 12 (Dec. 1999), 40–46.

[2] AlFardan, N. J., Bernstein, D. J., Paterson, K. G., Poettering, B., and Schuldt, J. C. N. On the Security of RC4 in TLS. In *Proceedings of the 22nd USENIX Conference on Security* (Berkeley, CA, USA, 2013), SEC'13, USENIX Association, pp. 305–320.

[3] Berners-Lee, T., Fielding, R., and Frystyk, H. The Hypertext Transfer Protocol HTTP/1.0. RFC 1945 (Informational), May 1996.

[4] Bonneau, J. The science of guessing: Analyzing an anonymized corpus of 70 million passwords. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA* (2012), IEEE Computer Society, pp. 538–552.

[5] Bonneau, J., and Preibusch, S. The password thicket: Technical and market failures in human authentication on the web. In *9th Annual Workshop on the Economics of Information Security, WEIS 2010, Harvard University, Cambridge, MA, USA, June 7 - 8* (2010).

[6] Dierks, T., and Allen, C. The TLS Protocol Version 1.0. RFC 2246, Internet Engineering Task Force, Jan. 1999.

[7] Dierks, T., and Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346, Internet Engineering Task Force, Apr. 2006.

[8] Dierks, T., and Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, Internet Engineering Task Force, Aug. 2008.

[9] Florencio, D., and Herley, C. A Large-scale Study of Web Password Habits. In *Proceedings of the 16th International Conference on World Wide Web* (New York, NY, USA, 2007), WWW '07, ACM, pp. 657–666.

[10] Fluhrer, S. R., and McGrew, D. Statistical analysis of the alleged RC4 keystream generator. In *FSE* (2000), B. Schneier, Ed., vol. 1978 of *Lecture Notes in Computer Science*, Springer, pp. 19–30.

[11] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and Stewart, L. HTTP Authentication: Basic and Digest Access authentication. RFC 2617 (Informational), June 1999.

[12] Garman, C., Paterson, K. G., and van der Merwe, T. Attacks only get better: Password recovery attacks against RC4 in TLS. Full version of this paper. Available from `http://www.isg.rhul.ac.uk/tls/RC4mustdie.html`.

[13] Isobe, T., Ohigashi, T., Watanabe, Y., and Morii, M. Full plaintext recovery attack on broadcast RC4. In *Preproceedings of FSE* (2013).

[14] Mantin, I. Predicting and distinguishing attacks on RC4 keystream generator. In *EUROCRYPT* (2005), R. Cramer, Ed., vol. 3494 of *Lecture Notes in Computer Science*, Springer, pp. 491–506.

[15] Mantin, I., and Shamir, A. A practical attack on broadcast RC4. In *FSE* (2001), M. Matsui, Ed., vol. 2355 of *Lecture Notes in Computer Science*, Springer, pp. 152–164.

[16] Ohigashi, T., Isobe, T., Watanabe, Y., and Morii, M. How to recover any byte of plaintext on RC4. In *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers* (2013), T. Lange, K. E. Lauter, and P. Lisonek, Eds., vol. 8282 of *Lecture Notes in Computer Science*, Springer, pp. 155–173.

[17] Salowey, J., Zhou, H., Eronen, P., and Tschofenig, H. Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 5077 (Proposed Standard), Jan. 2008.

[18] Sarkar, S., Sen Gupta, S., Paul, G., and Maitra, S. Proving TLS-attack related open biases of RC4. *IACR Cryptology ePrint Archive 2013* (2013), 502.

[19] Sen Gupta, S., Maitra, S., Paul, G., and Sarkar, S. (Non-) random sequences from (non-) random permutations – analysis of RC4 stream cipher. *Journal of Cryptology 27*, 1 (2012), 67–108.

[20] Weir, M., Aggarwal, S., Collins, M. P., and Stern, H. Testing metrics for password creation policies by attacking large sets of revealed passwords. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010* (2010), E. Al-Shaer, A. D. Keromytis, and V. Shmatikov, Eds., ACM, pp. 162–175.

[21] Yan, J., Blackwell, A., Anderson, R., and Grant, A. Password Memorability and Security: Empirical Results. *IEEE Security and Privacy 2*, 5 (Sept. 2004), 25–31.

[22] Zviran, M., and Haga, W. J. Password Security: An Empirical Study. *J. Manage. Inf. Syst. 15*, 4 (Mar. 1999), 161–185.

[7] `https://bitbucket.org/mattinfosec/wordhound`.

Eclipse Attacks on Bitcoin's Peer-to-Peer Network

Ethan Heilman[*] *Alison Kendler*[*] *Aviv Zohar*[†] *Sharon Goldberg*[*]
[*]*Boston University* [†]*Hebrew University/MSR Israel*

Abstract

We present eclipse attacks on bitcoin's peer-to-peer network. Our attack allows an adversary controlling a sufficient number of IP addresses to monopolize all connections to and from a victim bitcoin node. The attacker can then exploit the victim for attacks on bitcoin's mining and consensus system, including N-confirmation double spending, selfish mining, and adversarial forks in the blockchain. We take a detailed look at bitcoin's peer-to-peer network, and quantify the resources involved in our attack via probabilistic analysis, Monte Carlo simulations, measurements and experiments with live bitcoin nodes. Finally, we present countermeasures, inspired by botnet architectures, that are designed to raise the bar for eclipse attacks while preserving the openness and decentralization of bitcoin's current network architecture.

1 Introduction

While cryptocurrency has been studied since the 1980s [22, 25, 28], bitcoin is the first to see widespread adoption.A key reason for bitcoin's success is its baked-in decentralization. Instead of using a central bank to regulate currency, bitcoin uses a decentralized network of nodes that use computational proofs-of-work to reach consensus on a distributed public ledger of transactions, *aka.,* the *blockchain*. Satoshi Nakamoto [52] argues that bitcoin is secure against attackers that seek to shift the blockchain to an inconsistent/incorrect state, as long as these attackers control less than half of the computational power in the network. But underlying this security analysis is the crucial assumption of *perfect information*; namely, that all members of the bitcoin ecosystem can observe the proofs-of-work done by their peers.

While the last few years have seen extensive research into the security of bitcoin's computational proof-of-work protocol *e.g.,* [14, 29, 36, 37, 45, 49, 50, 52, 58, 60], less attention has been paid to the peer-to-peer network used to broadcast information between bitcoin nodes (see Section 8). The bitcoin peer-to-peer network, which is bundled into the core bitcoind implementation, *aka.,* the Satoshi client, is designed to be open, decentralized, and independent of a public-key infrastructure. As such, cryptographic authentication between peers is not used, and nodes are identified by their IP addresses (Section 2). Each node uses a randomized protocol to select eight peers with which it forms long-lived *outgoing connections*, and to propagate and store addresses of other potential peers in the network. Nodes with public IPs also accept up to 117 *unsolicited incoming connections* from any IP address. Nodes exchange views of the state of the blockchain with their incoming and outgoing peers.

Eclipse attacks. This openness, however, also makes it possible for adversarial nodes to join and attack the peer-to-peer network. In this paper, we present and quantify the resources required for *eclipse attacks* on nodes with public IPs running bitcoind version 0.9.3. In an eclipse attack [27, 61, 62], the attacker monopolizes all of the victim's incoming and outgoing connections, thus isolating the victim from the rest of its peers in the network. The attacker can then filter the victim's view of the blockchain, force the victim to waste compute power on obsolete views of the blockchain, or coopt the victim's compute power for its own nefarious purposes (Section 1.1). We present *off-path* attacks, where the attacker controls endhosts, but not key network infrastructure between the victim and the rest of the bitcoin network. Our attack involves rapidly and repeatedly forming unsolicited incoming connections to the victim from a set of endhosts at attacker-controlled IP addresses, sending bogus network information, and waiting until the victim restarts (Section 3). With high probability, the victim then forms all eight of its outgoing connections to attacker-controlled addresses, and the attacker also monopolizes the victim's 117 incoming connections.

Our eclipse attack uses extremely low-rate TCP connections, so the main challenge for the attacker is to obtain a sufficient number of IP addresses (Section 4). We consider two attack types: (1) infrastructure attacks, modeling the threat of an ISP, company, or nation-state that holds several *contiguous* IP address blocks and seeks to subvert bitcoin by attacking its peer-to-peer network, and (2) botnet attacks, launched by bots with addresses in *diverse* IP address ranges. We use probabilistic analysis, (Section 4) measurements (Section 5), and experiments on our own live bitcoin nodes (Section 6) to find that while botnet attacks require far fewer IP addresses, there are hundreds of organizations that have sufficient IP resources to launch eclipse attacks (Section 4.2.1). For example, we show how an infrastructure attacker with 32 distinct /24 IP address blocks (8192 address total), or a botnet of 4600 bots, can always eclipse a victim with at least 85% probability; this is independent of the number of nodes in the network. Moreover, 400 bots sufficed in tests on our live bitcoin nodes. To put this in context, if 8192 attack nodes joined today's network (containing $\approx$ 7200 public-IP nodes [4]) and honestly followed the peer-to-peer protocol, they could eclipse a target with probability about $(\frac{8192}{7200+8192})^8 = 0.6\%$.

Our attack is only for nodes with public IPs; nodes with private IPs may be affected if all of their outgoing connections are to eclipsed public-IP nodes.

Countermeasures. Large miners, merchant clients and online wallets have been known to modify bitcoin's networking code to reduce the risk of network-based attacks. Two countermeasures are typically recommended [3]: (1) disabling incoming connections, and (2) choosing 'specific' outgoing connections to well-connected peers or known miners (*i.e.,* use whitelists). However, there are several problems with scaling this to the full bitcoin network. First, if incoming connections are banned, how do new nodes join the network? Second, how does one decide which 'specific' peers to connect to? Should bitcoin nodes form a private network? If so, how do they ensure compute power is sufficiently decentralized to prevent mining attacks?

Indeed, if bitcoin is to live up to its promise as an open and decentralized cryptocurrency, we believe its peer-to-peer network should be open and decentralized as well. Thus, our next contribution is a set of countermeasures that preserve openness by allowing unsolicited incoming connections, while raising the bar for eclipse attacks (Section 7). Today, an attacker with enough addresses can eclipse *any* victim that accepts incoming connections and then restarts. Our countermeasures ensure that, with high probability, if a victim stores enough legitimate addresses that accept incoming connections, then the victim be cannot eclipsed *regardless of the number of IP addresses the attacker controls.* Our countermeasures 1, 2, and 6 have been deployed in bitcoind v0.10.1; we also developed a patch [40] with Countermeasures 3,4.

1.1 Implications of eclipse attacks

Apart from disrupting the bitcoin network or selectively filtering a victim's view of the blockchain, eclipse attacks are a useful building block for other attacks.

Engineering block races. A block race occurs when two miners discover blocks at the same time; one block will become part of the blockchain, while the other "orphan block" will be ignored, yielding no mining rewards for the miner that discovered it. An attacker that eclipses many miners can engineer block races by hording blocks discovered by eclipsed miners, and releasing blocks to both the eclipsed and non-eclipsed miners once a competing block has been found. Thus, the eclipsed miners waste effort on orphan blocks.

Splitting mining power. Eclipsing an x-fraction of miners eliminates their mining power from the rest of the network, making it easier to launch mining attacks (*e.g.,* the 51% attack [52]). To hide the change in mining power under natural variations [19], miners could be eclipsed gradually or intermittently.

Selfish mining. With selfish mining [14,29,37,60], the attacker strategically withholds blocks to win more than its fair share of mining rewards. The attack's success is parameterized by two values: α, the ratio of mining power controlled by the attacker, and γ, the ratio of honest mining power that will mine on the attacker's blocks during a block race. If γ is large, then α can be small. By eclipsing miners, the attacker increases γ, and thus decreases α so that selfish mining is easier. To do this, the attacker drops any blocks discovered by eclipsed miners that compete with the blocks discovered by the selfish miners. Next, the attacker increases γ by feeding only the selfish miner's view of the blockchain to the eclipsed miner; this coopts the eclipsed miner's compute power, using it to mine on the selfish-miner's blockchain.

Attacks on miners can harm the entire bitcoin ecosystem; mining pools are also vulnerable if their gateways to the public bitcoin network can be eclipsed. Eclipsing can also be used for double-spend attacks on non-miners, where the attacker spends some bitcoins multiple times:

0-confirmation double spend. In a 0-confirmation transaction, a customer pays a transaction to a merchant who releases goods to the customer *before* seeing a block confirmation *i.e.,* seeing the transaction in the blockchain [18]. These transactions are used when it is inappropriate to wait the 5-10 minutes typically needed to for a block confirmation [20], *e.g.,* in retail point-of-sale systems like BitPay [5], or online gambling sites like Betcoin [57]. To launch a double-spend attack against the merchant [46], the attacker eclipses the merchant's bitcoin node, sends the merchant a transaction T for goods, and sends transaction T' double-spending those bitcoins to the rest of the network. The merchant releases the goods to the attacker, but since the attacker controls all of the merchant's connections, the merchant cannot tell the rest of the network about T, which meanwhile confirms T'. The attacker thus obtains the goods without paying. 0-confirmation double-spends have occurred in the wild [57]. This attack is as effective as a Finney attack [39], but uses eclipsing instead of mining power.

N-confirmation double spend. If the attacker has eclipsed an x-fraction of miners, it can also launch N-confirmation double-spending attacks on an eclipsed merchant. In an N-confirmation transaction, a merchant releases goods only after the transaction is confirmed in a block of depth $N-1$ in the blockchain [18]. The attacker sends its transaction to the eclipsed miners, who incorporate it into their (obsolete) view of the blockchain. The attacker then shows this view of blockchain to the eclipsed merchant, receives the goods, and sends both the merchant and eclipsed miners the (non-obsolete) view of blockchain from the non-eclipsed miners. The eclipsed miners' blockchain is orphaned, and the attacker obtains goods without paying. This is similar to an attack launched by a mining pool [10], but our attacker eclipses miners instead of using his own mining power.

Other attacks exist, *e.g.,* a transaction hiding attack on nodes running in SPV mode [16].

2 Bitcoin's Peer-to-Peer Network

We now describe bitcoin's peer-to-peer network, based on bitcoind version 0.9.3, the most current release from 9/27/2014 to 2/16/2015, whose networking code was largely unchanged since 2013. This client was originally written by Satoshi Nakamoto, and has near universal market share for public-IP nodes (97% of public-IP nodes according to Bitnode.io on 2/11/2015 [4]).

Peers in the bitcoin network are identified by their IP addresses. A node with a public IP can initiate up to *eight outgoing connections* with other bitcoin nodes, and accept up to 117 *incoming connections*.[1] A node with a private IP only initiates eight outgoing connections. Connections are over TCP. Nodes only propagate and store public IPs; a node can determine if its peer has a public IP by comparing the IP packet header with the bitcoin VERSION message. A node can also connect via Tor; we do not study this, see [16, 17] instead. We now describe how nodes propagate and store network information, and how they select outgoing connections.

2.1 Propagating network information

Network information propagates through the bitcoin network via DNS seeders and ADDR messages.

DNS seeders. A DNS seeder is a server that responds to DNS queries from bitcoin nodes with a (not cryptographically-authenticated) list of IP addresses for bitcoin nodes. The seeder obtains these addresses by periodically crawling the bitcoin network. The bitcoin network has six seeders which are queried in two cases only. The first when a new node joins the network for the first time; it tries to connect to the seeders to get a list of active IPs, and otherwise fails over to a hardcoded list of about 600 IP addresses. The second is when an existing node restarts and reconnects to new peers; here, the seeder is queried only if 11 seconds have elapsed since the node began attempting to establish connections and the node has less than two outgoing connections.

ADDR messages. ADDR messages, containing up to 1000 IP address and their timestamps, are used to obtain network information from peers. Nodes accept unsolicited ADDR messages. An ADDR message is solicited *only* upon establishing a outgoing connection with a peer; the peer responds with up to three ADDR message each containing up to 1000 addresses randomly selected from its tables. Nodes push ADDR messages to peers in two cases. Each day, a node sends its own IP address in a ADDR message to each peer. Also, when a node receives an ADDR message with no more than 10 addresses, it forwards the ADDR message to two randomly-selected connected peers.

2.2 Storing network information

Public IPs are stored in a node's tried and new tables. Tables are stored on disk and persist when a node restarts.

The tried table. The tried table consists of 64 *buckets*, each of which can store up to 64 unique addresses for peers to whom the node has successfully established an incoming or outgoing connection. Along with each stored peer's address, the node keeps the timestamp for the most recent successful connection to this peer.

Each peer's address is mapped to a bucket in tried by taking the hash of the peer's (a) IP address and (b) *group*, where the group defined is the /16 IPv4 prefix containing the peer's IP address.A bucket is selected as follows:

```
SK = random value chosen when node is born.
IP = the peer's IP address and port number.
Group = the peer's group

i = Hash( SK, IP ) % 4
Bucket = Hash( SK, Group, i ) % 64
return Bucket
```

Thus, every IP address maps to a single bucket in tried, and each group maps to up to four buckets.

[1] This is a configurable. Our analysis only assumes that nodes have 8 outgoing connections, which was confirmed by [51]'s measurements.

When a node successfully connects to a peer, the peer's address is inserted into the appropriate `tried` bucket. If the bucket is full (*i.e.,* contains 64 addresses), then *bitcoin eviction* is used: four addresses are randomly selected from the bucket, and the oldest is (1) replaced by the new peer's address in `tried`, and then (2) inserted into the `new` table. If the peer's address is already present in the bucket, the timestamp associated with the peer's address is updated. The timestamp is also updated when an actively connected peer sends a `VERSION`, `ADDR`, `INVENTORY`, `GETDATA` or `PING` message and more than 20 minutes elapsed since the last update.

The `new` table. The `new` table consists of 256 buckets, each of which can hold up 64 addresses for peers to whom the node has not yet initiated a successful connection. A node populates the `new` table with information learned from the DNS seeders, or from `ADDR` messages.

Every address a inserted in `new` belongs to (1) a *group*, defined in our description of the `tried` table, and (2) a *source group*, the group the contains the IP address of the connected peer or DNS seeder from which the node learned address a. The bucket is selected as follows:

```
SK = random value chosen when node is born.
Group     = /16 containing IP to be inserted.
Src_Group = /16 containing IP of peer sending IP.

i = Hash( SK, Src_Group, Group ) % 32
Bucket = Hash( SK, Src_Group, i) % 256
return Bucket
```

Each (*group*, *source group*) pair hashes to a single `new` bucket, while each *group* selects up to 32 buckets in `new`. Each bucket holds unique addresses. If a bucket is full, then a function called `isTerrible` is run over all 64 addresses in the bucket; if any one of the addresses is terrible, in that it is (a) more than 30 days old, or (b) has had too many failed connection attempts, then the terrible address is evicted in favor of the new address; otherwise, *bitcoin eviction* is used with the small change that the evicted address is discarded.

2.3 Selecting peers

New outgoing connections are selected if a node restarts or if an outgoing connection is dropped by the network. A bitcoin node never deliberately drops a connection, except when a blacklisting condition is met (*e.g.,* the peer sends `ADDR` messages that are too large).

A node with $\omega \in [0,7]$ outgoing connections selects the $\omega + 1^{th}$ connection as follows:

(1) Decide whether to select from `tried` or `new`, where

$$\Pr[\text{Select from } \texttt{tried}] = \frac{\sqrt{\rho}(9-\omega)}{(\omega+1)+\sqrt{\rho}(9-\omega)} \qquad (1)$$

and ρ is the ratio between the number of addresses stored in `tried` and the number of addresses stored in `new`.

(2) Select a random address from the table, with a bias towards addresses with fresher timestamps: (i) Choose a random non-empty bucket in the table. (ii) Choose a random position in that bucket. (ii) If there is an address at that position, return the address with probability

$$p(r,\tau) = \min(1, \tfrac{1.2^r}{1+\tau}) \qquad (2)$$

else, reject the address and return to (i). The acceptance probability $p(r,\tau)$ is a function of r, the number of addresses that have been rejected so far, and τ, the difference between the address's timestamp and the current time in measured in ten minute increments.[2]

(3) Connect to the address. If connection fails, go to (1).

3 The Eclipse Attack

Our attack is for a victim with a public IP. Our attacker (1) populates the `tried` table with addresses for its attack nodes, and (2) overwrites addresses in the `new` table with "trash" IP addresses that are not part of the bitcoin network. The "trash" addresses are unallocated (*e.g.,* listed as "available" by [56]) or as "reserved for future use" by [43] (*e.g.,* 252.0.0.0/8). We fill `new` with "trash" because, unlike attacker addresses, "trash" is not a scarce resource. The attack continues until (3) the victim node restarts and chooses new outgoing connections from the `tried` and `new` tables in its persistant storage (Section 2.3). With high probability, the victim establishes all eight outgoing connections to attacker addresses; all eight addresses will be from `tried`, since the victim cannot connect to the "trash" in `new`. Finally, the attacker (5) occupies the victim's remaining 117 incoming connections. We now detail each step of our attack.

3.1 Populating `tried` and `new`

The attacker exploits the following to fill `tried` and `new`:

1. Addresses from unsolicited incoming connections are stored in the `tried` table; thus, the attacker can insert an address into the victim's `tried` table simply by connecting to the victim from that address. Moreover, the *bitcoin eviction* discipline means that the attacker's fresher addresses are likely to evict any older legitimate addresses stored in the `tried` table (Section 2.2).

2. A node accepts unsolicited `ADDR` messages; these addresses are inserted directly into the `new` table without testing their connectivity (Section 2.2). Thus, when our attacker connects to the victim from an adversarial address, it can also send `ADDR` messages with 1000 "trash"

[2] The algorithm also considers the number of failed connections to this address; we omit this because it does not affect our analysis.

addresses. Eventually, the trash overwrites all legitimate addresses in new. We use "trash" because we do not want to waste our IP address resources on overwriting new.

3. Nodes only rarely solicit network information from peers and DNS seeders (Section 2.1). Thus, while the attacker overwrites the victim's tried and new tables, the victim almost never counteracts the flood of adversarial information by querying legitimate peers or seeders.

3.2 Restarting the victim

Our attack requires the victim to restart so it can connect to adversarial addresses. There are several reasons why a bitcoin node could restart, including ISP outages, power failures, and upgrades, failures or attacks on the host OS; indeed, [16] found that a node with a public IP has a 25% chance of going offline after 10 hours. Another predictable reason to restart is a software update; on 1/10/2014, for example, bitnodes.io saw 942 nodes running Satoshi client version 0.9.3, and by 29/12/2014, that number had risen to 3018 nodes, corresponding to over 2000 restarts. Since updating is often *not* optional, especially when it corresponds to critical security issues; 2013 saw three such bitcoin upgrades, and the heartbleed bug [53] caused one in 2014. Also, since the community needs to be notified about an upgrade in advance, the attacker could watch for notifications and then commence its attack [2]. Restarts can also be deliberately elicited via DDoS [47, 65], memory exhaustion [16], or packets-of-death (which have been found for bitcoind [6,7]). The bottom line is that the security of the peer-to-peer network should not rely on 100% node uptime.

3.3 Selecting outgoing connections

Our attack succeeds if, upon restart, the victim makes all its outgoing connections to attacker addresses. To do this, we exploit the bias towards selecting addresses with fresh timestamps from tried; by investing extra time into the attack, our attacker ensures its addresses are fresh, while all legitimate addresses become increasingly stale. We analyze this with few simple assumptions:

1. An f-fraction of the addresses in the victim's tried table are controlled by the adversary and the remaining $1-f$-fraction are legitimate. (Section 4 analyzes how many addresses the adversary therefore must control.)

2. *All* addresses in new are "trash"; all connections to addresses in new fail, and the victim is forced to connect to addresses from tried (Section 2.3).

3. The attack proceeds in *rounds*, and repeats each round until the moment that the victim restarts. During a single round, the attacker connects to the victim from each of its adversarial IP addresses. A round takes time τ_a, so all adversarial addresses in tried are younger than τ_a.

4. An f'-fraction addresses in tried are actively connected to the victim before the victim restarts. The timestamps on these legitimate addresses are updated every 20 minute or more (Section 2.2). We assume these timestamps are fresh (*i.e.,* $\tau = 0$) when the victim restarts; this is the worst case for the attacker.

5. The *time invested in the attack* τ_ℓ is the time elapsed from the moment the adversary starts the attack, until the victim restarts. If the victim did not obtain new legitimate network information during of the attack, then, excluding the f'-fraction described above, the legitimate addresses in tried are older than τ_ℓ.

Success probability. If the adversary owns an f-fraction of the addresses in tried, the probability that an adversarial address is accepted on the first try is $p(1,\tau_a)\cdot f$ where $p(1,\tau_a)$ is as in equation (2); here we use the fact that the adversary's addresses are no older than τ_a, the length of the round. If $r-1$ addresses were rejected during this attempt to select an address from tried, then the probability that an adversarial address is accepted on the r^{th} try is bounded by

$$p(r,\tau_a)\cdot f\prod_{i=1}^{r-1} g(i,f,f',\tau_a,\tau_\ell)$$

where

$$\begin{aligned} g(i,f,f',\tau_a,\tau_\ell) = {} & (1-p(i,\tau_a))\cdot f + (1-p(i,0))\cdot f' \\ & + (1-p(i,\tau_\ell))\cdot(1-f-f') \end{aligned}$$

is the probability that an address was rejected on the i^{th} try given that it was also rejected on the $i-1^{th}$ try. An adversarial address is thus accepted with probability

$$q(f,f',\tau_a,\tau_\ell) = \sum_{r=1}^{\infty} p(r,\tau_a)\cdot f\prod_{i=1}^{r-1} g(i,f,f',\tau_a,\tau_\ell) \tag{3}$$

and the victim is eclipsed if all eight outgoing connections are to adversarial addresses, which happens with probability $q(f,f',\tau_a,\tau_\ell)^8$. Figure 1 plots $q(f,f',\tau_a,\tau_\ell)^8$ vs f for $\tau_a = 27$ minutes and different choices of τ_ℓ; we assume that $f' = \frac{8}{64\times 64}$, which corresponds to a full tried table containing eight addresses that are actively connected before the victim restarts.

Random selection. Figure 1 also shows success probability if addresses were just selected uniformly at random from each table. We do this by plotting f^8 vs f. Without random selection, the adversary has a 90% success probability even if it only fills $f = 72\%$ of tried, as long as it attacks for $\tau_\ell = 48$ hours with $\tau_a = 27$ minute rounds. With random selection, 90% success probability requires $f = 98.7\%$ of tried to be attacker addresses.

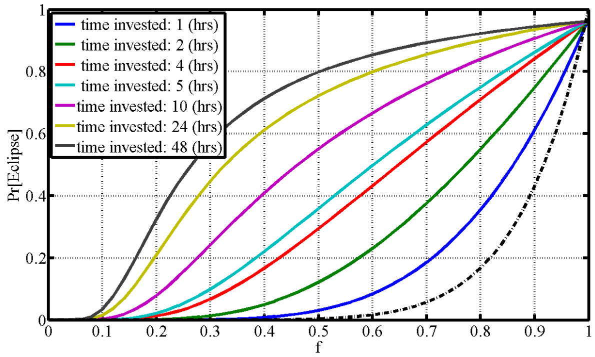


Figure 1: Probability of eclipsing a node $q(f, f', \tau_a, \tau_\ell)^8$ (equation (3)) vs f the fraction of adversarial addresses in tried, for different values of time invested in the attack τ_ℓ. Round length is $\tau_a = 27$ minutes, and $f' = \frac{8}{64\times 64}$. The dotted line shows the probability of eclipsing a node if random selection is used instead.

3.4 Monopolizing the eclipsed victim

Figure 1 assumes that the victim has exactly eight *outgoing connections*; all we require in terms of *incoming connections* is that the victim has a few open slots to accept incoming TCP connections from the attacker.

While it is often assumed that the number of TCP connections a computer can make is limited by the OS or the number of source ports, this applies only when OS-provided TCP sockets are used; a dedicated attacker can open an arbitrary number of TCP connections using a custom TCP stack. A custom TCP stack (see *e.g.,* zmap [35]) requires minimal CPU and memory, and is typically bottlenecked only by bandwidth, and the bandwidth cost of our attack is minimal:

Attack connections. To fill the tried table, our attacker repeatedly connects to the victim from each of its addresses. Each connection consists of a TCP handshake, bitcoin VERSION message, and then disconnection via TCP RST; this costs 371 bytes upstream and 377 bytes downstream. Some attack connections also send one ADDR message containing 1000 addresses; these ADDR messages cost 120087 bytes upstream and 437 bytes downstream including TCP ACKs.

Monopolizing connections. If that attack succeeds, the victim has eight outgoing connections to the attack nodes, and the attacker must occupy the victim's remaining incoming connections. To prevent others from connecting to the victim, these TCP connections could be maintained for 30 days, at which point the victim's address is terrible and forgotten by the network. While bitcoin supports block inventory requests and the sending of blocks and transactions, this consumes significant bandwidth; our attacker thus does not to respond to inventory requests. As such, setting up each TCP connection costs 377 bytes upstream and 377 bytes downstream, and is maintained by ping-pong packets and TCP ACKs consuming 164 bytes every 80 minutes.

We experimentally confirmed that a bitcoin node will accept all incoming connections from the same IP address. (We presume this is done to allow multiple nodes behind a NAT to connect to the same node.) Maintaining the default 117 incoming TCP connections costs $\frac{164\times 117}{80\times 60} \approx 4$ bytes per second, easily allowing one computer to monopolize multiple victims at the same time. As an aside, this also allows for *connection starvation attacks* [32], where an attacker monopolizes all the incoming connections in the peer-to-peer network, making it impossible for new nodes to connect to new peers.

4 How Many Attack Addresses?

Section 3.3 showed that the success of our attack depends heavily on τ_ℓ, the time invested in the attack, and f, the fraction of attacker addresses in the victim's tried table. We now use probabilistic analysis to determine how many addresses the attacker must control for a given value of f; it's important to remember, however, that even if f is small, our attacker can still succeed by increasing τ_ℓ. Recall from Section 2.2 that bitcoin is careful to ensure that a node does not store too many IP addresses from the same *group* (*i.e.,* /16 IPv4 address block). We therefore consider two attack variants:

Botnet attack (Section 4.1). The attacker holds several IP addresses, each in a *distinct* group. This models attacks by a botnet of hosts scattered in diverse IP address blocks. Section 4.1.1 explains why many botnets have enough IP address diversity for this attack.

Infrastructure attack (Section 4.2). The attacker controls several IP address blocks, and can intercept bitcoin traffic sent to any IP address in the block, *i.e.,* the attacker holds multiple sets of addresses in the same *group*. This models a company or nation-state that seeks to undermine bitcoin by attacking its network. Section 4.2.1 discusses organizations that can launch this attack.

We focus here on tried; Appendix B considers how to send "trash"-filled ADDR messages that overwrite new.

4.1 Botnet attack

The botnet attacker holds t addresses in distinct groups. We model each address as hashing to a uniformly-random bucket in tried, so the number of addresses hashing to each bucket is binomally distributed[3] as $B(t, \frac{1}{64})$. How many of the 64×64 entries in tried

[3] $B(n, p)$ is a binomial distribution counting successes in a sequence of n independent yes/no trials, each yielding 'yes' with probability p.

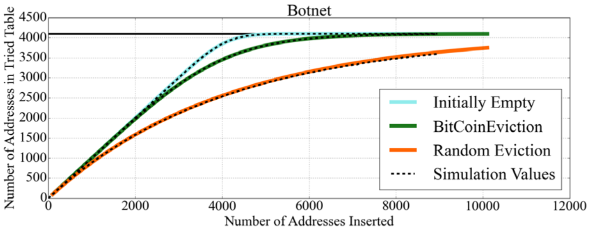


Figure 2: Botnet attack: the expected number of addresses stored in `tried` for different scenarios vs the number of addresses (bots) t. Values were computed from equations (4), (7) and (8), and confirmed by Monte Carlo simulations (with 100 trials/data point).

can the attacker occupy? We model various scenarios, and plot results in Figure 2.

1. Initially empty. In the best case for the attacker, all 64 buckets are initially empty and the expected number of adversarial addresses stored in the `tried` table is

$$64E[\min(64, B(t, \tfrac{1}{64}))] \qquad (4)$$

2. Bitcoin eviction. Now consider the worst case for the attacker, where each bucket i is full of 64 legitimate addresses. These addresses, however, will be *older* than all A_i distinct adversarial addresses that the adversary attempts to insert into to bucket i. Since the bitcoin eviction discipline requires each newly inserted address to select four random addresses stored in the bucket and to evict the oldest, if one of the four selected addresses is a legitimate address (which will be older than all of the adversary's addresses), the legitimate address will be overwritten by the adversarial addresses.

For $a = 0....A_i$, let Y_a be the number of adversarial addresses actually stored in bucket i, given that the adversary inserted a unique addresses into bucket i. Let $X_a = 1$ if the a^{th} inserted address successfully overwrites a legitimate address, and $X_a = 0$ otherwise. Then,

$$E[X_a|Y_{a-1}] = 1 - (\tfrac{Y_{a-1}}{64})^4$$

and it follows that

$$E[Y_a|Y_{a-1}] = Y_{a-1} + 1 - (\tfrac{Y_{a-1}}{64})^4 \qquad (5)$$

$$E[Y_1] = 1 \qquad (6)$$

where (6) follows because the bucket is initially full of legitimate addresses. We now have a recurrence relation for $E[Y_a]$, which we can solve numerically. The expected number of adversarial addresses in all buckets is thus

$$64 \sum_{a=1}^{t} E[Y_a] \Pr[B(t, \tfrac{1}{64}) = a] \qquad (7)$$

3. Random eviction. We again consider the attacker's worst case, where each bucket is full of legitimate addresses, but now we assume that each inserted address evicts a randomly-selected address. (This is not what bitcoin does, but we analyze it for comparison.) Applying Lemma A.1 (Appendix A) we find the expected number of adversarial addresses in all buckets is

$$4096(1 - (\tfrac{4095}{4096})^t) \qquad (8)$$

4. Exploiting multiple rounds. Our eclipse attack proceeds in *rounds*; in each round the attacker repeatedly inserts each of his t addresses into the `tried` table. While each address always maps to the same bucket in `tried` in each round, bitcoin eviction maps each address to a *different slot* in that bucket in every round. Thus, an adversarial address that is not stored into its `tried` bucket at the end of one round, might still be successfully stored into that bucket in a future round. Thus far, this section has only considered a single round. But, more addresses can be stored in `tried` by repeating the attack for multiple rounds. After sufficient rounds, the expected number of addresses is given by equation (4), *i.e.,* the attack performs as in the best-case for the attacker!

4.1.1 Who can launch a botnet attack?

The 'initially empty' line in Figure 2 indicates that a botnet exploiting multiple rounds can completely fill `tried` with ≈ 6000 addresses. While such an attack cannot easily be launched from a legitimate cloud service (which typically allocates < 20 addresses per tenant [1, 8, 9]), botnets of this size and larger than this have attacked bitcoin [45, 47, 65]; the Miner botnet, for example, had 29,000 hosts with public IPs [54]. While some botnet infestations concentrate in a few IP address ranges [63], it is important to remember that our botnet attack requires no more than ≈ 6000 groups; many botnets are orders of magnitude larger [59]. For example, the Walowdac botnet was mostly in ranges 58.x-100.x and 188.x-233.x [63], which creates $42 \times 2^8 + 55 \times 2^8 = 24832$ groups. Randomly sampling from the list of hosts in the Carna botnet [26] 5000 times, we find that 1250 bots gives on average 402 distinct groups, enough to attack our live bitcoin nodes (Section 6). Furthermore, we soon show in Figure 3 that an infrastructure attack with $s > 200$ groups easily fills every bucket in `tried`; thus, with $s > 400$ groups, the attack performs as in Figure 2, even if many bots are in the same group. .

4.2 Infrastructure attack

The attacker holds addresses in s distinct *groups*. We determine how much of `tried` can be filled by an attacker controlling s groups s containing t IP addresses/group.

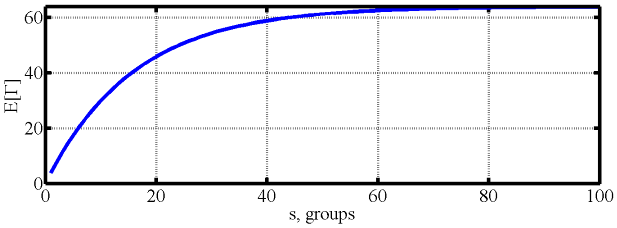


Figure 3: Infrastructure attack. $E[\Gamma]$ (expected number of non-empty buckets) in `tried` vs s (number of groups).

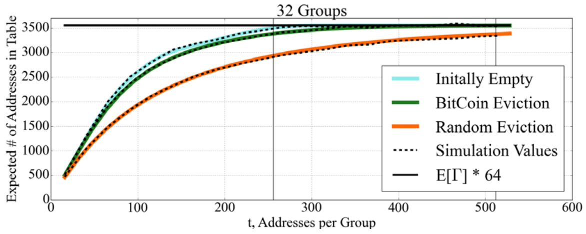


Figure 4: Infrastructure attack with $s = 32$ groups: the expected number of addresses stored in `tried` for different scenarios vs the number of addresses per group t. Results obtained by taking the product of equation (9) and equations from the full version [41], and confirmed by Monte Carlo simulations (100 trials/data point). The horizontal line assumes all $E[\Gamma]$ buckets per (9) are full.

How many groups? We model the process of populating `tried` (per Section 2.2) by supposing that four independent hash functions map each of the s groups to one of 64 buckets in `tried`. Thus, if $\Gamma \in [0,64]$ counts the number of non-empty buckets in `tried`, we use Lemma A.1 to find that

$$E[\Gamma] = 64\left(1-\left(\tfrac{63}{64}\right)^{4s}\right) \approx \left(1-e^{-\frac{4s}{64}}\right) \qquad (9)$$

Figure 3 plots $E[\Gamma]$; we expect to fill 55.5 of 64 buckets with $s = 32$, and all but one bucket with $s > 67$ groups.

How full is the `tried` table? The full version [41] determines the expected number of addresses stored per bucket for the first three scenarios described in Section 4.1; the expected fraction $E[f]$ of `tried` filled by adversarial addresses is plotted in in Figure 4. The horizontal line in Figure 4 show what happens if each of $E[\Gamma]$ buckets per equation (9) is full of attack addresses.

The adversary's task is easiest when all buckets are initially empty, or when a sufficient number of rounds are used; a single /24 address block of 256 addresses suffices to fill each bucket when $s = 32$ grouips is used. Moreover, as in Section 4.1, an attack that exploits multiple rounds performs as in the 'initially empty' scenario. Concretely, with 32 groups of 256 addresses each (8192 addresses in total) an adversary can expect to fill about $f = 86\%$ of the `tried` table after a sufficient number of rounds. The attacker is almost as effective in the bitcoin-eviction scenario with only one round; meanwhile, one round is much less effective with random eviction.

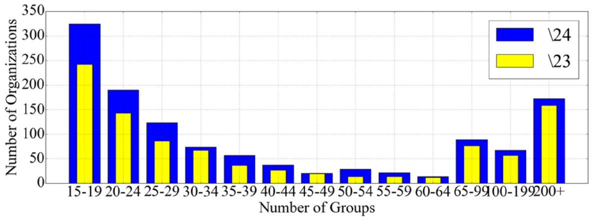


Figure 5: Histogram of the number of organizations with s groups. For the /24 data, we require $t = 256$ addresses per group; for /23, we require $t = 512$.

4.2.1 Who can launch an infrastructure attack?

Which organizations have enough IP address resources to launch infrastructure attacks? We compiled data mapping IPv4 address allocation to organizations, using CAIDA's AS to organization dataset [23] and AS to prefix dataset [24] from July 2014, supplementing our data with information from the RIPE database [55]. We determined how many groups (*i.e.,* addresses in the same /16 IPv4 address block) and addresses per group are allocated to each organization; see Figure 5. There are 448 organizations with over $s = 32$ groups and at least $t = 256$ addresses per group; if these organizations invest $\tau_\ell = 5$ hours into an attack with a $\tau_a = 27$-minute round, then they eclipse the victim with probability greater than 80%.

National ISPs in various countries hold a sufficient number of groups ($s \geq 32$) for this purpose; for example, in Sudan (Sudanese Mobile), Columbia (ETB), UAE (Etisalat), Guatemala (Telgua), Tunisia (Tunisia Telecom), Saudi Arabia (Saudi Telecom Company) and Dominica (Cable and Wireless). The United States Department of the Interior has enough groups ($s = 35$), as does the S. Korean Ministry of Information and Communication ($s = 41$), as do hundreds of others.

4.3 Summary: infrastructure or botnet?

Figures 4, 2 show that the botnet attack is far superior to the infrastructure attack. Filling $f = 98\%$ of the victim's `tried` table requires a 4600 node botnet (attacking for a sufficient number of rounds, per equation (4)). By contrast, an infrastructure attacker needs 16,000 addresses, consisting of $s = 63$ groups (equation (9)) with $t = 256$ addresses per group. However, per Section 3.3, if our attacker increases the time invested in the attack τ_ℓ, it can be far less aggressive about filling `tried`. For example, per Figure 1, attacking for $\tau_\ell = 24$ hours with $\tau_a = 27$ minute rounds, our success probability exceeds

oldest	#		Age of addresses (in days)				
addr	addr	% live	<1	$1-5$	$5-10$	$10-30$	>30
38 d*	243	28%	36	71	28	79	29
41 d*	162	28%	23	29	27	44	39
42 d*	244	19%	25	45	29	95	50
42 d*	195	23%	23	40	23	64	45
43 d*	219	20%	66	57	23	50	23
103 d	4096	8%	722	645	236	819	1674
127 d	4096	8%	90	290	328	897	2491
271 d	4096	8%	750	693	356	809	1488
240 d	4096	6%	419	445	32	79	3121
373 d	4096	5%	9	14	1	216	3856

Table 1: Age and churn of addresses in `tried` for our nodes (marked with *) and donated peers files.

85% with just $f = 72\%$; in the worst case for the attacker, this requires only 3000 bots, or an infrastructure attack of $s = 20$ groups and $t = 256$ addresses per group (5120 addresses). The same attack ($f = 72\%$, $\tau_a = 27$ minutes) running for just 4 hours still has $> 55\%$ success probability. To put this in context, if 3000 bots joined today's network (with < 7200 public-IP nodes [4]) and honestly followed the peer-to-peer protocol, they could eclipse a victim with probability $\approx (\frac{3000}{7200+3000})^8 = 0.006\%$.

5 Measuring Live Bitcoin Nodes

We briefly consider how parameters affecting the success of our eclipse attacks look on "typical" bitcoin nodes. We thus instrumented five bitcoin nodes with public IPs that we ran (continuously, without restarting) for 43 days from 12/23/2014 to 2/4/2015. We also analyze several peers files that others donated to us on 2/15/2015. Note that there is evidence of wide variations in metrics for nodes of different ages and in different regions [46]; as such, our analysis (Section 3-4) and some of our experiments (Section 6) focus on the attacker's worst-case scenario, where tables are initially full of fresh addresses.

Number of connections. Our attack requires the victim to have available slots for incoming connections. Figure 6 shows the number of connections over time for one of our bitcoin nodes, broken out by connections to public or private IPs. There are plenty of available slots; while our node can accommodate 125 connections, we never see more than 60 at a time. Similar measurements in [17] indicate that 80% of bitcoin peers allow at least 40 incoming connections. Our node saw, on average, 9.9 connections to public IPs over the course of its lifetime; of these, 8 correspond to *outgoing* connections, which means we rarely see incoming connections from public IPs. Results for our other nodes are similar.

Connection length. Because public bitcoin nodes rarely drop outgoing connections to their peers (except upon restart, network failure, or due to blacklisting, see Section 2.3), many connections are fairly long lived. When we sampled our nodes on 2/4/2015, across all of our nodes, 17% of connections had lasted more than 15 days, and of these, 65.6% were to public IPs. On the other hand, many bitcoin nodes restart frequently; we saw that 43% of connections lasted less than two days and of these, 97% were to nodes with private IPs. This may explain why we see so few incoming connections from public IPs; many public-IP nodes stick to their mature long-term peers, rather than our young-ish nodes.

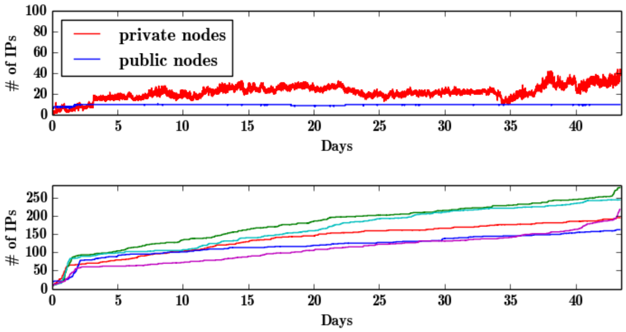


Figure 6: (Top) Incoming + outgoing connections vs time for one of our nodes. (Bottom) Number of addresses in `tried` vs time for all our nodes.

Size of `tried` and `new` tables. In our worst case attack, we supposed that the `tried` and `new` tables were completely full of fresh addresses. While our Bitcoin nodes' `new` tables filled up quite quickly (99% within 48 hours), Table 1 reveals that their `tried` tables were far from full of fresh addresses. Even after 43 days, the `tried` tables for our nodes were no more than $300/4096 \approx 8\%$ full. This likely follows because our nodes had very few incoming connections from public IPs; thus, most addresses in `tried` result from successful outgoing connections to public IPs (infrequently) drawn from `new`.

Freshness of `tried`. Even those few addresses in `tried` are not especially fresh. Table 1 shows the age distribution of the addresses in `tried` for our nodes and from donated peers files. For our nodes, 17% of addresses were more than 30 days old, and 48% were more than 10 days old; these addresses will therefore be less preferred than the adversarial ones inserted during an eclipse attack, even if the adversary does not invest much time τ_ℓ in attacking the victim.

Churn. Table 1 also shows that a small fraction of addresses in `tried` were online when we tried connecting to them on 2/17/2015.[4] This suggests further vulnerability to eclipse attacks, because if most legitimate addresses in `tried` are offline when a victim resets, the victim is likely to connect to an adversarial address.

[4] For consistency with the rest of this section, we tested our nodes tables from 2/4/2015. We also repeated this test for tables taken from our nodes on 2/17/2015, and the results did not deviate more than 6% from those of Table 1.

	Attacker resources					Experiment							Predicted		
	grps	addrs/	total	τ_ℓ, time	τ_a,	Total pre-attack		Total post-attack		Attack addrs			Attack addrs		
Attack Type	s	grp t	addrs	invest	round	`new`	`tried`	`new`	`tried`	`new`	`tried`	**Wins**	`new`	`tried`	**Wins**
Infra (Worstcase)	32	256	8192	10 h	43 m	16384	4090	16384	4096	15871	3404	**98%**	16064	3501	**87%**
Infra (Transplant)	20	256	5120	1 hr	27 m	16380	278	16383	3087	14974	2947	**82%**	15040	2868	**77%**
Infra (Transplant)	20	256	5120	2 hr	27 m	16380	278	16383	3088	14920	2966	**78%**	15040	2868	**87%**
Infra (Transplant)	20	256	5120	4 hr	27 m	16380	278	16384	3088	14819	2972	**86%**	15040	2868	**91%**
Infra (Live)	20	256	5120	1 hr	27 m	16381	346	16384	3116	14341	2942	**84%**	15040	2868	**75%**
Bots (Worstcase)	2300	2	4600	5 h	26 m	16080	4093	16384	4096	16383	4015	**100%**	16384	4048	**96%**
Bots (Transplant)	200	1	200	1 hr	74 s	16380	278	16384	448	16375	200	**60%**	16384	200	**11%**
Bots (Transplant)	400	1	400	1 hr	90 s	16380	278	16384	648	16384	400	**88%**	16384	400	**34%**
Bots (Transplant)	400	1	400	4 hr	90 s	16380	278	16384	650	16383	400	**84%**	16384	400	**61%**
Bots (Transplant)	600	1	600	1 hr	209 s	16380	278	16384	848	16384	600	**96%**	16384	600	**47%**
Bots (Live)	400	1	400	1 hr	90 s	16380	298	16384	698	16384	400	**84%**	16384	400	**28%**

Table 2: Summary of our experiments.

6 Experiments

We now validate our analysis with experiments.

Methodology. In each of our experiments, the victim (bitcoind) node is on a virtual machine on the attacking machine; we also instrument the victim's code. The victim node runs on the public bitcoin network (*aka,* mainnet). The attacking machine can read all the victim's packets to/from the public bitcoin network, and can therefore forge TCP connections from arbitrary IP addresses. To launch the attack, the attacking machine forges TCP connections from each of its attacker addresses, making an incoming connection to the victim, sending a `VERSION` message and sometimes also an `ADDR` message (per Appendix B) and then disconnecting; the attack connections, which are launched at regular intervals, rarely occupy all of the victim's available slots for incoming connections. To avoid harming the public bitcoin network, (1) we use "reserved for future use" [43] IPs in 240.0.0.0/8-249.0.0.0/8 as attack addresses, and 252.0.0.0/8 as "trash" sent in `ADDR` messages, and (2) we drop any `ADDR` messages the (polluted) victim attempts to send to the public network.

At the end of the attack, we repeatedly restart the victim and see what outgoing connections it makes, dropping connections to the "trash" addresses and forging connections for the attacker addresses. If all 8 outgoing connections are to attacker addresses, the attack succeeds, and otherwise it fails. Each experiment restarts the victim 50 times, and reports the fraction of successes. At each restart, we revert the victim's tables to their state at the end of the attack, and rewind the victim's system time to the moment the attack ended (to avoid dating timestamps in `tried` and `new`). We restart the victim 50 times to measure the success rate of our (probabilistic) attack; in a real attack, the victim would only restart once.

Initial conditions. We try various initial conditions:

1. Worst case. In the attacker's worst-case scenario, the victim initially has `tried` and `new` tables that are completely full of legitimate addresses with fresh timestamps. To set up the initial condition, we run our attack for no longer than one hour on a freshly-born victim node, filling `tried` and `new` with IP addresses from 251.0.0.0/8, 253.0.0.0/8 and 254.0.0.0/8, which we designate as "legitimate addresses"; these addresses are no older than one hour when the attack starts. We then restart the victim and commence attacking it.

2. Transplant case. In our transplant experiments, we copied the `tried` and `new` tables from one of our five live bitcoin nodes on 8/2/2015, installed them in a fresh victim with a different public IP address, restarted the victim, waited for it to establish eight outgoing connections, and then commenced attacking. This allowed us to try various attacks with a consistent initial condition.

3. Live case. Finally, on 2/17/2015 and 2/18/2015 we attacked our live bitcoin nodes while they were connected to the public bitcoin network; at this point our nodes had been online for 52 or 53 days.

Results (Table 2). Results are in Table 2. The first five columns summarize attacker resources (the number of groups s, addresses per group t, time invested in the attack τ_ℓ, and length of a round τ_a per Sections 3-4). The next two columns present the initial condition: the number of addresses in `tried` and `new` prior to the attack. The following four columns give the size of `tried` and `new`, and the number of attacker addresses they store, at the end of the attack (when the victim first restarts). The *wins* columns counts the fraction of times our attack succeeds after restarting the victim 50 times.

The final three columns give predictions from Sections 3.3, 4. The *attack addrs* columns give the expected number of addresses in `new` (Appendix B) and `tried`. For `tried`, we assume that the attacker runs his attack for enough rounds so that the expected number of addresses in `tried` is governed by equation (4) for the botnet, and the 'initially empty' curve of Figure 4 for the infrastructure attack. The final column predicts success per Section 3.3 using *experimental values* of τ_a, τ_ℓ, f, f'.

Observations. Our results indicate the following:

1. Success in worst case. Our experiments confirm that an infrastructure attack with 32 groups of size /24 (8192 attack addresses total) succeeds in the worst case with very high probability. We also confirm that botnets are superior to infrastructure attacks; 4600 bots had 100% success even with a worst-case initial condition.

2. Accuracy of predictions. Almost all of our attacks had an experimental success rate that was *higher* than the predicted success rate. To explain this, recall that our predictions from Section 3.3 assume that legitimate addresses are exactly τ_ℓ old (where τ_ℓ is the time invested in the attack); in practice, legitimate addresses are likely to be even older, especially when we work with `tried` tables of real nodes (Table 1). Thus, Section 3.3's predictions are a lower bound on the success rate.

Our experimental botnet attacks were dramatically more successful than their predictions (*e.g.,* 88% actual vs. 34% predicted), most likely because the addresses initially in `tried` were already very stale prior to the attack (Table 1). Our infrastructure attacks were also more successful then their predictions, but here the difference was much less dramatic. To explain this, we look to the `new` table. While our success-rate predictions assume that `new` is completely overwritten, our infrastructure attacks failed to completely overwrite the `new` table;[5] thus, we have some extra failures because the victim made outgoing connections to addresses in `new`.

3. Success in a 'typical' case. Our attacks are successful with even fewer addresses when we test them on our live nodes, or on tables taken from those live nodes. Most strikingly, a small botnet of 400 bots succeeds with very high probability; while this botnet completely overwrites `new`, it fills only $400/650 = 62\%$ of `tried`, and still manages to win with more than 80% probability.

7 Countermeasures

We have shown how an attacker with enough IP addresses and time can eclipse any target victim, regardless of the state of the victim's `tried` and `new` tables. We now present countermeasures that make eclipse attacks more difficult. Our countermeasures are inspired by botnet architectures (Section 8), and designed to be faithful to bitcoin's network architecture.

The following five countermeasures ensure that: (1) If the victim has h legitimate addresses in `tried` before the attack, and a p-fraction of them accept incoming connections during the attack when the victim restarts, then even an attacker *with an unbounded number of addresses* cannot eclipse the victim with probability exceeding equation (10). (2) If the victim's oldest outgoing connection is to a legitimate peer before the attack, then the eclipse attack *fails* if that peer accepts incoming connections when the victim restarts.

1. Deterministic random eviction. Replace bitcoin eviction as follows: just as each address deterministically hashes to a single bucket in `tried` and `new` (Section 2.2), an address also deterministically hashes to a single slot in that bucket. This way, an attacker cannot increase the number of addresses stored by repeatedly inserting the same address in multiple rounds (Section 4.1). Instead, addresses stored in `tried` are given by the 'random eviction' curves in Figures 2, 4, reducing the attack addresses stored in `tried`.

2. Random selection. Our attacks also exploit the heavy bias towards forming outgoing connections to addresses with fresh timestamps, so that an attacker that owns only a small fraction $f = 30\%$ of the victim's `tried` table can increase its success probability (to say 50%) by increasing τ_ℓ, the time it invests in the attack (Section 3.3). We can eliminate this advantage for the attacker if addresses are selected at random from `tried` and `new`; this way, a success rate of 50% always requires the adversary to fill $\sqrt[8]{0.5} = 91.7\%$ of `tried`, which requires 40 groups in an infrastructure attack, or about 3680 peers in a botnet attack. Combining this with deterministic random eviction, the figure jumps to 10194 bots for 50% success probability.

These countermeasures harden the network, but still allow an attacker with enough addresses to overwrite all of `tried`. The next countermeasure remedies this:

3. Test before evict. Before storing an address in its (deterministically-chosen) slot in a bucket in `tried`, first check if there is an older address stored in that slot. If so, briefly attempt to connect to the older address, and if connection is successful, then the older address is *not* evicted from the `tried` table; the new address is stored in `tried` only if the connection fails.

We analyze these three countermeasures. Suppose that there are h legitimate addresses in the `tried` table prior to the attack, and model network churn by supposing that each of the h legitimate addresses in `tried` is live (*i.e.,* accepts incoming connections) independently with probability p. With test-before-evict, the adversary cannot evict $p \times h$ legitimate addresses (in expectation) from `tried`, regardless of the number of distinct addresses it controls. Thus, even if the rest of `tried` is full of adversarial addresses, the probability of eclipsing the victim is bounded to about

$$\Pr[\text{eclipse}] = f^8 < \left(1 - \tfrac{p \times h}{64 \times 64}\right)^8 \tag{10}$$

This is in stark contrast to today's protocol, where attackers with enough addresses have *unbounded* success probability even if `tried` is *full* of legitimate addresses.

[5] The `new` table holds 16384 addresses and from 6^{th} last column of Table 2 we see the `new` is not full for our infrastructure attacks. Indeed, we predict this in Appendix B.

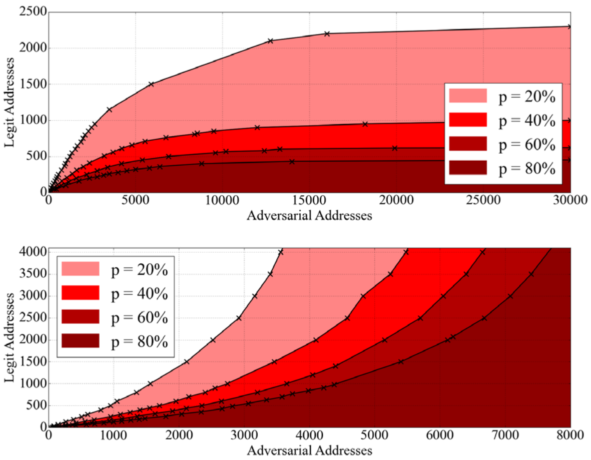


Figure 7: The area below each curve corresponds to a number of bots a that can eclipse a victim with probability at least 50%, given that the victim initially has h legitimate addresses in tried. We show one curve per churn rate p. (Top) With test before evict. (Bottom) Without.

We perform Monte-Carlo simulations assuming churn p, h legitimate addresses initially stored in tried, and a botnet inserting a addresses into tried via unsolicited incoming connections. The area below each curve in Figure 7 is the number of bots a that can eclipse a victim with probability at least 50%, given that there are initially h legitimate addresses in tried. With test-before-evict, the curves plateau horizontally at $h = 4096(1-\sqrt[8]{0.5})/p$; as long as h is greater than this quantity, even a botnet *with an infinite number of addresses* has success probability bounded by 50%. Importantly, the plateau is absent without test-before-evict; a botnet with enough addresses can eclipse a victim *regardless* of the number of legitimate addresses h initially in tried.

There is one problem, however. Our bitcoin nodes saw high churn rates (Table 1). With a $p = 28\%$ churn rate, for example, bounding the adversary's success probability to 10% requires about $h = 3700$ addresses in tried; our nodes had $h < 400$. Our next countermeasure thus adds more legitimate addresses to tried:

4. Feeler Connections. Add an outgoing connection that establish short-lived test connections to randomly-selected addresses in new. If connection succeeds, the address is evicted from new and inserted into tried; otherwise, the address is evicted from new.

Feeler connections clean trash out of new while increasing the number of fresh address in tried that are likely to be online when a node restarts. Our fifth countermeasure is orthogonal to those above:

5. Anchor connections. Inspired by Tor entry guard rotation rates [33], we add two connections that persist between restarts. Thus, we add an anchor table, recording addresses of current outgoing connections and the time of first connection to each address. Upon restart, the node dedicates two extra outgoing connections to the oldest anchor addresses that accept incoming connections. Now, in addition to defeating our other countermeasures, a successful attacker must also disrupt anchor connections; eclipse attacks fail if the victim connects to an anchor address not controlled by the attacker.

Apart from these five countermeasures, a few other ideas can raise the bar for eclipse attacks:

6. More buckets. Among the most obvious countermeasure is to increase the size of the tried and new tables. Suppose we doubled the number of buckets in the tried table. If we consider the infrastructure attack, the buckets filled by s groups jumps from $(1-e^{-\frac{4s}{64}})$ (per equation (9) to $(1-e^{-\frac{4s}{128}})$. Thus, an infrastructure attacker needs double the number of groups in order to expect to fill the same fraction of tried. Similarly, a botnet needs to double the number of bots. Importantly, however, this countermeasure is helpful only when tried already contains many legitimate addresses, so that attacker owns a smaller fraction of the addresses in tried. However, if tried is mostly empty (or contains mostly stale addresses for nodes that are no longer online), the attacker will still own a large fraction of the addresses in tried, even though the number of tried buckets has increased. Thus, this countermeasure should also be accompanied by another countermeasure (*e.g.,* feeler connections) that increases the number of legitimate addresses stored in tried.

7. More outgoing connections. Figure 6 indicates our test bitcoin nodes had at least 65 connections slots available, and [17] indicates that 80% of bitcoin peers allow at least 40 incoming connections. Thus, we can require nodes to make a few additional outgoing connections without risking that the network will run out of connection capacity. Indeed, recent measurements [51] indicate that certain nodes (*e.g.,* mining-pool gateways) do this already. For example, using twelve outgoing connections instead of eight (in addition to the feeler connection and two anchor connections), decreases the attack's success probability from f^8 to f^{12}; to achieve 50% success probability the infrastructure attacker now needs 46 groups, and the botnet needs 11796 bots.

8. Ban unsolicited ADDR messages. A node could choose not to accept large unsolicited ADDR messages (with > 10 addresses) from incoming peers, and only solicit ADDR messages from outgoing connections when its new table is too empty. This prevents adversarial incoming connections from flooding a victim's new table with trash addresses. We argue that this change is not harmful, since even in the current network, there is no shortage of address in the new table (Section 5). To make this more concrete, note that a node request ADDR messages upon establishing an outgoing connection. The peer responds with n randomly selected addresses from its tried and new tables, where n is a random number between x and 2500 and x is 23% of the addresses the peer has stored. If each peer sends, say, about $n = 1700$ addresses, then new is already $8n/16384 = 83\%$ full the moment that the bitcoin node finishing establishing outgoing connections.

9. Diversify incoming connections. Today, a bitcoin node can have all of its incoming connections come from the same IP address, making it far too easy for a single computer to monopolize a victim's incoming connections during an eclipse attack or connection-starvation attack [32]. We suggest a node accept only a limited number of connections from the same IP address.

10. Anomaly detection. Our attack has several specific "signatures" that make it detectable including: (1) a flurry of short-lived incoming TCP connections from diverse IP addresses, that send (2) large ADDR messages (3) containing "trash" IP addresses. An attacker that suddenly connects a large number of nodes to the bitcoin network could also be detected, as could one that uses eclipsing per Section 1.1 to dramatically decrease the network's mining power. Thus, monitoring and anomaly detection systems that look for this behavior are also be useful; at the very least, they would force an eclipse attacker to attack at low rate, or to waste resources on overwriting new (instead of using "trash" IP addresses).

Status of our countermeasures. We disclosed our results to the bitcoin core developers in 02/2015. They deployed Countermeasures 1, 2, and 6 in the bitcoind v0.10.1 release, which now uses deterministic random eviction, random selection, and scales up the number of buckets in tried and new by a factor of four. To illustrate the efficacy of this, consider the worst-case scenario for the attacker where tried is completely full of legitimate addresses. We use Lemma A.1 to estimate the success rate of a botnet with t IP addresses as

$$\Pr[\text{Eclipse}] \approx \left(1 - (\tfrac{16383}{16384})^t\right)^8 \qquad (11)$$

Plotting (11) in Figure 8, we see that this botnet requires 163K addresses for a 50% success rate, and 284K address for a 90% success rate. This is good news, but we caution that ensuring that tried is full of legitimate address is still a challenge (Section 5), especially since there may be fewer than 16384 public-IP nodes in the bitcoin network at a given time. Countermeasures 3 and 4 are designed to deal with this, and so we have also developed a patch with these two countermeasures; see [40] for our implementation and its documentation.

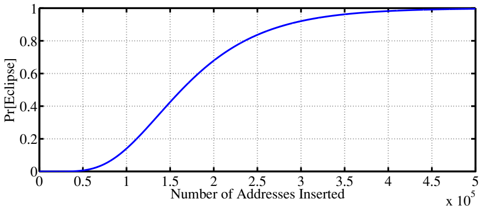


Figure 8: Probability of eclipsing a node vs the number of addresses (bots) t for bitcoind v0.10.1 (with Countermeasures 1,2 and 6) when tried is initially full of legitimate addresses per equation (11).

8 Related Work

The bitcoin peer-to-peer (p2p) network. Recent work considers how bitcoin's network can delay or prevent block propagation [31] or be used to deanonymize bitcoin users [16, 17, 48]. These works discuss aspects of bitcoin's networking protocol, with [16] providing an excellent description of ADDR message propagation; we focus instead on the structure of the tried and new tables, timestamps and their impact on address selection (Section 2). [17] shows that nodes connecting over Tor can be eclipsed by a Tor exit node that manipulates both bitcoin and Tor. Other work has mapped bitcoin peers to autonomous systems [38], geolocated peers and measured churn [34], and used side channels to learn the bitcoin network topology [16, 51].

p2p and botnet architectures. There has been extensive research on eclipse attacks [27, 61, 62] in structured p2p networks built upon distributed hash tables (DHTs); see [64] for a survey. Many proposals defend against eclipse attacks by adding more structure; [61] constrains peer degree, while others use constraints based on distance metrics like latency [42] or DHT identifiers [13]. Bitcoin, by contrast, uses an unstructured network. While we have focused on exploiting specific quirks in bitcoin's existing network, other works *e.g.,* [11, 15, 21, 44] design new unstructured networks that are robust to Byzantine attacks. [44] blacklists misbehaving peers. Puppetcast's [15] centralized solution is based on public-key infrastructure [15], which is not appropriate for bitcoin. Brahms [21] is fully decentralized, and instead constrains the rate at which peers exchange network information—a useful idea that is a significant departure from bitcoin's current approach. Meanwhile, our goals are also more modest than those in these works; rather than requiring that each node is *equally likely* to be sampled by an honest node, we just want to limit eclipse attacks on initially well-connected nodes. Thus, our countermeasures are inspired by botnet architectures, which share this same goal. Rossow *et al.* [59] finds that many botnets, like bitcoin, use unstructured peer-to-peer networks and gossip (*i.e.,* ADDR messages), and describes how botnets defend against attacks that flood local address tables with bogus information. The Sality botnet refuses to evict "high-reputation" addresses; our anchor countermeasure is similar (Section 7). Storm uses test-before-evict [30], which we have also recommended for bitcoin. Zeus [12] disallows connections from multiple IP in the same /20, and regularly clean tables by testing if peers are online; our feeler connections are similar.

9 Conclusion

We presented an eclipse attack on bitcoin's peer-to-peer network that undermines bitcoin's core security guarantees, allowing attacks on the mining and consensus system, including *N*-confirmation double spending and adversarial forks in the blockchain. Our attack is for nodes with public IPs. We developed mathematical models of our attack, and validated them with Monte Carlo simulations, measurements and experiments. We demonstrated the practically of our attack by performing it on our own live bitcoin nodes, finding that an attacker with 32 distinct /24 IP address blocks, or a 4600-node botnet, can eclipse a victim with over 85% probability in the attacker's *worst case*. Moreover, even a 400-node botnet sufficed to attack our own live bitcoin nodes. Finally, we proposed countermeasures that make eclipse attacks more difficult while still preserving bitcoin's openness and decentralization; several of these were incorporated in a recent bitcoin software upgrade.

Acknowledgements

We thank Foteini Baldimtsi, Wil Koch, and the USENIX Security reviewers for comments on this paper, various bitcoin users for donating their peers files, and the bitcoin core devs for discussions and for implementing Countermeasures 1,2,6. E.H., A.K., S.G. were supported in part by NSF award 1350733, and A.Z. by ISF Grants 616/13, 1773/13, and the Israel Smart Grid (ISG) Consortium.

References

[1] Amazon web services elastic ip. http://aws.amazon.com/ec2/faqs/#elastic-ip. Accessed: 2014-06-18.

[2] Bitcoin: Common vulnerabilities and exposures. https://en.bitcoin.it/wiki/Common_Vulnerabilities_and_Exposures. Accessed: 2014-02-11.

[3] Bitcoin wiki: Double-spending. https://en.bitcoin.it/wiki/Double-spending. Accessed: 2014-02-09.

[4] Bitnode.io snapshot of reachable nodes. https://getaddr.bitnodes.io/nodes/. Accessed: 2014-02-11.

[5] Bitpay: What is transaction speed? https://support.bitpay.com/hc/en-us/articles/202943915-What-is-Transaction-Speed-. Accessed: 2014-02-09.

[6] Bug bounty requested: 10 btc for huge dos bug in all current bitcoin clients. Bitcoin Forum. https://bitcointalk.org/index.php?topic=944369.msg10376763#msg10376763. Accessed: 2014-06-17.

[7] CVE-2013-5700: Remote p2p crash via bloom filters. https://en.bitcoin.it/wiki/Common_Vulnerabilities_and_Exposures. Accessed: 2014-02-11.

[8] Microsoft azure ip address pricing. http://azure.microsoft.com/en-us/pricing/details/ip-addresses/. Accessed: 2014-06-18.

[9] Rackspace: Requesting additional ipv4 addresses for cloud servers. http://www.rackspace.com/knowledge_center/article/requesting-additional-ipv4-addresses-for-cloud-servers. Accessed: 2014-06-18.

[10] Ghash.io and double-spending against betcoin dice, October 30 2013.

[11] Anceaume, E., Busnel, Y., and Gambs, S. On the power of the adversary to solve the node sampling problem. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XI*. Springer, 2013, pp. 102–126.

[12] Andriesse, D., and Bos, H. An analysis of the zeus peer-to-peer protocol, April 2014.

[13] Awerbuch, B., and Scheideler, C. Robust random number generation for peer-to-peer systems. In *Principles of Distributed Systems*. Springer, 2006, pp. 275–289.

[14] Bahack, L. Theoretical bitcoin attacks with less than half of the computational power (draft). *arXiv preprint arXiv:1312.7013* (2013).

[15] Bakker, A., and Van Steen, M. Puppetcast: A secure peer sampling protocol. In *European Conference on Computer Network Defense (EC2ND)* (2008), IEEE, pp. 3–10.

[16] Biryukov, A., Khovratovich, D., and Pustogarov, I. Deanonymisation of clients in Bitcoin P2P network. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 15–29.

[17] Biryukov, A., and Pustogarov, I. Bitcoin over tor isn't a good idea. *arXiv preprint arXiv:1410.6079* (2014).

[18] Bitcoin Wiki. Confirmation. https://en.bitcoin.it/wiki/Confirmation, February 2015.

[19] Bitcoin Wisdom. Bitcoin difficulty and hash rate chart. https://bitcoinwisdom.com/bitcoin/difficulty, February 2015.

[20] blockchain.io. Average transaction confirmation time. https://blockchain.info/charts/avg-confirmation-time, February 2015.

[21] Bortnikov, E., Gurevich, M., Keidar, I., Kliot, G., and Shraer, A. Brahms: Byzantine resilient random membership sampling. *Computer Networks 53*, 13 (2009), 2340–2359.

[22] Brands, S. Untraceable off-line cash in wallets with observers (extended abstract). In *CRYPTO* (1993).

[23] CAIDA. AS to Organization Mapping Dataset, July 2014.

[24] CAIDA. Routeviews prefix to AS Mappings Dataset for IPv4 and IPv6, July 2014.

[25] Camenisch, J., Hohenberger, S., and Lysyanskaya, A. Compact e-cash. In *EUROCRYPT* (2005).

[26] CarnaBotnet. Internet census 2012. http://internetcensus2012.bitbucket.org/paper.html, 2012.

[27] Castro, M., Druschel, P., Ganesh, A., Rowstron, A., and Wallach, D. S. Secure routing for structured peer-to-peer overlay networks. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 299–314.

[28] Chaum, D. Blind signature system. In *CRYPTO* (1983).

[29] Courtois, N. T., and Bahack, L. On subversive miner strategies and block withholding attack in bitcoin digital currency. *arXiv preprint arXiv:1402.1718* (2014).

[30] Davis, C. R., Fernandez, J. M., Neville, S., and McHugh, J. Sybil attacks as a mitigation strategy against the storm botnet. In *3rd International Conference on Malicious and Unwanted Software, 2008.* (2008), IEEE, pp. 32–40.

[31] Decker, C., and Wattenhofer, R. Information propagation in the bitcoin network. In *IEEE Thirteenth International Conference on Peer-to-Peer Computing (P2P)* (2013), IEEE, pp. 1–10.

[32] Dillon, J. Bitcoin-development mailinglist: Protecting bitcoin against network-wide dos attack. `http://sourceforge.net/p/bitcoin/mailman/message/31168096/`, 2013. Accessed: 2014-02-11.

[33] Dingledine, R., Hopper, N., Kadianakis, G., and Mathewson, N. One fast guard for life (or 9 months). In *7th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs 2014)* (2014).

[34] Donet, J. A. D., Pérez-Sola, C., and Herrera-Joancomartí, J. The bitcoin p2p network. In *Financial Cryptography and Data Security*. Springer, 2014, pp. 87–102.

[35] Durumeric, Z., Wustrow, E., and Halderman, J. A. ZMap: Fast Internet-wide scanning and its security applications. In *Proceedings of the 22nd USENIX Security Symposium* (Aug. 2013).

[36] Eyal, I. The miner's dilemma. *arXiv preprint arXiv:1411.7099* (2014).

[37] Eyal, I., and Sirer, E. G. Majority is not enough: Bitcoin mining is vulnerable. In *Financial Cryptography and Data Security*. Springer, 2014, pp. 436–454.

[38] Feld, S., Schönfeld, M., and Werner, M. Analyzing the deployment of bitcoin's p2p network under an as-level perspective. *Procedia Computer Science 32* (2014), 1121–1126.

[39] Finney, H. Bitcoin talk: Finney attack. `https://bitcointalk.org/index.php?topic=3441.msg48384#msg48384`, 2011. Accessed: 2014-02-12.

[40] Heilman, E. Bitcoin: Added test-before-evict discipline in addrman, feeler connections. `https://github.com/bitcoin/bitcoin/pull/6355`.

[41] Heilman, E., Kendler, A., Zohar, A., and Goldberg, S. Eclipse attacks on bitcoins peer-to-peer network (full version). Tech. Rep. 2015/263, ePrint Cryptology Archive, `http://eprint.iacr.org/2015/263.pdf`, 2015.

[42] Hildrum, K., and Kubiatowicz, J. Asymptotically efficient approaches to fault-tolerance in peer-to-peer networks. In *Distributed Computing*. Springer, 2003, pp. 321–336.

[43] IANA. Iana ipv4 address space registry. `http://www.iana.org/assignments/ipv4-address-space/ipv4-address-space.xhtml`, January 2015.

[44] Jesi, G. P., Montresor, A., and van Steen, M. Secure peer sampling. *Computer Networks 54*, 12 (2010), 2086–2098.

[45] Johnson, B., Laszka, A., Grossklags, J., Vasek, M., and Moore, T. Game-theoretic analysis of ddos attacks against bitcoin mining pools. In *Financial Cryptography and Data Security*. Springer, 2014, pp. 72–86.

[46] Karame, G., Androulaki, E., and Capkun, S. Two bitcoins at the price of one? double-spending attacks on fast payments in bitcoin. *IACR Cryptology ePrint Archive 2012* (2012), 248.

[47] King, L. Bitcoin hit by 'massive' ddos attack as tensions rise. *Forbes* *`http://www.forbes.com/sites/leoking/2014/02/12/bitcoin-hit-by-massive-ddos-attack-as-tensions-rise/`* (December 2 2014).

[48] Koshy, P., Koshy, D., and McDaniel, P. An analysis of anonymity in bitcoin using p2p network traffic. In *Financial Cryptography and Data Security*. 2014.

[49] Kroll, J. A., Davey, I. C., and Felten, E. W. The economics of bitcoin mining, or bitcoin in the presence of adversaries. In *Proceedings of WEIS* (2013), vol. 2013.

[50] Laszka, A., Johnson, B., and Grossklags, J. When bitcoin mining pools run dry. *2nd Workshop on Bitcoin Research (BITCOIN)* (2015).

[51] Miller, A., Litton, J., Pachulski, A., Gupta, N., Levin, D., Spring, N., and Bhattacharjee, B. Discovering bitcoin's network topology and influential nodes. Tech. rep., University of Maryland, 2015.

[52] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system.

[53] OpenSSL. TLS heartbeat read overrun (CVE-2014-0160). `https://www.openssl.org/news/secadv_20140407.txt`, April 7 2014.

[54] Plohmann, D., and Gerhards-Padilla, E. Case study of the miner botnet. In *Cyber Conflict (CYCON), 2012 4th International Conference on* (2012), IEEE, pp. 1–16.

[55] RIPE. Ripestat. `https://stat.ripe.net/data/announced-prefixes`, October 2014.

[56] RIPE. Latest delegations. `ftp://ftp.ripe.net/pub/stats/ripencc/delegated-ripencc-extended-latest`, 2015.

[57] RoadTrain. Bitcoin-talk: Ghash.io and double-spending against betcoin dice. `https://bitcointalk.org/index.php?topic=321630.msg3445371#msg3445371`, 2013. Accessed: 2014-02-14.

[58] Rosenfeld, M. Analysis of hashrate-based double spending. *arXiv preprint arXiv:1402.2009* (2014).

[59] Rossow, C., Andriesse, D., Werner, T., Stone-Gross, B., Plohmann, D., Dietrich, C. J., and Bos, H. Sok: P2pwned-modeling and evaluating the resilience of peer-to-peer botnets. In *IEEE Symposium on Security and Privacy* (2013), IEEE, pp. 97–111.

[60] Shomer, A. On the phase space of block-hiding strategies. *IACR Cryptology ePrint Archive 2014* (2014), 139.

[61] Singh, A., Ngan, T.-W. J., Druschel, P., and Wallach, D. S. Eclipse attacks on overlay networks: Threats and defenses. In *In IEEE INFOCOM* (2006).

[62] Sit, E., and Morris, R. Security considerations for peer-to-peer distributed hash tables. In *Peer-to-Peer Systems*. Springer, 2002, pp. 261–269.

[63] Stock, B., Gobel, J., Engelberth, M., Freiling, F. C., and Holz, T. Walowdac: Analysis of a peer-to-peer botnet. In *European Conference on Computer Network Defense (EC2ND)* (2009), IEEE, pp. 13–20.

[64] Urdaneta, G., Pierre, G., and Steen, M. V. A survey of dht security techniques. *ACM Computing Surveys (CSUR) 43*, 2 (2011), 8.

[65] Vasek, M., Thornton, M., and Moore, T. Empirical analysis of denial-of-service attacks in the bitcoin ecosystem. In *Financial Cryptography and Data Security*. Springer, 2014, pp. 57–71.

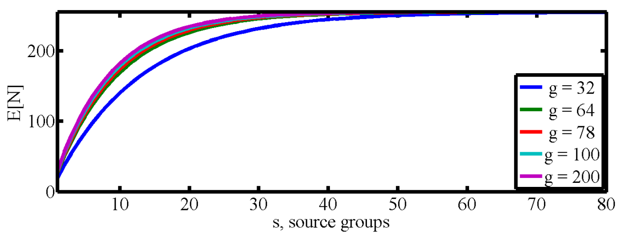


Figure 9: $E[N]$ vs s (the number of source groups) for different choices of g (number of groups per source group) when overwriting the new table per equation (13).

A A Useful Lemma

Lemma A.1. *If k items are randomly and independently inserted into n buckets, and X is a random variable counting the number of non-empty buckets, then*

$$E[X] = n\left(1 - (\tfrac{n-1}{n})^k\right) \approx n(1 - e^{-\frac{k}{n}}) \qquad (12)$$

Proof. Let $X_i = 1$ if bucket i is non-empty, and $X_i = 0$ otherwise. The probability that the bucket i is empty after the first item is inserted is $(\frac{n-1}{n})$. After inserting k items

$$\Pr[X_i = 1] = 1 - \left(\tfrac{n-1}{n}\right)^k$$

It follows that

$$E[X] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} \Pr[X_i = 1] = n(1 - (\tfrac{n-1}{n})^k)$$

(12) follows since $(\frac{n-1}{n}) \approx e^{-1/n}$ for $n \gg 1$. □

B Overwriting the New Table

How should the attacker send ADDR messages that overwrite the new table with "trash" IP addresses? Our "trash" is from the unallocated Class A IPv4 address block 252.0.0.0/8, designated by IANA as "reserved for future use" [43]; any connections these addresses will fail, forcing the victim to choose an address from tried. Next, recall (Section 2.2) that the pair *(group, source group)* determines the bucket in which an address in an ADDR message is stored. Thus, if the attacker controls nodes in s different groups, then s is the number of *source groups*. We suppose that nodes in each source group can push ADDR messages containing addresses from g distinct groups; the "trash" 252.0.0.0/8 address block give an upper bound on g of $2^8 = 256$. Each group contains a distinct addresses. How large should s, g, and a be so that the new table is overwritten by "trash" addresses?

B.1 Infrastructure strategy

In an infrastructure attack, the number of source groups s is constrained, and the number of groups g is essentially unconstrained. By Lemma A.1, the expected number of buckets filled by a s source groups is

$$E[N] = 256(1 - (\tfrac{255}{256})^{32s}) \qquad (13)$$

We expect to fill ≈ 251 of 256 new buckets with $s = 32$.

Each (group, source group) pair maps to a unique bucket in new, and each bucket in new can hold 64 addresses. Bitcoin eviction is used, and we suppose each new bucket is completely full of legitimate addresses that are older than all the addresses inserted by the adversary via ADDR messages. Since all a addresses in a particular (group, source group) pair map to a single bucket, it follows that the number of addresses that actually stored in that bucket is given by $E[Y_a]$ in the recurrence relation of equations of (5)-(6). With $a = 125$ addresses, the adversary expects to overwrite $E[Y_a] = 63.8$ of the 64 legitimate addresses in the bucket. We thus require each source group to have 32 peers, and each peer to send ADDR messages with 8 distinct groups of $a = 125$ addresses. Thus, there are $g = 32 \times 8 = 256$ groups per source group, which is exactly the maximum number of groups available in our trash IP address block. Each peer sends exactly one ADDR message with $8 \times 125 = 1000$ address, for a total of $256 \times 125 \times s$ *distinct* addresses sent by all peers. (There are 2^{24} addresses in the 252.0.0.0/8 block, so all these addresses are distinct if $s < 524$.)

B.2 Botnet strategy

In a botnet attack, each of the attacker's t nodes is in a distinct source group. For $s = t > 200$, which is the case for all our botnet attacks, equation (13) shows that the number of source groups $s = t$ is essentially unconstrained. We thus require each peer to send a single ADDR message containing 1000 addresses with 250 distinct groups of four addresses each. Since $s = t$ is so large, we can model this by assuming that each (group, source group) pair selects a bucket in new uniformly at random, and inserts 4 addresses into that bucket; thus, the expected number of addresses inserted per bucket will be tightly concentrated around

$$4 \times E[B(250t, \tfrac{1}{256}] = 3.9t$$

For $t > 200$, we expect at least 780 address to be inserted into each bucket. From equations (5) and (6), we find $E[Y_{780}] \approx 64$, so that each new bucket is likely to be full.

Compiler-instrumented, Dynamic Secret-Redaction of Legacy Processes for Attacker Deception

Frederico Araujo and Kevin W. Hamlen
The University of Texas at Dallas
{frederico.araujo, hamlen}@utdallas.edu

Abstract

An enhanced dynamic taint-tracking semantics is presented and implemented, facilitating fast and precise runtime secret redaction from legacy processes, such as those compiled from C/C++. The enhanced semantics reduce the annotation burden imposed upon developers seeking to add secret-redaction capabilities to legacy code, while curtailing over-tainting and label creep.

An implementation for LLVM's DataFlow Sanitizer automatically instruments taint-tracking and secret-redaction support into annotated C/C++ programs at compile-time, yielding programs that can self-censor their address spaces in response to emerging cyber-attacks. The technology is applied to produce the first information flow-based honey-patching architecture for the Apache web server. Rather than merely blocking intrusions, the modified server deceptively diverts attacker connections to secret-sanitized process clones that monitor attacker activities and disinform adversaries with honey-data.

1 Introduction

Redaction of sensitive information from documents has been used since ancient times as a means of concealing and removing secrets from texts intended for public release. As early as the 13th century B.C., Pharaoh Horemheb, in an effort to conceal the acts of his predecessors from future generations, so thoroughly located and erased their names from all monument inscriptions that their identities weren't rediscovered until the 19th century A.D. [22]. In the modern era of digitally manipulated data, *dynamic taint analysis* (cf., [40]) has become an important tool for automatically tracking the flow of secrets (*tainted data*) through computer programs as they execute. Taint analysis has myriad applications, including program vulnerability detection [5, 6, 9, 25, 33, 34, 37, 45, 46], malware analysis [19, 20, 36, 48], test set generation [3, 42], and information leak detection [4, 14, 21, 23, 24, 49].

Our research introduces and examines the associated challenge of secret redaction from program process images. Safe, efficient redaction of secrets from program address spaces has numerous potential applications, including the safe release of program memory dumps to software developers for debugging purposes, mitigation of cyber-attacks via runtime self-censoring in response to intrusions, and attacker deception through honey-potting.

A recent instantiation of the latter is *honey-patching* [2], which proposes crafting software security patches in such a way that future attempted exploits of the patched vulnerabilities appear successful to attackers. This frustrates attacker vulnerability probing, and affords defenders opportunities to disinform attackers by divulging "fake" secrets in response to attempted intrusions. In order for such deceptions to succeed, honey-patched programs must be imbued with the ability to impersonate unpatched software with all secrets replaced by honey-data. That is, they require a technology for rapidly and thoroughly redacting all secrets from the victim program's address space at runtime, yielding a vulnerable process that the attacker may further penetrate without risk of secret disclosure.

Realizing such runtime process secret redaction in practice educes at least two significant research challenges. First, the redaction step must yield a runnable program process. Non-secrets must therefore not be conservatively redacted, lest data critical for continuing the program's execution be deleted. Secret redaction for running processes is hence especially sensitive to *label creep* and *over-tainting* failures. Second, many real-world programs targeted by cyber-attacks were not originally designed with information flow tracking support, and are often expressed in low-level, type-unsafe languages, such as C/C++. A suitable solution must be amenable to retrofitting such low-level, legacy software with annotations sufficient to distinguish non-secrets from secrets, and with efficient flow-tracking logic that does not impair performance.

Our approach builds upon the LLVM compiler's [31] DataFlow Sanatizer (DFSan) infrastructure [18], which adds byte-granularity taint-tracking support to C/C++ programs at compile-time. At the source level, DFSan's taint-tracking capabilities are purveyed as runtime data-classification, data-declassification, and taint-checking operations, which programmers add to their programs to identify secrets and curtail their flow at runtime. Unfortunately, straightforward use of this interface for redaction of large, complex legacy codes can lead to severe over-tainting, or requires an unreasonably detailed retooling of the code with copious classification operations. This is unsafe, since missing even one of these classification points during retooling risks disclosing secrets to adversaries.

To overcome these deficiencies, we augment DFSan with a declarative, type annotation-based secret-labeling mechanism for easier secret identification; and we introduce a new label propagation semantics, called *Pointer Conditional-Combine Semantics* (PC^{2}S), that efficiently distinguishes secret data within C-style graph data structures from the non-secret structure that houses the data. This partitioning of the bytes greatly reduces over-tainting and the programmer's annotation burden, and proves critical for precisely redacting secret process data whilst preserving process operation after redaction.

Our innovations are showcased through the development of a taint tracking-based honey-patching framework for three production web servers, including the popular Apache HTTP server (∼2.2M SLOC). The modified servers respond to detected intrusions by transparently forking attacker sessions to unpatched process clones in confined decoy environments. Runtime redaction preserves attacker session data without preserving data owned by other users, yielding a deceptive process that continues servicing the attacker without divulging secrets. The decoy can then monitor attacker strategies, harvest attack data, and disinform the attacker with honey-data in the form of false files or process data.

Our contributions can be summarized as follows:

- We introduce a pointer tainting methodology through which secret sources are derived from statically annotated data structures, lifting the burden of identifying classification code-points in legacy C code.
- We propose and formalize taint propagation semantics that accurately track secrets while controlling taint spread. Our solution is implemented as a small extension to LLVM, allowing it to be applied to a large class of COTS applications.
- We implement a memory redactor for secure honey-patching. Evaluation shows that our implementation is both more efficient and more secure than previous pattern-matching based redaction approaches.
- Implementations and evaluations for three production web servers demonstrate that the approach is feasible for large-scale, performance-critical software with reasonable overheads.

Listing 1: Apache's URI parser function (excerpt)

```
1  /* first colon delimits username:password */
2  s1 = memchr(hostinfo, ':', s − hostinfo);
3  if (s1) {
4    uptr−>user = apr_pstrmemdup(p, hostinfo, s1 − hostinfo);
5    ++s1;
6    uptr−>password = apr_pstrmemdup(p, s1, s − s1);
7  }
```

2 Approach Overview

We first outline practical limitations of traditional dynamic taint-tracking for analyzing dataflows in server applications, motivating our research. We then overview our approach and its application to the problem of redacting secrets from runtime process memory images.

2.1 Dynamic Taint Analysis

Dynamic taint analyses enforce *taint policies*, which specify how data confidentiality and integrity classifications (*taints*) are introduced, propagated, and checked as a program executes. *Taint introduction* rules specify taint sources—typically a subset of program inputs. *Taint propagation* rules define how taints flow. For example, the result of summing tainted values might be a sum labeled with the union (or more generally, the *lattice join*) of the taints of the summands. *Taint checking* is the process of reading taints associated with data, usually to enforce an information security policy. Taints are usually checked at data usage or disclosure points, called *sinks*.

Extending taint-tracking to low-level, legacy code not designed with taint-tracking in mind is often difficult. For example, the standard approach of specifying taint introductions as annotated program inputs often proves too coarse for inputs comprising low-level, unstructured data streams, such as network sockets. Listing 1 exemplifies the problem using a code excerpt from the Apache web server [1]. The excerpt partitions a byte stream (stored in buffer `s1`) into a non-secret user name and a secret password, delimited by a colon character. Naïvely labeling input `s1` as secret to secure the password over-taints the user name (and the colon delimiter, and the rest of the stream), leading to excessive label creep—everything associated with the stream becomes secret, with the result that nothing can be safely divulged.

A correct solution must more precisely identify data field `uptr->password` (but not `uptr->user`) as secret after the unstructured data has been parsed. This is achieved in DFSan by manually inserting a runtime classification operation after line 6. However, on a larger scale this brute-force labeling strategy imposes a dangerously heavy annotation burden on developers, who must manually locate all such classification points. In C/C++ programs littered with pointer arithmetic, the correct classification points can often be obscure. Inadvertently omitting even one classification risks information leaks.

2.2 Sourcing & Tracking Secrets

To ease this burden, we introduce a mechanism whereby developers can identify secret-storing structures and fields *declaratively* rather than operationally. For example, to correctly label the password in Listing 1 as secret, users of our system may add type qualifier SECRET_STR to the password field's declaration in its abstract datatype definition. Our modified LLVM compiler responds to this static annotation by dynamically tainting all values assigned to the password field. Since datatypes typically have a single point of definition (in contrast to the many code points that access them), this greatly reduces the annotation burden imposed upon code maintainers.

In cases where the appropriate taint is not statically known (e.g., if each password requires a different, user-specific taint label), parameterized type-qualifier SECRET$\langle f \rangle$ identifies a user-implemented function f that computes the appropriate taint label at runtime.

Unlike traditional taint introduction semantics, which label program input values and sources with taints, recognizing structure fields as taint sources requires a new form of taint semantics that conceptually interprets dynamically identified *memory addresses* as taint sources. For example, a program that assigns address &(uptr->password) to pointer variable p, and then assigns a freshly allocated memory address to $*p$, must automatically identify the freshly allocated memory as a new taint source, and thereafter taint any values stored at $*p[i]$ (for all indexes i).

To achieve this, we leverage and extend DFSan's *pointer-combine semantics (PCS)* feature, which optionally combines (i.e., joins) the taints of pointers and pointees during pointer dereferences. Specifically, when *PCS on-load* is enabled, read-operation $*p$ yields a value tainted with the join of pointer p's taint and the taint of the value to which p points; and when *PCS on-store* is enabled, write-operation $*p := e$ taints the value stored into $*p$ with the join of p's and e's taints. Using PCS leads to a natural encoding of SECRET annotations as pointer taints. Continuing the previous example, PCS propagates uptr->password's taint to p, and subsequent dereferencing assignments propagate the two pointers' taints to secrets stored at their destinations.

PCS works well when secrets are always separated from the structures that house them by a level of pointer indirection, as in the example above (where uptr->password is a pointer to the secret rather than the secret itself). However, label creep difficulties arise when structures mix secret values with non-secret pointers. To illustrate, consider a simple linked list ℓ of secret integers, where each integer has a different taint. In order for PCS on-store to correctly classify values stored to ℓ->secret_int, pointer ℓ must have taint γ_1, where γ_1 is the desired taint of the first integer. But this causes stores to ℓ->next to incorrectly propagate taint γ_1 to the node's next-pointer, which propagates γ_1 to subsequent nodes when dereferenced. In the worst case, all nodes become labeled with all taints. Such issues have spotlighted effective pointer tainting as a significant challenge in the taint-tracking literature [17, 27, 40, 43].

Listing 2: Abbreviated Apache's session record struct

```
typedef struct {
    NONSECRET apr_pool_t *pool;
    NONSECRET apr_uuid_t *uuid;
    SECRET_STR const char *remote_user;
    apr_table_t *entries;
    ...
} SECRET session_rec;
```

To address this shortcoming, we introduce a new, generalized PC^{2}S semantics that augments PCS with pointer-combine *exemptions* conditional upon the static type of the pointee. In particular, a PC^{2}S taint-propagation policy may dictate that taint labels are not combined when the pointee has pointer type. Hence, ℓ->secret_int receives ℓ's taint because the assigned expression has integer type, whereas ℓ's taint is *not* propagated to ℓ->next because the latter's assigned expression has pointer type. We find that just a few strategically selected exemption rules expressed using this refined semantics suffices to vastly reduce label creep while correctly tracking all secrets in large legacy source codes.

In order to strike an acceptable balance between security and usability, our solution only automates tainting of C/C++ style structures whose non-pointer fields share a common taint. Non-pointer fields of mixed taintedness within a single struct are not supported automatically because C programs routinely use pointer arithmetic to reference multiple fields in a struct via a common pointer (imparting the pointer's taint to all the struct's non-pointer fields). Our work therefore targets the common case in which the taint policy is expressible at the granularity of structures, with exemptions for fields that point to other (differently tainted) structure instances. This corresponds to the usual scenario where a non-secret graph structure (e.g., a tree) stores secret data in its nodes.

Users of our system label structure datatypes as SECRET (implicitly introducing a taint to all fields within the structure), and additionally annotate pointer fields as NONSECRET to exempt their taints from pointer-combines during dereferences. Pointers to dynamic-length, null-terminated secrets get annotation SECRET_STR. For example, Listing 2 illustrates the annotation of session_req, used by Apache to store remote users' session data. Finer-granularity policies remain enforceable, but require manual instrumentation via DFSan's API, to precisely distinguish which of the code's pointer dereference operations propagate pointer taints. Our solution thus complements existing approaches.

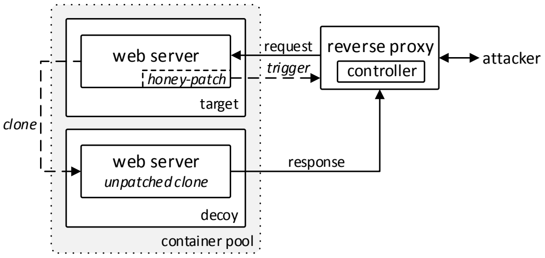


Figure 1: Architectural overview of honey-patching.

2.3 Application Study: Honey-Patching

Our discoveries are applied to realize practical, efficient honey-patching of legacy web servers for attacker deception. Typical software security patches fix newly discovered vulnerabilities at the price of advertising to attackers which systems have been patched. Cyber-criminals therefore easily probe today's Internet for vulnerable software, allowing them to focus their attacks on susceptible targets.

Honey-patching, depicted in Figure 1, is a recent strategy for frustrating such attacks. In response to malicious inputs, honey-patched applications clone the attacker session onto a confined, ephemeral, decoy environment, which behaves henceforth as an unpatched, vulnerable version of the software. This potentially augments the server with an embedded honeypot that waylays, monitors, and disinforms criminals.

Highly efficient cloning is critical for such architectures, since response delays risk alerting attackers to the deception. The cloning process must therefore rapidly locate and redact all secrets from the process address space, yielding a runnable process with only the attacker's session data preserved. Moreover, redaction must not be overly conservative. If redaction crashes the clone with high probability, or redacts obvious non-secrets, this too alerts the attacker. To our knowledge, no prior taint-tracking approach satisfies all of these demanding performance, precision, and legacy-maintainability requirements. We therefore select honey-patching of Apache as our flagship case-study.

3 Formal Semantics

For explanatory precision, we formally define our new taint-tracking semantics in terms of the simple, typed intermediate language (IL) in Figure 2, inspired by prior work [40]. The simplified IL abstracts irrelevant details of LLVM's IR language, capturing only those features needed to formalize our analysis.

3.1 Language Syntax

Programs $\mathcal{P}$ are lists of commands, denoted $\overline{c}$. Commands consist of variable assignments, pointer-dereferencing assignments (stores), conditional branches, function invocations, and function returns. Expressions evaluate to value-taint pairs $\langle u, \gamma \rangle$, where u ranges over typical value representations, and γ is the taint label associated with u. Labels denote sets of taints; they therefore comprise a lattice ordered by subset ($\sqsubseteq$), with the empty set $\bot$ at the bottom (denoting public data), and the universe $\top$ of all taints at the top (denoting maximally secret data). Join operation $\sqcup$ denotes least upper bound.

$$\begin{array}{ll}
\textit{programs} & \mathcal{P} ::= \overline{c} \\
\textit{commands} & c ::= v{:=}e \mid \texttt{store}(\tau, e_1, e_2) \mid \texttt{ret}(\tau,\ e) \\
 & \quad \mid \texttt{call}(\tau, e, \overline{args}) \mid \texttt{br}(e, e_1, e_0) \\
\textit{expressions} & e ::= v \mid \langle u, \gamma \rangle \mid \Diamond_b(\tau, e_1, e_2) \mid \texttt{load}(\tau, e) \\
\textit{binary ops} & \Diamond_b ::= \text{typical binary operators} \\
\textit{variables} & v \\
\textit{values} & u ::= \text{values of underlying IR language} \\
\textit{types} & \tau ::= \mathit{ptr}\ \tau \mid \tau\ \overline{\tau} \mid \text{primitive types} \\
\textit{taint labels} & \gamma \in (\Gamma, \sqsubseteq) \quad \text{(label lattice)} \\
 & \\
\textit{locations} & \ell ::= \text{memory addresses} \\
\textit{environment} & \Delta : v \rightharpoonup u \\
\textit{prog counter} & pc \\
\textit{stores} & \sigma : (\ell \rightharpoonup u) \cup (v \rightharpoonup \ell) \\
\textit{functions} & f \\
\textit{function table} & \phi : f \rightharpoonup \ell \\
\textit{taint contexts} & \lambda : (\ell \cup v) \rightharpoonup \gamma \\
\textit{propagation} & \rho : \overline{\gamma} \to \gamma \\
\textit{prop contexts} & \mathcal{A} : f \to \rho \\
\textit{call stack} & \Xi ::= \mathit{nil} \mid \langle f,\ pc,\ \Delta,\ \overline{\gamma} \rangle :: \Xi
\end{array}$$

Figure 2: Intermediate representation syntax.

Variable names range over identifiers and function names, and the type system supports pointer types, function types, and typical primitive types. Since DFSan's taint-tracking is dynamic, we here omit a formal static semantics and assume that programs are well-typed.

Execution contexts are comprised of a store σ relating locations to values and variables to locations, an environment Δ mapping variables to values, and a tainting context λ mapping locations and variables to taint labels. Additionally, to express the semantics of label propagation for external function calls (e.g., runtime library API calls), we include a function table ϕ that maps external function names to their entry points, a propagation context $\mathcal{A}$ that dictates whether and how each external function propagates its argument labels to its return value label, and the call stack Ξ. Taint propagation policies returned by $\mathcal{A}$ are expressed as customizable mappings ρ from argument labels $\overline{\gamma}$ to return labels γ.

$$\frac{}{\sigma, \Delta, \lambda \vdash u \Downarrow \langle u, \bot \rangle}\;\textsc{Val} \qquad \frac{}{\sigma, \Delta, \lambda \vdash v \Downarrow \langle \Delta(v), \lambda(v) \rangle}\;\textsc{Var}$$

$$\frac{\sigma, \Delta, \lambda \vdash e_1 \Downarrow \langle u_1, \gamma_1 \rangle \quad \sigma, \Delta, \lambda \vdash e_2 \Downarrow \langle u_2, \gamma_2 \rangle}{\sigma, \Delta, \lambda \vdash \Diamond_b(\tau,\ e_1,\ e_2) \Downarrow \langle u_1 \Diamond_b u_2, \gamma_1 \sqcup \gamma_2 \rangle}\;\textsc{BinOp} \qquad \frac{\sigma, \Delta, \lambda \vdash e \Downarrow \langle u, \gamma \rangle}{\sigma, \Delta, \lambda \vdash \texttt{load}(\tau,\ e) \Downarrow \langle \sigma(u), \rho_{load}(\tau, \gamma, \lambda(u)) \rangle}\;\textsc{Load}$$

$$\frac{\sigma, \Delta, \lambda \vdash e \Downarrow \langle u, \gamma \rangle \quad \Delta' = \Delta[v \mapsto u] \quad \lambda' = \lambda[v \mapsto \gamma]}{\langle \sigma, \Delta, \lambda, \Xi, pc, v := e \rangle \to_1 \langle \sigma, \Delta', \lambda', \Xi, pc + 1, \mathcal{P}[pc + 1] \rangle}\;\textsc{Assign}$$

$$\frac{\sigma, \Delta, \lambda \vdash e_1 \Downarrow \langle u_1, \gamma_1 \rangle \quad \sigma, \Delta, \lambda \vdash e_2 \Downarrow \langle u_2, \gamma_2 \rangle \quad \sigma' = \sigma[u_1 \mapsto u_2] \quad \lambda' = \lambda[u_1 \mapsto \rho_{store}(\tau, \gamma_1, \gamma_2)]}{\langle \sigma, \Delta, \lambda, \Xi, pc, \texttt{store}(\tau, e_1, e_2) \rangle \to_1 \langle \sigma', \Delta, \lambda', \Xi, pc + 1, \mathcal{P}[pc + 1] \rangle}\;\textsc{Store}$$

$$\frac{\sigma, \Delta, \lambda \vdash e \Downarrow \langle u, \gamma \rangle \qquad \sigma, \Delta, \lambda \vdash e_{(u\,?\,1\,:\,0)} \Downarrow \langle u', \gamma' \rangle}{\langle \sigma, \Delta, \lambda, \Xi, pc, \texttt{br}(e, e_1, e_0) \rangle \to_1 \langle \sigma, \Delta, \lambda, \Xi, u', \mathcal{P}[u'] \rangle}\;\textsc{Cond}$$

$$\frac{\begin{array}{c}\sigma, \Delta, \lambda \vdash e_1 \Downarrow \langle u_1, \gamma_1 \rangle \quad \cdots \quad \sigma, \Delta, \lambda \vdash e_n \Downarrow \langle u_n, \gamma_n \rangle \\ \Delta' = \Delta[\overline{params_f} \mapsto \overline{u_1 \cdots u_n}] \quad \lambda' = \lambda[\overline{params_f} \mapsto \overline{\gamma_1 \cdots \gamma_n}] \quad fr = \langle f, pc + 1, \Delta, \overline{\gamma_1 \cdots \gamma_n} \rangle\end{array}}{\langle \sigma, \Delta, \lambda, \Xi, pc, \texttt{call}(\tau, f, \overline{e_1 \cdots e_n}) \rangle \to_1 \langle \sigma, \Delta', \lambda', fr :: \Xi, \phi(f), \mathcal{P}[\phi(f)] \rangle}\;\textsc{Call}$$

$$\frac{\sigma, \Delta, \lambda \vdash e \Downarrow \langle u, \gamma \rangle \quad fr = \langle f, pc', \Delta', \overline{\gamma} \rangle \quad \lambda' = \lambda[v_{ret} \mapsto \mathcal{A}\, f\, \overline{\gamma}]}{\langle \sigma, \Delta, \lambda, fr :: \Xi, pc, \texttt{ret}(\tau, e) \rangle \to_1 \langle \sigma, \Delta'[v_{ret} \mapsto u], \lambda', \Xi, pc', \mathcal{P}[pc'] \rangle}\;\textsc{Ret}$$

Figure 3: Operational semantics of a generalized label propagation semantics.

3.2 Operational Semantics

Figure 3 presents an operational semantics defining how taint labels propagate in an instrumented program. Expression judgments are large-step ($\Downarrow$), while command judgments are small-step ($\to_1$). At the IL level, expressions are pure and programs are non-reflective.

Abstract machine configurations consist of tuples $\langle \sigma, \Delta, \lambda, \Xi, pc, \iota \rangle$, where pc is the program pointer and ι is the current instruction. Notation $\Delta[v \mapsto u]$ denotes function Δ with v remapped to u, and notation $\mathcal{P}[pc]$ refers to the program instruction at address pc. For brevity, we omit $\mathcal{P}$ from machine configurations, since it is fixed.

Rule Val expresses the typical convention that hard-coded program constants are initially untainted ($\bot$). Binary operations are eager, and label their outputs with the join ($\sqcup$) of their operand labels.

The semantics of $\texttt{load}(\tau, e)$ read the value stored in location e, where the label associated with the loaded value is obtained by propagation function ρ_{load}. Dually, $\texttt{store}(\tau, e_1, e_2)$ stores e_2 into location e_1, updating λ according to ρ_{store}. In C programs, these model pointer dereferences and dereferencing assignments, respectively. Parameterizing these rules in terms of abstract propagation functions ρ_{load} and ρ_{store} allows us to instantiate them with customized propagation policies at compile-time, as detailed in §3.3.

External function calls $\texttt{call}(\tau, f, \overline{e_1 \cdots e_n})$ evaluate arguments $\overline{e_1 \cdots e_n}$, create a new stack frame fr, and jump to the callee's entry point. Returns then consult propagation context $\mathcal{A}$ to appropriately label the value returned by the function based on the labels of its arguments. Context $\mathcal{A}$ can be customized by the user to specify how labels propagate through external libraries compiled without taint-tracking support.

NCS	$\rho_{\{load,store\}}(\tau, \gamma_1, \gamma_2) := \gamma_2$
PCS	$\rho_{\{load,store\}}(\tau, \gamma_1, \gamma_2) := \gamma_1 \sqcup \gamma_2$
PC^{2}S	$\rho_{\{load,store\}}(\tau, \gamma_1, \gamma_2) := (\tau \text{ is } ptr)\ ?\ \gamma_2 : (\gamma_1 \sqcup \gamma_2)$

Figure 4: Polymorphic functions modeling no-combine, pointer-combine, and PC^{2}S label propagation policies.

3.3 Label Propagation Semantics

The operational semantics are parameterized by propagation functions ρ that can be instantiated to a specific propagation policy at compile-time. This provides a base framework through which we can study different propagation policies and their differing characteristics.

Figure 4 presents three polymorphic functions that can be used to instantiate propagation policies. On-load propagation policies instantiate ρ_{load}, while on-store policies instantiate ρ_{store}. The instantiations in Figure 4 define no-combine semantics (DFSan's on-store default), PCS (DFSan's on-load default), and our PC^{2}S extensions:

No-combine. The no-combine semantics (NCS) model a traditional, pointer-transparent propagation policy. Pointer labels are ignored during loads and stores, causing loaded and stored data retain their labels irrespective of the labels of the pointers being dereferenced.

Pointer-Combine Semantics. In contrast, PCS joins pointer labels with loaded and stored data labels during loads and stores. Using this policy, a value is tainted on-load (resp., on-store) if its source memory location (resp., source operand) is tainted or the pointer value dereferenced during the operation is tainted. If both are tainted with different labels, the labels are joined to obtain a new label that denotes the union of the originals.

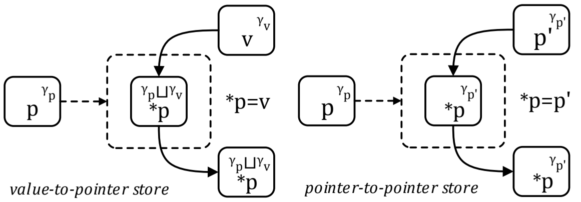


Figure 5: PC^{2}S propagation policy on store commands.

Pointer Conditional-Combine Semantics. PC^{2}S generalizes PCS by conditioning the label-join on the static type of the data operand. If the loaded/stored data has pointer type, it applies the NCS rule; otherwise, it applies the PCS rule. The resulting label propagation for stores is depicted in Figure 5.

This can be leveraged to obtain the best of both worlds. PC^{2}S pointer taints retain most of the advantages of PCS—they can identify and track aliases to birthplaces of secrets, such as data structures where secrets are stored immediately after parsing, and they automatically propagate their labels to data stored there. But PC^{2}S resists PCS's over-tainting and label creep problems by avoiding propagation of pointer labels through levels of pointer indirection, which usually encode relationships with other data whose labels must remain distinct and separately managed.

Condition (τ is ptr) in Figure 4 can be further generalized to any decidable proposition on static types τ. We use this feature to distinguish pointers that cross data ownership boundaries (e.g., pointers to other instances of the parent structure) from pointers that target value data (e.g., strings). The former receive NCS treatment by default to resist over-tainting, while the latter receive PCS treatment by default to capture secrets and keep the annotation burden low.

In addition, PC^{2}S is at least as efficient as PCS because propagation policy ρ is partially evaluated at compile-time. Thus, the choice of NCS or PCS semantics for each pointer operation is decided purely statically, conditional upon the static types of the operands. The appropriate specialized propagation implementation is then in-lined into the resulting object code during compilation.

Example. To illustrate how each semantics propagate taint, consider the IL pseudo-code in Listing 3, which revisits the linked-list example informally presented in §2.2. Input stream s includes a non-secret request identifier and a secret key of primitive type (e.g., unsigned long).

If one labels stream s secret, then the public $request\_id$ becomes over-tainted in all three semantics, which is undesirable because a redaction of $request\_id$ may crash the program (when $request\_id$ is later used as an array index). A better solution is to label pointer p secret and employ PCS, which correctly labels the key at the moment it is stored. However, PCS additionally taints the $next$-pointer, leading to over-tainting of all the nodes in the containing linked-list, some of which may contain keys owned by other users. PC^{2}S avoids this over-tainting by exempting the next pointer from the combine-semantics. This preserves the data structure while correctly labeling the secret data it contains.

Listing 3: IL pseudo-code for storing public ids and secret keys from an unstructured input stream into a linked list.

```
store(id, request_id, get(s, id_size));
store(key, p[request_id]->key, get (s, key_size));
store(ctx_t*, p[request_id]->next, queue_head);
```

4 Implementation

Figure 6 presents an architectural overview of our implementation, SignaC[1] (Secret Information Graph iNstrumentation for Annotated C). At a high level, the implementation consists of three components: (1) a source-to-source preprocessor, which (a) automatically propagates user-supplied, source-level type annotations to containing datatypes, and (b) in-lines taint introduction logic into dynamic memory allocation operations; (2) a modified LLVM compiler that instruments programs with PC^{2}S taint propagation logic during compilation; and (3) a runtime library that the instrumented code invokes during program execution to introduce taints and perform redaction. Each component is described below.

4.1 Source-Code Rewriting

Type attributes. Users first annotate data structures containing secrets with the type qualifier `SECRET`. This instructs the taint-tracker to treat all instantiations (e.g., dynamic allocations) of these structures as taint sources. Additionally, qualifier `NONSECRET` may be applied to pointer fields within these structures to exempt them from PCS. The instrumentation pass generates NCS logic instead for operations involving such members. Finally, qualifier `SECRET_STR` may be applied to pointer fields whose destinations are dynamic-length byte sequences bounded by a null terminator (strings).

To avoid augmenting the source language's grammar, these type qualifiers are defined using source-level attributes (specified with `_attribute_`) followed by a specifier. `SECRET` uses the `annotate` specifier, which defines a purely syntactic qualifier visible only at the compiler's front-end. In contrast, `NONSECRET` and `SECRET_STR` are required during the back-end instrumentation. To this end, we leverage Quala [39], which extends LLVM with an overlay type system. Quala's `type_annotate` specifier propagates the type qualifiers throughout the IL code.

[1] named after *pointillism* co-founder Paul Signac

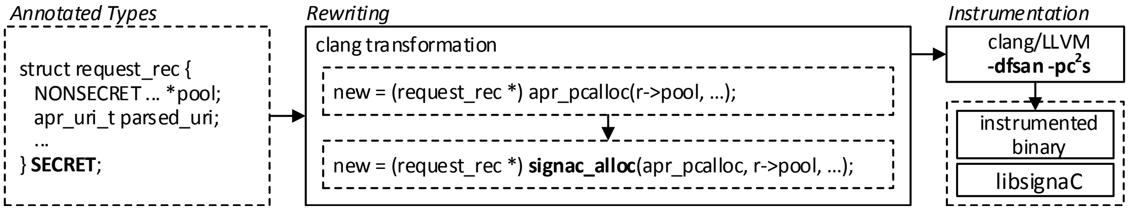


Figure 6: Architectural overview of SignaC illustrating its three-step, static instrumentation process: (1) annotation of security-relevant types, (2) source-code rewriting, and (3) compilation with the sanitizer's instrumentation pass.

Type attribute rewriting. In the preprocessing step, the target application undergoes a source-to-source transformation pass that rewrites all dynamic allocations of annotated data types with taint-introducing wrappers. Implementing this transformation at the source level allows us to utilize the full type information that is available at the compiler's front-end, including purely syntactic attributes such as `SECRET` annotations.

Our implementation leverages Clang's tooling API [12] to traverse and apply the desired transformations directly into the program's AST. At a high-level, the rewriting algorithm takes the following steps:

1. It first amasses a list of all *security-relevant datatypes*, which are defined as (a) all structs and unions annotated `SECRET`, (b) all types defined as aliases (e.g., via typedef) of security-relevant datatypes, and (c) all structs and unions containing secret-relevant datatypes not separated from the containing structure by a level of pointer indirection (e.g., nested struct definitions). This definition is recursive, so the list is computed iteratively from the transitive closure of the graph of datatype definition references.
2. It next finds all calls to memory allocation functions (e.g., malloc, calloc) whose return values are *explicitly* or *implicitly* cast to a security-relevant datatype. Such calls are wrapped in calls to SignaC's runtime library, which dynamically introduces an appropriate taint label to the newly allocated structure.

The task of identifying memory allocation functions is facilitated by a user-supplied list that specifies the memory allocation API. This allows the rewriter to handle programs that employ custom memory management. For example, Apache defines custom allocators in its Apache Portable Runtime (APR) memory management interface.

4.2 PC^{2}S Instrumentation

The instrumentation pass next introduces LLVM IR code during compilation that propagates taint labels during program execution. Our implementation extends DFSan with the PC^{2}S label propagation policy specified in §3.

Taint representation. To support a large number of taint labels, DFSan adopts a low-overhead representation of labels as 16-bit integers, with new labels allocated sequentially from a pool. Rather than reserving 2^n labels to represent the full power set of a set of n primitive taints, DFSan lazily reserves labels denoting non-singleton sets on-demand. When a label union operation is requested at a join point (e.g., during binary operations on tainted operands), the instrumentation first checks whether a new label is required. If a label denoting the union has already been reserved, or if one operand label subsumes the other, DFSan returns the already-reserved label; otherwise, it reserves a fresh union label from the label pool. The fresh label is defined by pointers to the two labels that were joined to form it. Union labels are thus organized as a dynamically growing binary DAG—the *union table*.

This strategy benefits applications whose label-joins are sparse, visiting only a small subset of the universe of possible labels. Our PC^{2}S semantics' curtailment of label creep thus synergizes with DFSan's lazy label allocation strategy, allowing us to realize taint-tracking for legacy code that otherwise exceeds the maximum label limit. This benefit is further evidenced in our evaluation (§5).

Table 1 shows the memory layout of an instrumented program. DFSan maps (without reserving) the lower 32 TB of the process address space for *shadow memory*, which stores the taint labels of the values stored at the corresponding application memory addresses. This layout allows for efficient lookup of shadow addresses by masking and shifting the application's addresses. Labels of values not stored in memory (e.g., those stored in machine registers or optimized away at compile-time) are tracked at the IL level in SSA registers, and compiled to suitable taint-tracking object code.

Function calls. Propagation context $\mathcal{A}$ defined in §3 models label propagation across external library function calls, expressed in DFSan as an Application Binary Interface (ABI). The ABI lists functions whose label-propagation

Table 1: Memory layout of an instrumented program.

Start	End	Memory Region
0x700000008000	0x800000000000	application memory
0x200000000000	0x200200000000	union table
0x000000010000	0x200000000000	shadow memory
0x000000000000	0x000000010000	reserved by kernel

behavior (if any) should be replaced with a fixed, user-defined propagation policy at call sites. For each such function, the ABI specifies how the labels of its arguments relate to the label of its return value.

DFSan natively supports three such semantics: (1) *discard*, which corresponds to propagation function $\rho_{\mathit{dis}}(\overline{\gamma}) := \bot$ (return value is unlabeled); (2) *functional*, corresponding to propagation function $\rho_{\mathit{fun}}(\overline{\gamma}) := \bigsqcup\overline{\gamma}$ (label of return value is the union of labels of the function arguments); and (3) *custom*, denoting a custom-defined label propagation wrapper function.

DFSan pre-defines an ABI list that covers glibc's interface. Users may extend this with the API functions of external libraries for which source code is not available or cannot be instrumented. For example, to instrument Apache with `mod_ssl`, we mapped OpenSSL's API functions to the ABI list. In addition, we extended the custom ABI wrappers of *memory transfer functions* (e.g., `strcpy`, `strdup`) and *input functions* (e.g., `read`, `pread`) to implement PC^{2}S. For instance, we modified the wrapper for `strcpy(`$dest$`,`src`)` to taint $dest$ with $\gamma_{src} \sqcup \gamma_{dest}$ when instrumenting code under PC^{2}S.

Static instrumentation. The instrumentation pass is placed at the end of LLVM's optimization pipeline. This ensures that only memory accesses surviving all compiler optimizations are instrumented, and that instrumentation takes place just before target code is generated. Like other LLVM *transform* passes, the program transformation operates on LLVM IR, traversing the entire program to insert label propagation code. At the front-end, compilation flags parametrize the label propagation policies for the store and load operations discussed in §3.3.

String handling. Strings in C are not first-class types; they are implemented as character pointers. C's type system does not track their lengths or enforce proper termination. This means that purely static typing information is insufficient for the instrumentation to reliably identify strings or propagate their taints to all constituent bytes on store. To overcome this problem, users must annotate secret-containing, string fields with `SECRET_STR`. This cues the runtime library to taint up to and including the pointee's null terminator when a string is assigned to such a field. For safety, our runtime library (see §4.3) zeros the first byte of all fresh memory allocations, so that uninitialized strings are always null-terminated.

Store instructions. Listing 4 summarizes the instrumentation procedure for stores in diff style. By default, DFSan instruments NCS on store instructions: it reads the shadow memory of the value operand (line 1) and copies it onto the shadow of the pointer operand (line 10). If PC^{2}S is enabled (lines 2 and 11), the instrumentation consults the static type of the value operand and checks whether it is a non-pointer or non-exempt pointer field (which also subsumes `SECRET_STR`) in lines 3–4. If so, the shadows of the pointer and value operands are joined (lines 5–6), and the resulting label is stored into the shadow of the pointer operand. If the instruction stores a string annotated with `SECRET_STR`, the instrumentation calls a runtime library function that copies the computed shadow to all bytes of the null-terminated string (lines 12–15).

Listing 4: Store instruction instrumentation

```
 1     Value* Shadow = DFSF.getShadow(SI.getValueOperand());
 2  +  if (Cl_PC2S_OnStore) {
 3  +    Type *t = SI.getValueOperand()->getType();
 4  +    if (!t->isPointerTy() || !isExemptPtr(&SI)) {
 5  +      Value *PtrShadow = DFSF.getShadow(SI.getPointerOperand());
 6  +      Shadow = DFSF.combineShadows(Shadow, PtrShadow, &SI);
 7  +    }
 8  +  }
 9     ...
10     DFSF.storeShadow(SI.getPointerOperand(), Size, Align, Shadow, &SI);
11  +  if (Cl_PC2S_OnStore) {
12  +    if (isSecretStr(&SI)) {
13  +      Value *Str = IRB.CreateBitCast(v, Type::getInt8PtrTy(Ctx));
14  +      IRB.CreateCall2(DFSF.DFS.DFSanSetLabelStrFn, Shadow, Str);
15  +    }
16  +  }
```

Listing 5: Load instruction instrumentation

```
1    Value *Shadow = DFSF.loadShadow(LI.getPointerOperand(), Size, ...);
2  + if (Cl_PC2S_OnLoad) {
3  +   if (!isExemptPtr(&LI)) {
4  +     Value *PtrShadow = DFSF.getShadow(LI.getPointerOperand());
5  +     Shadow = DFSF.combineShadows(Shadow, PtrShadow, &LI);
6  +   }
7  + }
8    ...
9    DFSF.setShadow(&LI, Shadow);
```

Load instructions. Listing 5 summarizes the analogous instrumentation for load instructions. First, the instrumentation loads the shadow of the value pointed by the pointer operand (line 1). If PC^{2}S is enabled (line 2), then the instrumentation checks whether the dereferenced pointer is tainted (line 3). If so, the shadow of the pointer operand is joined with the shadow of its value (lines 4–5), and the resulting label is saved to the shadow (line 9).

Memory transfer intrinsics. LLVM defines intrinsics for standard memory transfer operations, such as `memcpy` and `memmove`. These functions accept a source pointer src, a destination pointer dst, and the number of bytes len to be transferred. DFSan's default instrumentation destructively copies the shadow associated with src to the shadow of dst, which is not the intended propagation policy of PC^{2}S. We therefore instrument these functions as shown in Listing 6. The instrumentation reads the shadows of src and dst (lines 2–3), computes the union of the two shadows (line 4), and stores the combined shadows to the shadow of dst (line 5).

4.3 Runtime Library

Runtime support for the type annotation mechanism is encapsulated in a tiny C library, allowing for low coupling

Listing 6: Memory transfer intrinsics instrumentation

```
+ if (Cl_PC2S_OnStore && !isExemptPtr(&I)) {
+   Value *DestShadow = DFSF.getShadow(I.getDest());
+   Value *SrcShadow = DFSF.getShadow(I.getSource());
+   DestShadow = DFSF.combineShadows(SrcShadow, DestShadow, &I);
+   DFSF.storeShadow(I.getDest(), Size, Align, DestShadow, &I);
+ }
```

Listing 7: Taint-introducing memory allocations

```
1  #define signac_alloc(alloc, args...) ({ \
2    void *__p = alloc ( args ); \
3    signac_taint(&__p, sizeof(void*)); \
4    __p; })
```

between a target application and the sanitizer's logic. The source-to-source rewriter and instrumentation phases inline logic that calls this library at runtime to introduce taints, handle special taint-propagation cases (e.g., string support), and check taints at sinks (e.g., during redaction). The library exposes three API functions:

- `signac_init(`pl`)`: initialize a tainting context with a fresh label instantiation pl for the current principal.
- `signac_taint(`$addr$`,`$size$`)`: taint each address in interval $[addr, addr{+}size)$ with pl.
- `signac_alloc(`$alloc$`,...)`: wrap allocator $alloc$ and taint the address of its returned pointer with pl.

Function `signac_init` instantiates a fresh taint label and stores it in a thread-global context, which function f of annotation SECRET$\langle f \rangle$ may consult to identify the owning principal at taint-introduction points. In typical web server architectures, this function is strategically hooked at the start of a new connection's processing cycle. Function `signac_taint` sets the labels of each address in interval $[addr, addr{+}size)$ with the label pl retrieved from the session's context.

Listing 7 details `signac_alloc`, which wraps allocations of SECRET-annotated data structures. This variadic macro takes a memory allocation function *alloc* and its arguments, invokes it (line 2), and taints the address of the pointer returned by the allocator (line 3).

4.4 Apache Instrumentation

To instrument a particular server application, such as Apache, our approach requires two small, one-time developer interventions: First, add a call to `signac_init` at the start of a user session to initialize a new tainting context for the newly identified principal. Second, annotate the security-relevant data structures whose instances are to be tracked. For instance, in Apache, `signac_init` is called upon the acceptance of a new server connection, and annotated types include `request_rec`, `connection_rec`, `session_rec`, and `modssl_ctx_t`. These structures are where Apache stores URI parameters and request content information, private connection data such as remote IPs, key-value entries in user sessions, and encrypted connection information.

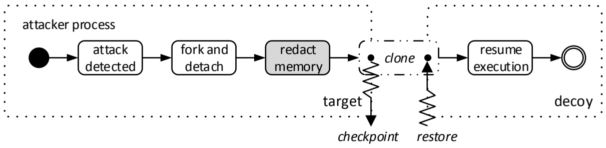


Figure 7: Honey-patch response to an intrusion attempt.

5 Evaluation

This section demonstrates the practical advantages and feasibility of our approach for retrofitting large legacy C codes with taint-tracking, through the development and evaluation of a honey-patching memory redaction architecture for three production web servers. All experiments were performed on a quad-core VM with 8 GB RAM running 64-bit Ubuntu 14.04. The host machine is an Intel Xeon E5645 workstation running 64-bit Windows 7.

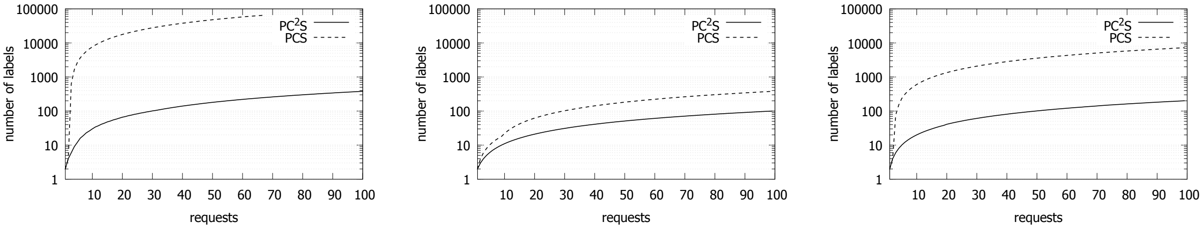
5.1 Honey-patching

Figure 7 illustrates how honey-patches respond to intrusions by cloning attacker sessions to decoys. Upon intrusion detection, the honey-patch forks a shallow, local clone of the victim process. The cloning step redacts all secrets from the clone's address space, optionally replacing them with honey-data. It then resumes execution in the decoy by emulating an unpatched implementation. This impersonates a successful intrusion, luring the attacker away from vulnerable victims, and offering defenders opportunities to monitor and disinform adversaries.

Prior honey-patches implement secret redaction as a brute-force memory sweep that identifies and replaces plaintext string secrets. This is both slow and unsafe; the sweep constitutes a majority of the response delay overhead during cloning [2], and it can miss binary data secrets difficult to express reliably as regular expressions. Using SignaC, we implemented an information flow-based redaction strategy for honey-patching that is faster and more reliable than prior approaches.

Our redaction scheme instruments the server with dynamic taint-tracking. At redaction time, it scans the resulting shadow memory for labels denoting secrets owned by user sessions other than the attacker's, and redacts such secrets. The shadow memory and taint-tracking libraries are then unloaded, leaving a decoy process that masquerades as undefended and vulnerable.

Evaluated software. We implemented taint tracking-based honey-patching for three production web servers: Apache, Nginx, and Lighttpd. Apache and Nginx are the top two servers of all active websites, with 50.1% and 14.8% market share, respectively [32]. Apache comprises 2.27M SLOC mostly in C [35]. Nginx and Lighttpd are smaller, having about 146K and 138K SLOC, respectively. All three are commercial-grade, feature-rich, open-source



(a) Apache

(b) Nginx

(c) Lighttpd

Figure 8: Experiment comparing label creeping behavior of PC^{2}S and PCS on Apache, Nginx, and Lighttpd.

software products without any built-in support for information flow tracking.

To augment these products with PC^{2}S-style taint-tracking support, we manually annotated secret-storing structures and pointer fields. Altogether, we added approximately 45, 30, and 25 such annotations to Apache, Nginx, and Lighttpd, respectively. For consistent evaluation comparisons, we only annotated Apache's core modules for serving static and dynamic content, encrypting connections, and storing session data; we omitted its optional modules. We also manually added about 20–30 SLOC to each server to initialize the taint-tracker. Considering the sizes and complexity of these products, we consider the PC^{2}S annotation burden exceptionally light relative to prior approaches.

5.2 Taint Spread

Over-tainting protection. To test our approach's resistance to taint explosions, we submitted a stream of (non keep-alive) requests to each instrumented web server, recording a cumulative tally of distinct labels instantiated during taint-tracking. Figure 8 plots the results, comparing traditional PCS to our PC^{2}S extensions. On Apache, traditional PCS is impractical, exceeding the maximum label limit in just 68 requests. In contrast, PC^{2}S instantiates vastly fewer labels (note that the y-axes are *logarithmic scale*). After extrapolation, we conclude that an average 16,384 requests are required to exceed the label limit under PC^{2}S—well above the standard 10K-request TTL limit for worker threads.

Taint spread control is equally critical for preserving program functionality after redaction. To demonstrate, we repeated the experiment with a simulated intrusion after $n \in [1, 100]$ legitimate requests. Figure 9 plots the cumulative tally of how many bytes received a taint during the history of the run on Apache. In all cases, redaction crashed PCS-instrumented processes cloned after just 2–3 legitimate requests (due to erasure of over-tainted bytes). In contrast, PC^{2}S-instrumented processes never crashed; their decoy clones continued running after redaction, impersonating vulnerable servers. This demonstrates our approach's facility to realize effective taint-tracking in legacy codes for which prior approaches fail.

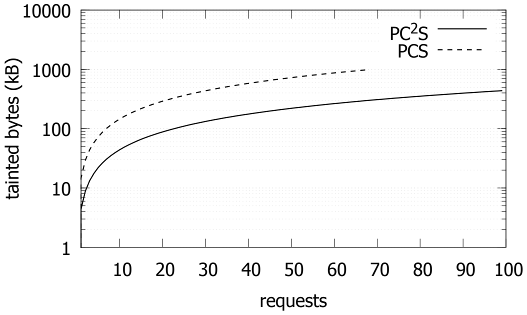


Figure 9: Cumulative tally of bytes tainted on Apache.

Table 2: Honey-patched security vulnerabilities

Software	Version	CVE-ID	Description
Bash[1]	4.3	CVE-2014-6271	Improper parsing of environment variables
OpenSSL[1]	1.0.1f	CVE-2014-0160	Buffer over-read in heartbeat protocol extension
Apache	2.2.21	CVE-2011-3368	Improper URL validation
Apache	2.2.9	CVE-2010-2791	Improper timeouts of keep-alive connections
Apache	2.2.15	CVE-2010-1452	Bad request handling
Apache	2.2.11	CVE-2009-1890	Request content length out of bounds
Apache	2.0.55	CVE-2005-3357	Bad SSL protocol check

[1] tested with Apache 2.4.6

Under-tainting protection. To double-check that PC^{2}S redaction was actually erasing all secrets, we created a workload of legitimate post requests with pre-seeded secrets to a web-form application. We then automated exploits of the honey-patched vulnerabilities listed in Table 2, including the famous Shellshock and Heartbleed vulnerabilities. For each exploit, we ran the legacy, brute-force memory sweep redactor after SignaC's redactor to confirm that the former finds no secrets missed by the latter. We also manually inspected memory dumps of each clone to confirm that none of the pre-seeded secrets

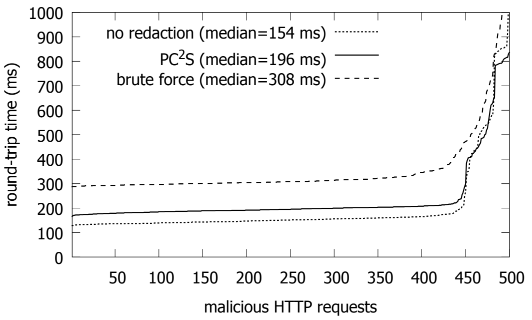


Figure 10: Request round-trip times for attacker session forking on honey-patched Apache.

survived. In all cases, the honey-patch responds to the exploits as a vulnerable decoy server devoid of secrets.

5.3 Performance

Redaction performance. To evaluate the performance overhead of redacting secrets, we benchmarked three honey-patched Apache deployments: (1) a baseline instance without memory redaction, (2) brute-force memory sweep redaction, and (3) our PC^{2}S redactor. We used Apache's server benchmarking tool (*ab*) to launch 500 malicious HTTP requests against each setup, each configured with a pool of 25 decoys.

Figure 10 shows request round-trip times for each deployment. PC^{2}S redaction is about $1.6\times$ faster than brute-force memory sweep redaction; the former's request times average 0.196s, while the latter's average 0.308s. This significant reduction in cloning delay considerably improves the technique's deceptiveness, making it more transparent to attackers. Nginx and Lighttpd also exhibit improved response times of 16% (0.165s down to 0.138s) and 21% (0.155s down to 0.122s), respectively.

Taint-tracking performance. To evaluate the performance overhead of the static instrumentation, three Apache setups were tested: a static-content HTML website ($\sim$20 KB page size), a CGI-based Bash application that returns the server's environment variables, and a dynamic PHP website displaying the server's configuration. For each web server setup, *ab* was executed with four concurrency levels c (i.e., the number of parallel threads). Each run comprises 500 concurrent requests, plotted in ascendant order of their round-trip times (RTT).

Figure 11 shows the results for $c = 1, 10, 50$, and 100, and the average overheads observed for each test profile are summarized in Table 3. Our measurements show overheads of $2.4\times$, $1.1\times$, and $0.3\times$ for the static-content, CGI, and PHP websites, respectively, which is consistent with dynamic taint-tracking overheads reported in the prior literature [41]. Since server computation accounts for only about 10% of overall web site response delay in practice [44], this corresponds to observable overheads of about 24%, 11%, and 3% (respectively).

Table 3: Average overhead of instrumentation

Benchmark	$c = 1$	$c = 10$	$c = 50$	$c = 100$
Static	2.50	2.34	2.56	2.32
CGI Bash	1.29	0.98	1.00	0.97
PHP	0.41	0.37	0.30	0.31

While such overhead characterizes feasibility, it is irrelevant to deception because unpatched, patched, and honey-patched vulnerabilities are all slowed equally by the taint-tracking instrumentation. The overhead therefore does not reveal which apparent vulnerabilities in a given server instance are genuine patching lapses and which are deceptions, and it does not distinguish honey-patched servers from servers that are slowed by any number of other factors (e.g., fewer computational resources). In addition, it is encouraging that high relative overheads were observed primarily for static websites that perform little or no significant computation. This suggests that the more modest 3% overhead for computationally heavier PHP sites is more representative of servers for which computational performance is an issue.

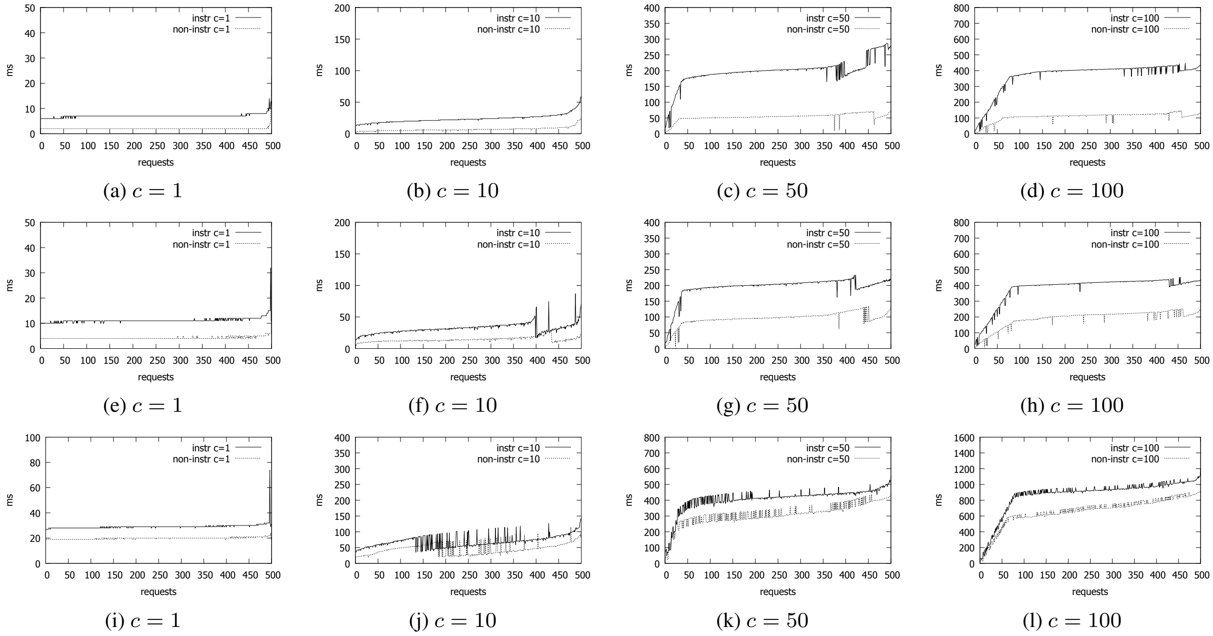
6 Discussion

6.1 Approach Limitations

Our research significantly eases the task of tracking secrets within standard, pointer-linked, graph data-structures as they are typically implemented in low-level languages, like C/C++. However, there are many non-standard, low-level programming paradigms that our approach does not fully support automatically. Such limitations are discussed below.

Pointer Pre-aliases. PC^{2}S fully tracks all pointer aliases via taint propagation starting from the point of taint-introduction (e.g., the code point where a secret is first assigned to an annotated structure field after parsing). However, if the taint-introduction policy misidentifies secret sources too late in the program flow, dynamic tracking cannot track pointer *pre-aliases*—aliases that predate the taint-introduction. For example, if a program first initializes string p_1, then aliases $p_2 := p_1$, and finally initializes secret-annotated field f via $f := p_1$, PC^{2}S automatically labels p_1 (and f) but not pre-alias p_2.

In most cases this mislabeling of pre-aliases can be mitigated by enabling PC^{2}S both on-load and on-store. This causes secrets stored via p_2 to receive the correct label on-load when they are later read via p_1 or f. Likewise, secrets read via p_2 retain the correct label if they were earlier stored via p_1 or f. Thus, only data stored *and* read purely using independent pre-alias p_2 remain untainted.



(a) $c = 1$ (b) $c = 10$ (c) $c = 50$ (d) $c = 100$

(e) $c = 1$ (f) $c = 10$ (g) $c = 50$ (h) $c = 100$

(i) $c = 1$ (j) $c = 10$ (k) $c = 50$ (l) $c = 100$

Figure 11: Dynamic taint-tracking performance (measured in request round-trip times) with varying concurrency c for a static web site (a–d), Bash CGI application (e–h), and PHP application (i–l).

This is a correct enforcement of the user's policy, since the policy identifies $f := p_1$ as the taint source, not p_2. If this treatment is not desired, the user must therefore specify a more precise policy that identifies the earlier origin of p_1 as the true taint source (e.g., by manually inserting a dynamic classification operation where p_1 is born), rather than identifying f as the taint source.

Structure granularity. Our automation of taint-tracking for graph data-structures implemented in low-level languages leads to taint annotations at the granularity of whole struct declarations, not individual value fields. Thus, all non-pointer fields within a secret-annotated C struct receive a common taint under our semantics. This coarse granularity is appropriate for C programs since such programs can (and often do) refer to multiple data fields within a given struct instance using a common pointer. For example, *marshalling* is typically implemented as a pointer-walk that reads a byte stream directly into all data fields (but not the pointer fields) of a struct instance byte-by-byte. All data fields therefore receive a common label after marshalling.

Reliable support for structs containing secrets of mixed taint therefore requires a finer-grained taint-introduction policy than is expressible by declarative annotations of C structure definitions. Such policies must be operationally specified in C through runtime classifications at secret-introducing code points. Our focus in this research is on automating the much more common case where each node of the graph structure holds secrets of uniform classification, toward lifting the user's annotation burden for this most common case.

Dynamic-length secrets. Our implementation provides built-in support for a particularly common form of dynamic-length secret—null-terminated strings. This can be extended to support other forms of dynamic-length secrets as needed. For example, strings with an explicit length count rather than a terminator, fat and bounded pointers [26], and other variable-length, dynamically allocated, data structures can be supported through the addition of an appropriate annotation type and a dynamic taint-propagating function that extends pointer taints to the entire pointee during assignments.

Implicit Flows. Our dynamic taint-tracking tracks explicit information flows, but not implicit flows that disclose information through control-flows rather than dataflows. Tracking implicit flows generally requires static information flow analysis to reason about disclosures through inaction (non-observed control-flows) rather than merely actions. Such analysis is often intractable (and generally undecidable) for low-level languages like C, whose control-flows include unstructured and dynamically computed transitions.

Likewise, dynamic taint-tracking does not monitor side-channels, such as resource consumption (e.g., memory or power consumption), runtimes, or program termination, which can also divulge information. For our problem domain (program process redaction), such channels are largely irrelevant, since attackers may only exfiltrate information after redaction, which leaves no secrets for the attacker to glean, directly or indirectly.

6.2 Process Memory Redaction

Our research introduces live process memory image sanitization as a new problem domain for information flow analysis. Process memory redaction raises unique challenges relative to prior information flow applications. It is exceptionally sensitive to over-tainting and label creep, since it must preserve process execution (e.g., for process debugging, continued service availability, or attacker deception); it demands exceptionally high performance; and its security applications prominently involve large, low-level, legacy codes, which are the most frequent victims of cyber-attacks. Future work should expand the search for solutions to this difficult problem to consider the suitability of other information flow technologies, such as static type-based analyses.

6.3 Language Compatibility

While our implementation targets one particularly ubiquitous source language (C/C++), our general approach is applicable to other similarly low-level languages, as well as scripting languages whose interpreters are implemented in C (e.g., PHP, Bash). Such languages are common choices for implementing web services, and targeting them is therefore a natural next step for the web security thrust of our research.

7 Related Work

Dynamic tracking of in-memory secrets. Dynamic taint-tracking lends itself as a natural technique for tracking secrets in software. It has been applied to study sensitive data lifetime (i.e., propagation and duration in memory) in commodity applications [10, 11], analyze spyware behavior [19, 48], and impede the propagation of secrets to unauthorized sinks [21, 23, 49].

TaintBochs [10] uses whole-system simulation to understand secret propagation patterns in several large, widely deployed applications, including Apache, and implements *secure deallocation* [11] to reduce the risk of exposure of in-memory secrets. Panorama [48] builds a system-level information-flow graph using process emulation to identify malicious software tampering with information that was not intended for their consumption. Egele et al. [19] also utilize whole-system dynamic tainting to analyze spyware behavior in web browser components. While valuable, the performance impact of whole-system analyses—often on the order of 2000% [10, 19, 48]—remains a significant obstacle, rendering such approaches impractical for most live, high-performance, production server applications.

More recently, there has been growing interest in runtime detection of information leaks [21, 49]. For instance, TaintDroid [21] extends Android's virtualized architecture with taint-tracking support to detect misuses of users' private information across mobile apps. TaintEraser [49] uses dynamic instrumentation to apply taint analysis on binaries for the purpose of identifying and blocking information leaking to restricted output channels. To achieve this, it monitors and rewrites sensitive bytes escaping to the network and the local file system. Our work adopts a different strategy to instrument secret-redaction support into programs, resulting in applications that can proactively respond to attacks by self-censoring their address spaces with minimal overhead.

Pointer taintedness. In security contexts, many categories of widely exploited, memory-overwrite vulnerabilities (e.g., format string, memory corruption, buffer overflow) have been recognized as detectable by dynamic taint-checking on pointer dereferences [7, 8, 15, 16, 28]. Hookfinder [47] employs data and pointer tainting semantics in a full-system emulation approach to identify malware hooking behaviors in victim systems. Other systems follow a similar technique to capture system-wide information-flow and detect privacy-breaching malware [19, 48].

With this high practical utility come numerous theoretical and practical challenges for effective pointer tainting [17, 27, 43]. On the theoretical side, there are varied views of how to interpret a pointer's label. (Does it express a property of the pointer value, the values it points to, values read or stored by dereferencing the pointer, or all three?) Different taint tracking application contexts solicit differing interpretations, and the differing interpretations lead to differing taint-tracking methodologies. Our contributions include a pointer tainting methodology that is conducive to tracking in-memory secrets.

On the practical side, imprudent pointer tainting often leads to taint explosion in the form of over-tainting or label-creep [40, 43]. This can impair the feasibility of the analysis and increase the likelihood of crashes in programs that implement data-rewriting policies [49]. To help overcome this, sophisticated strategies involving pointer injection (PI) analysis have been proposed [16, 28]. PI uses a taint bit to track the flow of legitimate pointers and another bit to track the flow of untrusted data, disallowing dereferences of tainted values that do not have a corresponding pointer tainted. Our approach uses static typing information in lieu of PI bits to achieve lower runtime overheads and broader compatibility with low-level legacy code.

Application-level instrumentation. Much of the prior work on dynamic taint analysis has employed dynamic binary instrumentation (DBI) frameworks [9,13,29,33,38,49] to enforce taint-tracking policies on software. These approaches do not require application recompilation, nor do they depend on source code information.

However, despite many optimization advances over the years, dynamic instrumentation still suffers from significant performance overheads, and therefore cannot support high-performance applications, such as the redaction speeds required for attacker-deceiving honey-patching of production server code. Our work benefits from research advances on static-instrumented, dynamic data flow analysis [6, 18, 30, 46] to achieve both high performance and high accuracy by leveraging LLVM's compilation infrastructure to instrument taint-propagating code into server code binaries.

8 Conclusion

PC^{2}S significantly improves the feasibility of dynamic taint-tracking for low-level legacy code that stores secrets in graph data structures. To ease the programmer's annotation burden and avoid taint explosions suffered by prior approaches, it introduces a novel pointer-combine semantics that resists taint over-propagation through graph edges. Our LLVM implementation extends C/C++ with declarative type qualifiers for secrets, and instruments programs with taint-tracking capabilities at compile-time.

The new infrastructure is applied to realize efficient, precise honey-patching of production web servers for attacker deception. The deceptive servers self-redact their address spaces in response to intrusions, affording defenders a new tool for attacker monitoring and disinformation.

9 Acknowledgments

The research reported herein was supported in part by AFOSR Award FA9550-14-1-0173, NSF CAREER Award #1054629, and ONR Award N00014-14-1-0030. Any opinions, recommendations, or conclusions expressed are those of the authors and not necessarily of the AFOSR, NSF, or ONR.

References

[1] Apache. Apache HTTP server project. http://httpd.apache.org, 2014.

[2] Araujo, F., Hamlen, K. W., Biedermann, S., and Katzenbeisser, S. From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation. In *Proc. ACM Conf. Computer and Communications Security (CCS)* (2014), pp. 942–953.

[3] Attariyan, M., and Flinn, J. Automating configuration troubleshooting with dynamic information flow analysis. In *Proc. USENIX Sym. Operating Systems Design and Implementation (OSDI)* (2010), pp. 1–11.

[4] Bauer, L., Cai, S., Jia, L., Passaro, T., Stroucken, M., and Tian, Y. Run-time monitoring and formal analysis of information flows in Chromium. In *Proc. Annual Network & Distributed System Security Sym. (NDSS)* (2015).

[5] Bosman, E., Slowinska, A., and Bos, H. Minemu: The world's fastest taint tracker. In *Proc. Int. Sym. Recent Advances in Intrusion Detection (RAID)* (2011), pp. 1–20.

[6] Chang, W., Streiff, B., and Lin, C. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proc. ACM Conf. Computer and Communications Security (CCS)* (2008), pp. 39–50.

[7] Chen, S., Pattabiraman, K., Kalbarczyk, Z., and Iyer, R. K. Formal reasoning of various categories of widely exploited security vulnerabilities by pointer taintedness semantics. In *Proc. IFIP TC11 Int. Conf. Information Security (SEC)* (2004), pp. 83–100.

[8] Chen, S., Xu, J., Nakka, N., Kalbarczyk, Z., and Iyer, R. K. Defeating memory corruption attacks via pointer taintedness detection. In *Proc. Int. Conf. Dependable Systems and Networks (DSN)* (2005), pp. 378–387.

[9] Cheng, W., Zhao, Q., Yu, B., and Hiroshige, S. TaintTrace: Efficient flow tracing with dynamic binary rewriting. In *Proc. IEEE Sym. Computers and Communications (ISCC)* (2006), pp. 749–754.

[10] Chow, J., Pfaff, B., Garfinkel, T., Christopher, K., and Rosenblum, M. Understanding data lifetime via whole system simulation. In *Proc. USENIX Security Symposium* (2004), pp. 321–336.

[11] Chow, J., Pfaff, B., Garfinkel, T., and Rosenblum, M. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proc. USENIX Security Symposium* (2005), pp. 331–346.

[12] Clang. clang.llvm.org. http://clang.llvm.org.

[13] Clause, J., Li, W., and Orso, A. Dytan: A generic dynamic taint analysis framework. In *Proc. ACM/SIGSOFT Int. Sym. Software Testing and Analysis (ISSTA)* (2007), pp. 196–206.

[14] Cox, L. P., Gilbert, P., Lawler, G., Pistol, V., Razeen, A., Wu, B., and Cheemalapati, S. Spandex: Secure password tracking for Android. In *Proc. USENIX Security Sym.* (2014).

[15] Dalton, M., Kannan, H., and Kozyrakis, C. Raksha: A flexible information flow architecture for software security. In *Proc. Int. Sym. Computer Architecture (ISCA)* (2007), pp. 482–493.

[16] Dalton, M., Kannan, H., and Kozyrakis, C. Real-world buffer overflow protection for userspace & kernelspace. In *Proc. USENIX Security Symposium* (2008), pp. 395–410.

[17] Dalton, M., Kannan, H., and Kozyrakis, C. Tainting is not pointless. *ACM/SIGOPS Operating Systems Review (OSR) 44*, 2 (2010), 88–92.

[18] DFSan. Clang DataFlowSanitizer. http://clang.llvm.org/docs/DataFlowSanitizer.html.

[19] Egele, M., Kruegel, C., Kirda, E., Yin, H., and Song, D. Dynamic spyware analysis. In *Proc. USENIX Annual Technical Conf. (ATC)* (2007), pp. 233–246.

[20] Egele, M., Scholte, T., Kirda, E., and Kruegel, C. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR) 44*, 2 (2012), 1–42.

[21] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. TaintDroid: An information flow tracking system for real-time privacy monitoring on smartphones. *Communications of the ACM (CACM) 57*, 3 (2014), 99–106.

[22] Epigraphic Survey, The Oriental Institute of the University of Chicago, Ed. *Reliefs and Inscriptions at Luxor Temple*, vol. 1–2 of *The University of Chicago Oriental Institute*

Publications. Oriental Institute of the University of Chicago, Chicago, 1994, 1998.

[23] GIBLER, C., CRUSSELL, J., ERICKSON, J., AND CHEN, H. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *Proc. Int. Conf. Trust and Trustworthy Computing (TRUST)* (2012), pp. 291–307.

[24] GU, A. B., LI, X., LI, G., CHAMPION, CHEN, Z., QIN, F., AND XUAN, D. D2Taint: Differentiated and dynamic information flow tracking on smartphones for numerous data sources. In *Proc. IEEE Conf. Computer Communications (INFOCOM)* (2013), pp. 791–799.

[25] HO, A., FETTERMAN, M., CLARK, C., WARFIELD, A., AND HAND, S. Practical taint-based protection using demand emulation. In *Proc. ACM SIGOPS/EuroSys European Conf. Computer Systems (EuroSys)* (2006), pp. 29–41.

[26] JIM, T., MORRISETT, J. G., GROSSMAN, D., HICKS, M. W., CHENEY, J., AND WANG, Y. Cyclone: A safe dialect of C. In *Proc. USENIX Annual Technical Conf. (ATC)* (2002), pp. 275–288.

[27] KANG, M. G., MCCAMANT, S., POOSANKAM, P., AND SONG, D. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proc. Annual Network & Distributed System Security Sym. (NDSS)* (2011).

[28] KATSUNUMA, S., KURITA, H., SHIOYA, R., SHIMIZU, K., IRIE, H., GOSHIMA, M., AND SAKAI, S. Base address recognition with data flow tracking for injection attack detection. In *Proc. Pacific Rim Int. Sym. Dependable Computing (PRDC)* (2006), pp. 165–172.

[29] KEMERLIS, V. P., PORTOKALIDIS, G., JEE, K., AND KEROMYTIS, A. D. Libdft: Practical dynamic data flow tracking for commodity systems. In *Proc. Conf. Virtual Execution Environments (VEE)* (2012), pp. 121–132.

[30] LAM, L. C., AND CHIUEH, T. A general dynamic information flow tracking framework for security applications. In *Proc. Annual Computer Security Applications Conf. (ACSAC)* (2006), pp. 463–472.

[31] LATTNER, C., AND ADVE, V. S. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. IEEE/ACM Int. Sym. Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO)* (2004), pp. 75–88.

[32] NETCRAFT. Web surver survey. http://news.netcraft.com/archives/category/web-server-survey, January 2015.

[33] NEWSOME, J., AND SONG, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. Annual Network & Distributed System Security Sym. (NDSS)* (2005).

[34] NGUYEN-TUONG, A., GUARNIERI, S., GREENE, D., AND EVANS, D. Automatically hardening web applications using precise tainting. In *Proc. IFIP TC11 Int. Conf. Information Security (SEC)* (2005), pp. 372–382.

[35] OHLOH. Apache HTTP server statistics. http://www.ohloh.net/p/apache.

[36] PAPAGIANNIS, I., MIGLIAVACCA, M., AND PIETZUCH, P. PHP Aspis: Using partial taint tracking to protect against injection attacks. In *Proc. USENIX Conf. Web Application Development (WebApps)* (2011).

[37] PORTOKALIDIS, G., SLOWINSKA, A., AND BOS, H. Argos: An emulator for fingerprinting zero-day attacks. In *Proc. ACM SIGOPS/EuroSys European Conf. Computer Systems (EuroSys)* (2006), pp. 15–27.

[38] QIN, F., WANG, C., LI, Z., KIM, H., ZHOU, Y., AND WU, Y. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *Proc. Int. Sym. Microarchitecture (MICRO)* (2006), pp. 135–148.

[39] SAMPSON, A. Quala: Type qualifiers for LLVM/Clang. https://github.com/sampsyo/quala, 2014.

[40] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proc. IEEE Sym. Security & Privacy (S&P)* (2010), pp. 317–331.

[41] SEREBRYANY, K., BRUENING, D., POTAPENKO, A., AND VYUKOV, D. AddressSanitizer: A fast address sanity checker. In *Proc. USENIX Annual Technical Conf. (ATC)* (2012), pp. 309–318.

[42] SEZER, E. C., NING, P., KIL, C., AND XU, J. Memsherlock: An automated debugger for unknown memory corruption vulnerabilities. In *Proc. ACM Conf. Computer and Communications Security (CCS)* (2007), pp. 562–572.

[43] SLOWINSKA, A., AND BOS, H. Pointless tainting?: Evaluating the practicality of pointer tainting. In *Proc. ACM SIGOPS/EuroSys European Conf. Computer Systems (EuroSys)* (2009), pp. 61–74.

[44] SOUDERS, S. *High Performance Web Sites: Essential Knowledge for Front-End Engineers*. O'Reilly, 2007.

[45] SUH, G. E., LEE, J. W., ZHANG, D., AND DEVADAS, S. Secure program execution via dynamic information flow tracking. In *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2004), pp. 85–96.

[46] XU, W., BHATKAR, S., AND SEKAR, R. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proc. USENIX Security Symposium* (2006).

[47] YIN, H., LIANG, Z., AND SONG, D. HookFinder: Identifying and understanding malware hooking behaviors. In *Proc. Annual Network & Distributed System Security Sym. (NDSS)* (2008).

[48] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proc. ACM Conf. Computer and Communications Security (CCS)* (2007), pp. 116–127.

[49] ZHU, D. Y., JUNG, J., SONG, D., KOHNO, T., AND WETHERALL, D. TaintEraser: Protecting sensitive data leaks using application-level taint tracking. *ACM SIGOPS Operating Systems Review (OSR) 45*, 1 (2011), 142–154.

Control-Flow Bending: On the Effectiveness of Control-Flow Integrity

Nicolas Carlini
UC Berkeley

Antonio Barresi
ETH Zurich

Mathias Payer
Purdue University

David Wagner
UC Berkeley

Thomas R. Gross
ETH Zurich

Abstract

Control-Flow Integrity (CFI) is a defense which prevents control-flow hijacking attacks. While recent research has shown that coarse-grained CFI does not stop attacks, fine-grained CFI is believed to be secure.

We argue that assessing the effectiveness of practical CFI implementations is non-trivial and that common evaluation metrics fail to do so. We then evaluate fully-precise static CFI — the most restrictive CFI policy that does not break functionality — and reveal limitations in its security. Using a generalization of non-control-data attacks which we call Control-Flow Bending (CFB), we show how an attacker can leverage a memory corruption vulnerability to achieve Turing-complete computation on memory using just calls to the standard library. We use this attack technique to evaluate fully-precise static CFI on six real binaries and show that in five out of six cases, powerful attacks are still possible. Our results suggest that CFI may not be a reliable defense against memory corruption vulnerabilities.

We further evaluate shadow stacks in combination with CFI and find that their presence for security is necessary: deploying shadow stacks removes arbitrary code execution capabilities of attackers in three of six cases.

1 Introduction

Attacking software systems by exploiting memory-corruption vulnerabilities is one of the most common attack methods today according to the list of Common Vulnerabilities and Exposures. To counter these threats, several hardening techniques have been widely adopted, including ASLR [29], DEP [38], and stack canaries [10]. Each has limitations: stack canaries protect only against contiguous overwrites of the stack, DEP protects against code injection but not against code reuse, and ASLR does not protect against information leakage.

We classify defense mechanisms into two broad categories: *prevent-the-corruption* and *prevent-the-exploit*. Defenses that prevent the corruption stop the actual memory corruption before it can do any harm to the program (i.e., no attacker-controlled values are ever used out-of-context). Examples for prevent-the-corruption defenses are SoftBound [22], Data-Flow Integrity [6], or Code-Pointer Integrity [18]. In contrast, prevent-the-exploit defenses allow memory corruption to occur but protect the application from subsequent exploitation; they try to survive or tolerate adversarial corruption of memory. Examples for prevent-the-exploit defenses are DEP [38] or stack canaries [10].

Control-Flow Integrity (CFI) [1, 3, 12, 15, 27, 30, 31, 39, 41–44] is a promising stateless prevent-the-exploit defense mechanism that aims for complete protection against control-flow hijacking attacks under a threat model with a powerful attacker that can read and write into the process's address space. CFI ensures that program execution follows a valid path through the static Control-Flow Graph (CFG). Any deviation from the CFG is a CFI violation, terminating the application. CFI is not specific to any particular exploitation vector for control-flow hijacking. Rather, it enforces its policy on all indirect branch instructions. Therefore any attempt by an attacker to alter the control-flow in an invalid manner will be detected, regardless of *how* the attacker changes the target of the control-flow transfer instruction.

CFI is often coupled with a protected shadow stack, which ensures that each return statement matches the corresponding call and thereby prevents an attacker from tampering with return addresses. While the foundational work [1, 15] included a shadow stack as part of CFI, some more recent research has explored variants of CFI that omit the shadow stack for better performance [9]. Whereas conformance to the CFG is a stateless policy, shadow stacks are inherently dynamic and are more precise than any static policy can be with respect to returns.

Many prior attacks on CFI have focused on attacking a weak or suboptimal implementation of CFI. Our focus is on evaluating the effectiveness of CFI in its best achievable form, instead of artifacts of some (possibly weak) CFI implementation. We define *fully-precise static CFI* as the best achievable CFI policy as follows: a branch from one instruction to another is allowed if and only if some benign execution makes that same control-flow transfer. Such a policy could be imagined as taking any CFG over-approximation and removing edges until removing additional edges would break functionality.

Thus, fully-precise static CFI is the most restrictive stateless CFI policy that still allows the program to run as intended. Both coarse-grained and fine-grained CFI are less precise than fully-precise static CFI, because they both over-approximate the set of valid targets for each indirect transfer (though to a different degree). In contrast, fully-precise static CFI involves no approximation by definition. We acknowledge that fully-precise static CFI might be stricter than anything that can be practically implemented, but this makes any attacks all the more meaningful: our results help us understand fundamental limits on the effectiveness of the strongest possible CFI policy.

Through several methods of evaluation, we argue that fully-precise static CFI is neither completely broken (as most coarse-grained defenses are) nor totally secure. We explore what CFI can and cannot prevent, and hope that this will stimulate a broader discussion about ways to further strengthen CFI.

We evaluate the security of fully-precise static CFI both with and without shadow stacks. Recent research achieves better performance by omitting the shadow stack in favor of a static policy on return statements. We still call it fully-precise static CFI when we have added a shadow stack, because the shadow stack is orthogonal. This does not change the fact that the CFI policy is static.

CFI works by preventing an attacker from deviating from the control-flow graph. Our attacks do not involve breaking the CFI mechanism itself: we even assume the mechanism is implemented perfectly to its fullest extent. Rather, our analysis demonstrates that an attacker can still create exploits for most real applications, without causing execution to deviate from the control-flow graph.

This paper provides the following contributions:

1. formalization and evaluation of a space of different kinds of CFI schemes;
2. new attacks on fully-precise static CFI, which reveal fundamental limits on the effectiveness of CFI;
3. evidence that existing metrics for CFI security are ineffective;
4. evidence that CFI without a shadow stack is broken;
5. widely applicable Turing-complete attacks on CFI with shadow stacks; and,
6. practical case studies of the security of fully-precise static CFI for several existing applications.

2 Background and software attacks

Over the past few decades, one of the most common attack vectors has been exploitation of memory corruption within programs written in memory-unsafe languages.In response, operating systems and compilers have started to support countermeasures against specific exploitation vectors and vulnerability types, but current hardening techniques are still unable to stop all attacks. We briefly provide an overview of these attacks; more information may be found elsewhere [37].

2.1 Control-Flow Hijacking

One way to exploit a memory corruption bug involves hijacking control flow to execute attacker-supplied or already-existing code in an application's address space. These methods leverage the memory corruption bug to change the target of an indirect branch instruction (*ret*, *jmp* *, or *call* *). By doing so, an attacker can completely control the next instructions to execute.

2.2 Code-Reuse Attacks

Data Execution Prevention (DEP) prevents executing attacker-injected code. However, redirecting control-flow to already-existing executable code in memory remains feasible. One technique, return-to-libc [25, 36], reuses existing functions in the address space of the vulnerable process. Runtime libraries (such as libc) often provide powerful functions, e.g., wrapper functions for most system calls. One example is libc's `system()` function, which allows the attacker to execute shell commands. Code-reuse attacks are possible when attacker-needed code is already available in the address space of a vulnerable process.

2.3 Return Oriented Programming

Return Oriented Programming (ROP) [25, 36] is a more advanced form of code-reuse attack that lets the attacker perform arbitrary computation solely by reusing existing code. It relies upon short instruction sequences (called "gadgets") that end with an indirect branch instruction. This allows them to be chained, so the attacker can perform arbitrary computation by executing a carefully-chosen sequence of gadgets. ROP can be generalized to use indirect jump or call instructions instead of returns [4, 7].

2.4 Non-Control-Data Attacks

A non-control-data attack [8] is an attack where a memory corruption vulnerability is used to corrupt only data, but not any code pointer. (A *code pointer* is a pointer which refers to the code segment, for example, a return address or function pointer.) Depending on the circumstances, these attacks can be as effective as arbitrary code-execution attacks. For instance, corrupting the parameter to a sensitive function (e.g., libc's `execve()`) may allow an attacker to execute arbitrary programs. An attacker may also be able to overwrite security configuration values and disable security checks. Non-control-data attacks are realistic threats and hard to defend against, due to the fact that most defense mechanisms focus on the protection of code pointers.

2.5 Control-Flow Bending

We introduce a generalization of non-control-data attacks which we call *Control-Flow Bending* (CFB). While non-control-data attacks do not directly modify any control-flow data (e.g., return addresses, indirect branch targets), in control-flow bending we allow these modifications so long as the modified indirect branch target is still in the valid set of addresses as defined by the CFI policy (or any other enforced control-flow or code pointer integrity protection). CFB allows an attacker to bend the control-flow of the application (compared to hijacking it) but adheres to an imposed security policy.

We define a "data-only" attack as a non-control-data attack where the entire execution trace is identical to some feasible non-exploit execution trace. (An execution trace is the ordered sequence of instructions which execute, and does not include the effects those instructions have except with respect to control flow.) While data-only attacks may change the control flow of an application, the traces will still look legitimate, as the observed trace can also occur during valid execution. In contrast, CFB is more general: it refers to any attack where each control-flow transfer is within the valid CFG, but the execution trace is not necessarily required to match some valid non-exploit trace.

In general, defense mechanisms implement an abstract machine and can only observe security violations according to the restrictions of that machine, e.g., CFI enforces that control flow follows a finite state machine.

For example, an attacker who directly overwrites the arguments to `exec()` is performing a data-only attack: no control flow has been changed. An attacker who overwrites an `is_admin` flag half-way through processing a request is performing a non-control-data attack: the data that was overwritten is non-control-data, but it affects the control-flow of the program. An attacker who modifies a function pointer to point to a different (valid) call target is mounting a CFB attack.

3 Threat model and attacker goals

Threat model. For this paper we assume a powerful yet realistic threat model. We assume the attacker can write arbitrarily to memory at one point in time during the execution of the program. We assume the process is running with non-executable data and non-writeable code which is hardware enforced.

This threat model is a realistic generalization of memory corruption vulnerabilities: the vulnerability typically gives the attacker some control over memory. In practice there may be a set of specific constraints on what the attacker can write where; however, this is not something a defender can rely upon. To be a robust defense, CFI mechanisms must be able to cope with arbitrary memory corruptions, so in our threat model we allow the attacker full control over memory once.

Limiting the memory corruption to a single point in time does weaken the attacker. However, this makes our attacks all the more meaningful.

Attacker goals. There are three kinds of outcomes an attacker might seek, when exploiting a vulnerability:

1. *Arbitrary code execution:* The attacker can execute arbitrary code and can invoke arbitrary system calls with arbitrary parameters. In other words, the attacker can exercise all permissions that the application has. Code execution might involve injecting new code or re-using already-existing code; from the attacker's perspective, there is no difference as long as the effects are the same.

2. *Confined code execution:* The attacker can execute arbitrary code within the application's address space, but cannot invoke arbitrary system calls. The attacker might be able to invoke a limited set of system calls (e.g., the ones the program would usually execute, or just enough to send information back to the attacker) but cannot exercise all of the application's permissions. Reading and leaking arbitrary memory of the vulnerable program is still possible.

3. *Information leakage:* The attacker can read and leak arbitrary values from memory.

Ideally, a CFI defense would prevent all three attacker goals. The more it can prevent, the stronger the defense.

4 Definition of CFI flavors

Control-Flow Integrity (CFI) [1, 15] adds a stateless check before each indirect control-flow transfer (indirect jump/call, or function return) to ensure that the target location is in a static set defined by the control-flow graph.

4.1 Fully-Precise Static CFI

We define *Fully-Precise Static CFI* as follows: an indirect control-flow transfer along some edge is allowed only if there exists a non-malicious trace that follows that edge. (An execution is not malicious if it exercises only intended program behavior.) In other words, consider the most restrictive control-flow graph that still allows all feasible non-malicious executions, i.e., the CFG contains an edge if and only if that edge is used by some benign execution. Fully-precise static CFI then enforces that execution follows this CFG. Thus, fully-precise static CFI enforces the most precise (and most restrictive) policy possible that does not break functionality.

We know of no way to implement fully-precise static CFI: real implementations often use static analysis and over-approximate the CFG and thus are not fully precise. We do not design a better CFI scheme. The goal of our work is to evaluate the strongest form of CFI that could conceptually exist, and attempt to gain insight on its limitations. This notion of fully-precise static CFI allows us to transcend the recent arms race caused by defenders proposing forms of CFI [9, 28] and then attackers defeating them [5, 14, 16].

4.2 Practical CFI

Practical implementations of CFI are always limited by the precision of the CFG that can be obtained. Current CFI implementations face two sources of over-approximation. First, due to challenges in accurate static analysis, the set of allowed targets for each indirect call instruction typically depends only upon the function pointer type, and this set is often larger than necessary.

Second, most CFI mechanisms use a static points-to analysis to define the set of allowed targets for each indirect control transfer. Due to imprecisions and limitations of the analysis (e.g., aliasing in the case of points-to analysis) several sets may be merged, leading to an over-approximation of allowed targets for individual indirect control-flow transfers. The degree of over-approximation affects the precision and effectiveness of practical CFI mechanisms.

Previous work has classified practical CFI defenses into two categories: coarse-grained and fine-grained. Intuitively, a defense is fine-grained if it is a close approximation of fully-precise static CFI and coarse-grained if there are many unnecessary edges in the sets.

4.3 Stack integrity

The seminal work on CFI [1] combined two mechanisms: restricting indirect control transfers to the CFG, and a shadow call stack to restrict return instructions. The shadow stack keeps track of the current functions on the application call stack, storing the return instruction pointers in a separate region that the attacker cannot access. Each return instruction is then instrumented so that it can only return to the function that called it. For compatibility with exceptions, practical implementations often allow return instructions to return to any function on the shadow stack, not just the one on the top of the stack. As a result, when a protected shadow stack is in use, the attacker has very limited influence over return instructions: all the attacker can do is unwind stack frames. The attacker cannot cause return instructions to return to arbitrary other locations (e.g., other call-sites) in the code.

Unfortunately, a shadow stack does introduce performance overhead, so some modern schemes have proposed omitting the shadow stack [9]. We analyze both the security of CFI with a shadow stack and CFI without a shadow stack. We assume the shadow stack is protected somehow and cannot be overwritten; we do not consider attacks against the implementation of the shadow stack.

5 Evaluating practical CFI

While there has been considerable research on how to make CFI more fine-grained and efficient, most CFI publications still lack a thorough security evaluation. In fact, the security evaluation is often limited to coarse metrics such as Average Indirect target Reduction (AIR) or gadget reduction. Evaluating the security effectiveness of CFI this way does not answer how effective these policies are in preventing actual attacks.

In this section, we show that metrics such as AIR and gadget reduction are not good indicators for the effectiveness of a CFI policy, even for simple programs. We discuss CFI effectiveness and why it is difficult to measure with a single value and propose a simple test that indicates if a CFI policy is trivially broken.

5.1 AIR and gadget reduction

The AIR metric [44] measures the relative reduction in the average number of valid targets for all indirect branch instructions that a CFI scheme provides: without CFI, an indirect branch could target any instruction in the program; CFI limits this to a set of valid targets. The gadget reduction metric measures the relative reduction in the number of gadgets that can be found at locations that are valid targets for an indirect branch instruction.

These metrics measure how effectively a CFI implementation reduces the set of valid targets (or gadgets) for indirect branch instructions, *on average*. However, they fail to capture both (i) the target reduction of individual locations (e.g., a scheme can have high AIR even if one branch instruction has a large set of surplus targets, if the other locations are close to optimal) and (ii) the importance and risk of the allowed control transfers. Similarly, the gadget reduction metric does not weight targets according to their usefulness to an attacker: every code location or gadget is considered to be equally useful.

For example, consider an application with 10MB of executable memory and an AIR of 99%. An attacker would still have 1% of the executable memory at their disposal — 100,000 potential targets — to perform code-reuse attacks. A successful ROP attack requires only a handful of gadgets within these potential targets, and empirically, 100,000 targets is much more than is usually needed to find those gadgets [35]. As this illustrates, averages and metrics that are relative to the code size can be misleading. What is relevant is the absolute number of available gadgets and how useful they are to an attacker.

5.2 CFI security effectiveness

Unfortunately, it is not clear how to construct a single metric that accurately measures the effectiveness of CFI. Ideally, we would like to measure the ability of CFI to stop an attacker from mounting a control-flow hijack attack. More specifically, a CFI effectiveness metric should indicate whether control-flow hijacking and code-reuse attacks are still possible under a certain attacker model or not, and if so, how much harder it is for an attacker to perform a successful attack in the presence of CFI. However, what counts as successful exploitation of a software vulnerability depends on the goals of the attacker (see Section 3) and is not easily captured by a single number.

These observations suggest that assessing CFI effectiveness is hard, especially if no assumptions are made regarding what a successful attack is and what the binary image of the vulnerable program looks like.

5.3 Basic exploitation test

We propose a *Basic Exploitation Test* (BET): a simple test to quickly rule out some trivially broken implementations of CFI. Passing the BET is not a security guarantee, but failing the BET means that the CFI scheme is insecure.

In particular, the BET involves selecting a minimal program — a simple yet representative program that contains a realistic vulnerability — and then determining whether attacks are still possible if that minimal program is protected by the CFI scheme under evaluation. The minimal program should be chosen to use a subset of common run-time libraries normally found in real applications, and constructed so it contains a vulnerability that allows hijacking control flow in a way that is seen in real-life attacks. For instance, the minimal program might allow an attacker to overwrite a return address or the target of an indirect jump/call instruction.

The evaluator then applies the CFI scheme to the minimal program, selects an attacker goal from Section 3, and determines whether that goal is achievable on the protected program. If the attack is possible, the CFI scheme fails the BET. We argue that if a CFI scheme is unable to protect a minimal program it will also fail to protect larger real-life applications, as larger programs afford the attacker even more opportunities than are found in the minimal program.

5.4 BET for coarse-grained CFI

We apply the BET to a representative coarse-grained CFI policy. We show that the scheme is broken, even though its AIR and gadget reduction metrics are high. This demonstrates that AIR and gadget reduction numbers are not reliable indicators for the security effectiveness of a CFI scheme even for small programs. These results generalize the conclusion of recent work [5,14,16], by showing that coarse-grained CFI schemes are broken even for trivially small real-life applications.

Minimal program and attacker goals. Our minimal vulnerable program is shown in Figure 1. It is written in C, compiled with gcc version 4.6.3 under Ubuntu LTS 12.04 for x86 32-bit, and dynamically linked against `ld-linux` and `libc`. The program contains a stack-based buffer overflow. A vulnerability in `vulnFunc()` allows an attacker to hijack the return target of `vulnFunc()` and a memory leak in `memLeak()` allows the attacker to bypass stack canaries and ASLR.

Coarse-grained CFI policy. The coarse-grained CFI policy we analyze is a more precise version of several recently proposed static CFI schemes [43, 44]: each implementation is less accurate than our combined version. We use a similar combined static CFI policy as used in recent work [14, 16].

Our coarse-grained CFI policy has three equivalence classes, one for each indirect branch type. Returns and indirect jumps can target any instruction following a call instruction. Indirect calls can target any defined symbol, i.e., the potential start of any function. This policy is overly strict, especially for indirect jumps; attacking a stricter coarse-grained policy makes our results stronger.

Results. We see in Table 1 that our minimal program linked against its libraries achieves high AIR and gadget reduction numbers for our coarse-grained CFI policy. However, as we will show, all attacker goals from Section 3 can be achieved.

```
#include <stdio.h>
#include <string.h>
#define STDIN 0

void memLeak() {
    char buf[64];
    int nr, i;
    unsigned int *value;
    value = (unsigned int*)buf;
    scanf("%d", &nr);
    for (i = 0; i < nr; i++)
        printf("0x%08x␣", value[i]);
}

void vulnFunc() {
    char buf[1024];
    read(STDIN, buf, 2048);
}

int main(int argc, char* argv[]) {
    setbuf(stdout, NULL);
    printf("echo>␣");
    memLeak();
    printf("\nread>␣");
    vulnFunc();
    printf("\ndone.\n");
    return 0;
}
```

Figure 1: Our minimal vulnerable program that allows hijacking a return instruction target.

	AIR	Gadget red.	Targets	Gadgets
No CFI	0%	0%	1850580	128929
CFI	99.06%	98.86%	19611	1462

Table 1: Basic metrics for the minimal vulnerable program under no CFI and our coarse-grained CFI policy.

```
G1 # arbitrary load (1/2)
   f38ff:       pop    %edx
   f3900:       pop    %ecx
   f3901:       pop    %eax
   f3902:       ret

G2 # arbitrary load (2/2)
   412d2:       add    $0x20,%esp
   412d5:       xor    %eax,%eax
   412d7:       pop    %ebx
   412d8:       pop    %esi
   412d9:       pop    %edi
   412da:       ret

G3 # arbitrary read
   2ee25:       add    $0x1771cf,%ecx
   2ee2b:       mov    0x54(%ecx),%eax
   2ee31:       ret

G4 # arbitrary write
   3fb11:       pop    %ecx
   3fb12:       add    $0xa,%ecx
   3fb18:       mov    %ecx,(%edx)
   3fb1a:       ret

G5 # arbitrary call
   1b008:       mov    %esi,(%esp)
   1b00b:       call   *%edi
```

Figure 2: Our call-site gadgets within libc.

```
000b8d60 <execve>:
...
   b8d72:         call    ...
   b8d77:         add     $0xed27d,%ebx
   b8d7d:         mov     0xc(%esp),%edi
   b8d81:         xchg    %ebx,%edi
   b8d83:         mov     $0xb,%eax
   b8d88:         call    *%gs:0x10
```

Figure 3: Disassembly of libc's execve() function. There is an instruction (0xb8d77) that can be returned to by any return gadget under coarse-grained CFI.

We first identified all gadgets that can be reached without violating the given CFI policy. We found five gadgets that allow us to implement all attacker goals as defined in Section 3. All five gadgets were within libc and began immediately following a call instruction. Two gadgets can be used to load a set of general purpose registers from the attacker-controlled stack and then return. One gadget implements an arbitrary memory write ("write-what-where") and then returns. Another gadget implements an arbitrary memory read and then returns. Finally, we found a fifth gadget — a "call gadget" — that ends with an indirect call through one of the attacker-controlled registers, and thus can be used to perform arbitrary calls. The five gadgets are shown in Figure 2. By routing control-flow through the first four gadgets and then to the call gadget, the attacker can call any function.

The attacker can use these gadgets to execute arbitrary system calls by calling __kernel_vsyscall. In Linux systems (x86 32-bit), system calls are routed through a virtual dynamic shared object (linux-gate.so) mapped into user space by the kernel at a random address. The address is passed to the user space process. If the address is leaked, the attacker can execute arbitrary system calls by calling __kernel_vsyscall using a call gadget. Calls to __kernel_vsyscall are within the allowed call targets as libc itself calls __kernel_vsyscall.

Alternatively, the attacker could call libc's wrappers for each specific system call. For example, the attacker could call execve() within libc to execute the execve system call. Interestingly, if the wrapper functions contain calls, we can directly return to an instruction after such a call and before the system call is issued. For an example, see Figure 3: returning to 0xb8d77 allows us to directly issue the system call without using the call gadget (we simply direct one of the other gadgets to return there). There are some side effects on register ebx and edi but it is straightforward to take them into account.

Arbitrary code execution is also possible. In the absence of CFI, an attacker might write new code somewhere into memory, call mprotect() to make that memory region executable, and then jump to that location to execute the injected code. CFI will prevent this, as the location of the injected code will never be in one of the target sets. We bypass this protection by using `mprotect()` to make already-mapped code writeable. The attacker can overwrite these already-available code pages with malicious code and then transfer control to it using our call gadget. The result is that the attacker can inject and execute arbitrary code and invoke arbitrary system calls with arbitrary parameters. As an alternative `mmap()` could also be used to allocate readable and executable memory (if not prohibited).

The minimal program shown in Figure 1 contains a vulnerability that allows the attacker to overwrite a return address on the stack. We also analyzed other minimal programs that allow the attacker to hijack an indirect jump or indirect call instruction, with similar results. We omit the details of these analyses for brevity. A minimal vulnerable program for initial indirect jump or indirect call hijacking is found in Appendix A.

Based on these results we conclude that coarse-grained CFI policies are not effective in protecting even small and simple programs, such as our minimal vulnerable program example. Our analysis also shows that AIR and gadget reduction metrics fail to indicate whether a CFI scheme is effective at preventing attacks; if such attacks are possible on a small program, then attacks will be easier on larger programs where the absolute number of valid locations and gadgets is even higher.

6 Attacks on Fully-Precise Static CFI

We now turn to evaluating fully-precise static CFI. Recall from Section 2.5 that we define control-flow bending as a generalization of non-control-data attacks. We examine the potential for control-flow bending attacks on CFI schemes with and without a shadow stack.

6.1 Necessity of a shadow stack

To begin, we argue that CFI must have a shadow stack to be a strong defense. Without one, an attacker can easily traverse the CFG to reach almost any program location desired and thereby break the CFI scheme.

For a static, stateless policy like fully-precise static CFI without a shadow stack, the best possible policy for returns is to allow return instructions within a function F to target any instruction that follows a call to F. However, for functions that are called often, this set can be very large. For example, the number of possible targets for the return statements in `malloc()` is immense. Even though dynamically only one of these should be allowed at any given time, a stateless policy must allow all of these edges.

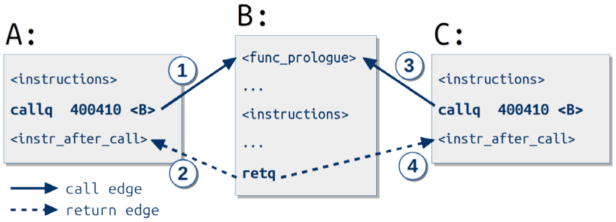


Figure 4: A control-flow graph where the lack of a shadow stack allows an attacker to mount a control-flow bending attack.

This is elaborated in Figure 4. Functions A and C both contain calls to function B. The return in function B must therefore be able to target the instruction following both of these calls. In normal execution, the program will execute edge 1 followed by edge 2, or edge 3 followed by edge 4. However, an attacker may be able to cause edge 3 to be followed by edge 2, or edge 1 to be followed by edge 4.

In practice this is even more problematic with tail-call optimizations, when signal handlers are used, or when the program calls `setjmp/longjmp`. We ignore these cases. This makes our job as an attacker more difficult, but we base our attacks on the fundamental properties of CFI instead of corner cases which might be handled separately.

6.1.1 Dispatcher functions

For an attacker to cause a function to return to a different location than it was called from, she must be able to overwrite the return address on the stack after the function is called yet before it returns. This is easy to arrange when the memory corruption vulnerability occurs within that specific function. However, often the vulnerability is found in uncommonly called (not well tested) functions.

To achieve more power, we make use of *dispatcher functions* (analogous to dispatcher gadgets for JOP [4]). A dispatcher function is one that can overwrite its own return address when given arguments supplied by an attacker. If we can find a dispatcher function that will be called later and use the vulnerability to control its arguments, we can make it overwrite its own return address. This lets us return to any location where this function was called.

Any function that contains a "write-what-where" primitive when the arguments are under the attacker's control can be used as a dispatcher function. Alternatively, a function that can write to only limited addresses can still work as long as the return address is within the limits. Not every function has this property, but a significant fraction of all functions do. For example, assume we control all of the arguments to `memcpy()`. We can

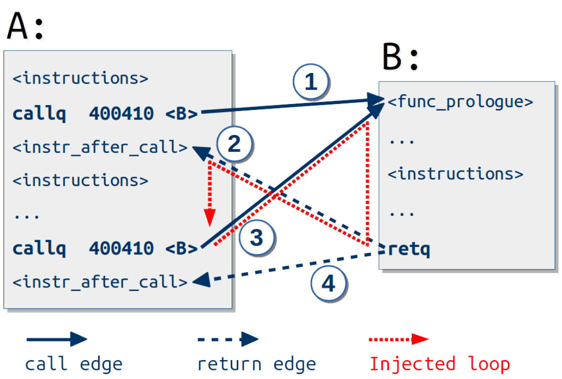


Figure 5: An example of loop injection. Execution follows call edge ③, then returns along edge ②.

point the source buffer to an attacker-controlled location, the target buffer to the address where `memcpy()`'s return address will be found, and set the length to the word size. Then, when `memcpy()` is invoked, `memcpy()` will overwrite its own return address and then return to some other location in the code chosen by the attacker. If this other location is in the valid CFG (i.e., it is an instruction following some call to `memcpy()`), then it is an allowed edge and CFI will allow the return. Thus, `memcpy()` is a simple example of a dispatcher function.

We found many dispatcher functions in `libc`, e.g.,

1. `memcpy()` — As described above.
2. `printf()` — Using the "%n" format specifier, the attacker can write an arbitrary value to an arbitrary location and thus cause `printf()` to overwrite its own return address.
3. `strcat()` — Similar to `memcpy()`. Only works if the address to return to does not contain null bytes.
4. `fputs()` — We rely on the fact that when `fputs()` is called, characters are first temporarily buffered to a location as specified in the `FILE` argument. An attacker can therefore specify a `FILE` where the temporary buffer is placed on top of the return address. Most functions that take a `FILE` struct as an argument can be used in a similar manner.

Similar functions also exist in Windows libraries. Application-specific dispatcher functions can be useful as well, as they may be called more often.

Any function that calls a dispatcher function is itself a dispatcher function: instead of having the callee overwrite its own address, it can be used to overwrite the return address of its caller (or higher on the call chain).

6.1.2 Loop injection

One further potential use of dispatcher functions is that they can be used to create loops in the control-flow graph when none were intended, a process which we call *loop injection*. We can use this to help us achieve Turing-complete computation if we require a loop.

Consider the case where there are two calls to the same dispatcher function, where the attacker controls the arguments to the second call and it is possible to reach the second call from the first through a valid CFG path. For example, it is common for programs to make multiple successive calls to `printf()`. If the second call to `printf()` allows an attacker to control the arguments, then this could cause a potential loop. This is achievable because the second call to `printf()` can return to the instruction following the first call to `printf()`. We can then reach the second call to `printf()` from there (by assumption) and we have completed the loop.

Figure 5 contains an example of this case. Under normal execution, function A would begin by executing the first call to function B on edge 1. Function B returns on edge 2, after which function A continues executing. The second call to function B is then executed, on edge 3. B this time returns on edge 4. Notice that the return instruction in function B has two valid outgoing edges.

An attacker can manipulate this to inject a loop when function B is a dispatcher function. The attacker allows the first call to B to proceed normally on edge 1, returning on edge 2. The attacker sets up memory so that when B is called the second time, the return will follow edge 2 instead of the usual edge 4. That is, even though the code was originally intended as straight-line execution, there exists a back-edge that will be allowed by any static, stateless CFI policy without a shadow stack. A shadow stack would block the transfer along edge 2.

6.2 Turing-complete computation

CFI ensures that the execution flow of a program stays within a predefined CFG. CFI implicitly assumes that the attacker *must* divert from this CFG for successful exploitation. We demonstrate that an attacker can achieve Turing-complete computation while following the CFG. This is not directly one of the attacker goals outlined in Section 3, however it is often a useful step in achieving attacks [14].

Specifically, we show that a single call to `printf()` allows an attacker to perform Turing-complete computation, even when protected with a shadow stack. We dub this printf-oriented programming. In our evaluation, we found it was possible to mount this kind of attack against all but one binary (which rewrote their own limited version of printf).

Our attack technique is not specific to `printf()`: we have constructed a similar attack using `fputs()` which is widely applicable but requires a loop obtained in the control-flow graph (via loop injection or otherwise) to be

Turing-complete. See Appendix C.

6.2.1 Printf-oriented programming

When we control the arguments to `printf()`, it is possible to obtain Turing-complete computation. We show this formally in Appendix B by giving calls to `printf()` which create logic gates. In this section, we give the intuition behind our attacks by showing how an attacker can conditionally write a value at a given location.

Assume address C contains a condition value, which is an integer that is promised to be either zero or one. If the value is one, then we wish to store the constant X at target address T. That is, we wish to perform the computation `*T = *C ? X : *T`. We show how this can be achieved using one call to `printf()`.

To do this, the attacker supplies the specially-crafted format string "`%s%hhnQ%*d%n`" and passes arguments $(C, S, X-2, 0, T)$, defined as follows:

1. C — the address of the condition. While the "`%s`" format specifier expects a string, we pass a pointer to the condition value, which is either the integer 0 or the integer 1. Because of the little-endian nature of x86, the integer 1 contains the byte 0x01 in the first (low) byte and 0x00 in the second byte. This means that when we print it as a string, if the condition value is 1 then exactly one byte will be written out whereas if it is 0 then nothing will be be printed.
2. S — the address of the Q in the format string (i.e., the address of the format string, plus 6). The "`%hhn`" specifier will write a single byte of output consisting of the number of characters printed so far, and will write it on top of the Q in the format string. If we write a 0, the null byte, then the format string will stop executing. If we write a 1, the format string will keep going. It is this action which creates the conditional.
3. $X-2$ — the constant we wish to store, minus two. This specifies the number of bytes to pad in the integer which will be printed. It is the value we wish to save minus two, because two bytes will have already been printed.
4. 0 — an integer to print. We do not care that we are actually printing a 0, only the padding matters.
5. T — the target save location. At this point in time, we have written exactly X bytes to the output, so "`%n`" will write that value at the target address.

Observe that in this example, we have made use of a *self-modifying* format string.

6.2.2 Practical printf-oriented programming

The previous section assumed that the attacker has control of the format string argument, which is usually not the case. We show using simple techniques it is possible to achieve the same results without this control.

We first define the *destination* of a `printf()` call according to its type. The destination of an `sprintf()` call is the address the first argument points to (the destination buffer). The destination of a `fprintf()` call is the address of the temporary buffer in the `FILE` struct. The destination of a plain `printf()` call is the destination buffer of `fprintf()` when called with `stdout`.

Our attack requires three conditions to hold:

- the attacker controls the destination buffer;
- the format string passed to the call to `printf()` already contains a "`%s`" specifier; and,
- the attacker controls the argument to the format specifier as well as a few of the words further down on the stack.

We mount our attack by pointing the destination buffer on top of the stack. We use the "`%s`" plus the controlled argument to overwrite the pointer to the format string (which is stored on the stack), replacing it with a pointer to an attacker-controlled format string. We then skip past any uncontrolled words on the stack with harmless ` ``%x'' ` specifiers. We can then use the remaining controlled words to pivot the `va_list` pointer.

If we do not control any buffer on the stack, we can obtain partial control of the stack by continuing our arbitrary write with the `%s` specifier to add arguments to `printf()`. Note that this does not allow us to use null bytes in arguments, which in 64-bit systems in particular makes exploitation difficult.

6.3 Implications

Our analysis of fully-precise static CFI, the strongest imaginable static CFI policy, shows that preventing attackers with partial control over memory from gaining Turing-complete computation is almost impossible. Run-time libraries and applications contain powerful functions that are part of the valid CFG and can be used by attackers to implement their malicious logic. Attackers can use dispatcher functions to bend control flow within the valid CFG to reach these powerful functions.

Furthermore, we see that if an attacker can find one of these functions and control arguments to it, the attacker will be able to both write to and read from arbitrary addresses at multiple points in time. Defenses which allow attackers to control arguments to these functions must be able to protect against this stronger threat model.

7 Fully-Precise Static CFI Case Studies

We now look at some practical case studies to examine how well fully-precise static CFI can defend against real-

	CFI without shadow stack				CFI with shadow stack			
Binary	Arbitrary write	Info. leakage	Confined code execution	Arbitrary code execution	Arbitrary write	Info. leakage	Confined code execution	Arbitrary code execution
nginx	yes	write	dispatcher	dispatcher	yes	write	no	no
apache	no	write	printf	dispatcher	no	write	write	write
smbclient	yes	printf	printf	printf	yes	printf	printf	printf
wireshark	yes	printf	printf	dispatcher	yes	printf	write	write
xpdf	?	dispatcher	printf	dispatcher	?	write	printf	no
mysql	?	dispatcher	printf	dispatcher	?	write	printf	no

Table 2: The results of our evaluation of the 6 binaries. The 2nd and 6th columns indicate whether the vulnerability we examined allows an attacker to control memory. The other columns indicate which attack goals would be achievable, assuming the attacker controls memory. A “no” indicates that we were not able to achieve that attack goal; anything else indicates it is achievable, and indicates the attack technique we used to achieve the goal. A “?” indicates we were not able to reproduce the exploit.

life exploits on vulnerable programs, both with and without a shadow stack. We split our evaluation into two parts. First, we show that attackers can indeed obtain arbitrary control over memory given actual vulnerabilities. Second, we show that given a program where the attacker controls memory at one point in time, it is possible to mount a control-flow bending attack. Our results are summarized in Table 2.

Our examples are all evaluated on a Debian 5 system running the binaries in x86 64-bit mode. We chose 64-bit mode because most modern systems are running as 64-bit, and attacks are more difficult on 64-bit due to the increased number of registers (data is loaded off of the stack less often).

We do not implement fully-precise static CFI. Instead, for each of our attacks, we manually verify that each indirect control-flow transfer is valid by checking that the edge taken occurs during normal program execution. Because of this, we do not need to handle dynamically linked libraries specially: we manually check those too.

7.1 Control over memory

The threat model we defined earlier allows the attacker to control memory at a single point in time. We argue that this level of control is achievable with most vulnerabilities, by analyzing four different binaries.

7.1.1 Nginx stack buffer overflow

We examined the vulnerability in CVE-2013-2028 [19]: a signedness bug in the chunked decoding component of nginx. We found it is possible to write arbitrary values to arbitrary locations, even when nginx is protected by fully-precise static CFI with a shadow stack, by modifying internal data structures to perform a control-flow bending attack.

The vulnerability occurs when an attacker supplies a large claimed buffer size, overflowing an integer and triggering a stack-based buffer overflow. An attacker can exploit this by redirecting control flow down a path that would never occur during normal execution. The Server Side Includes (SSI) module contains a call to `memcpy()` where all three arguments can be controlled by the attacker. We can arrange memory so after `memcpy()` completes, the process will not crash and will continue accepting requests. This allows us to send multiple requests and set memory to be exactly to the attacker’s choosing.

Under benign usage, this `memcpy()` method is called during the parsing of a SSI file. The stack overflow allows us to control the stack and overwrite the pointer to the request state (which is passed on the stack) to point to a forged request structure, constructed to contain a partially-completed SSI structure. This lets us re-direct control flow to this `memcpy()` call. We are able to control its source and length arguments easily because they point to data on the heap which we control. The destination buffer is not typically under our control: it is obtained by the result of a call to nginx’s memory allocator. However, we can cause the allocator to return a pointer to an arbitrary location by controlling the internal data structures of the memory allocator.

7.1.2 Apache off by one error

We examined an off-by-one vulnerability in Apache’s handling of URL parameters [11]. We found that it is no longer exploitable in practice, when Apache is protected with CFI.

The specific error overwrites a single extra word on the stack; however, this word is not under the attacker’s control. Instead, the word is a pointer to a string on the heap, and the string on the heap is under the attacker’s control. This is a very contrived exploit, and it was not exploitable on the majority of systems in the first place due to the word on the stack not containing any meaningful data. However, on some systems the overwritten word contained a pointer to a data structure which contains function pointers. Later, one of these function pointers would be invoked, allowing for a ROP attack.

When Apache is protected with CFI, the attacker is not able to meaningfully modify the function pointers, and therefore cannot actually gain anything. CFI is effective in this instance because the attacker never obtains control of the machine in the first place.

7.1.3 Smbclient printf vulnerability

We examined a format string vulnerability in smbclient [26]. Since we already fully control the format string of a `printf()` statement, we can trivially control all of memory with printf-oriented programming.

7.1.4 Wireshark stack buffer overflow

A vulnerability in Wireshark's parsing of mpeg files allows an attacker to supply a large packet and overflow a stack buffer. We identify a method of creating a repeatable arbitrary write given this vulnerability even in the presence of a shadow stack.

The vulnerability occurs in the `packet_list_dissect_and_cache_record` function where a fixed-size buffer is created on the stack. An attacker can use an integer overflow to create a buffer of an arbitrary size larger than the allocated space. This allows for a stack buffer overflow.

We achieve an arbitrary write even in the presence of a shadow stack by identifying an arbitrary write in the `packet_list_change_record` function. Normally, this would not be good enough, as this only writes a single memory location. However, an attacker can loop this write due to the fact that the GTK library method `gtk_tree_view_column_cell_set_cell_data`, which is on the call stack, already contains a loop that iterates an attacker-controllable number of times. These two taken together give full control over memory.

7.1.5 Xpdf & Mysql

For two of our six case studies, we were unable to reproduce the public exploit, and as such could not test if memory writes are possible from the vulnerability.

7.2 Exploitation assuming memory control

We now demonstrate that an attacker who can control memory at one point in time can achieve all three goals listed in Section 3, including the ability to issue attacker-desired system calls. (Our assumption is well-founded: in the prior section we showed this is possible.) Prior work has already shown that if arbitrary writes are possible (e.g., through a vulnerability) then data-only attacks are realistic [8]. We show that control-flow bending attacks that are not data-only attacks are also possible.

7.2.1 Evaluation of nginx

Assuming the attacker can perform arbitrary writes, we show that the attacker can read arbitrary files off of the server and relay them to the client, read arbitrary memory out of the server, and execute an arbitrary program with arbitrary arguments. The first two attack goals can be achieved even with a shadow stack; our third attack only works if there is no shadow stack. Nginx is the only binary which is not exploitable by printf-oriented programming, because nginx rewrote their own version of `printf()` and removed "`%n`".

An attacker can read any file that nginx has access to and cause their contents to be written to the output socket, using a purely non-control-data attack. For brevity, we do not describe this attack in detail: prior work has described that these types of exploits are possible.

Our second attack can be thought of as a more controlled version of the recent Heartbleed vulnerability [21], allowing the attacker to read from an *arbitrary* address and dump it to the attacker. The response handling in nginx has two main phases. First, it handles the header of the request and in the process initializes many structs. Then, it parses and handles the body of the request, using these structs. Since the vulnerability in nginx occurs during the parsing of the request body, we use our control over memory to create a forged struct that was not actually created during the initialization phase. In particular, we initialize the `postpone_filter` module data structure (which is not used under normal execution) with an internally-inconsistent state. This causes the module to read data from an arbitrary address of an arbitrary length and copy it to the response body.

Our final attack allows us to invoke `execve()` with arbitrary arguments, if fully-precise static CFI is used without a shadow stack. We use `memcpy()` as a dispatcher function to return into `ngx_sprintf()` and then again into `ngx_exec_new_binary()`, which later on calls `execve()`. By controlling its arguments, the attacker gets arbitrary code execution.

In contrast, when there is a shadow stack, we believe it is impossible for an attacker to trigger invocation of `execve()` due to privilege separation provided by fully-precise static CFI. The master process spawns children via `execve()`, but it is only ever called there — there is no code path that leads to `execve()` from any code point that is reachable within a child process. Thus, in this case CFI effectively provides a form of privilege separation for free, if used with a shadow stack.

7.2.2 Evaluation of apache

On Apache the attacker can invoke `execve()` with arbitrary arguments. Other attacks similar to those on nginx are possible; we omit them for brevity. When there is no shadow stack, we can run arbitrary code by using `strcat()` as a dispatcher gadget to return to a function which later invokes `execve()` under compilations which link the Windows `main` method. When there is a shadow stack, we found a loop that checks, for each module, if the module needs to be executed for the current request. By modifying the conditions on this loop we can cause `mod_cgi` to execute an arbitrary shell command under any compilation. Observe that this attack involves overwriting a function pointer, although to a valid target.

7.2.3 Evaluation of smbclient

Smbclient contains an interpreter that accepts commands from the user and sends them to a Samba fileserver. An attacker who controls memory can drive the interpreter to send any action she desired to the fileserver. This allows an attacker to perform any action on the Samba filesystem that the user could. This program is a demonstration that on some programs, CFI provides essentially no value due to the expressiveness of the original application.

This is one of the most difficult cases for CFI. The only value CFI adds to a binary is restricting it to its CFG: however, when the CFG is easy to traverse and gives powerful functions, CFI adds no more value than a system call filter.

7.2.4 Evaluation of wireshark

An attacker who controls memory can write to any file that the current user has access to. This gives power equivalent to arbitrary code execution by, for example, overwriting the `authorized_keys` file. This is possible because wireshark can save traces, and an attacker who controls memory can trivially overwrite the filename being written to with one the attacker picks.

If the attacker waits for the user to click save and simply overwrites the file argument, this would be a data-only attack under our definitions. It is also possible to use control-flow bending to invoke `file_save_as_cb()` directly, by returning into the GTK library and overwriting a code pointer with the file save method, which is within the CFG.

7.2.5 Evaluation of xpdf

Similar to wireshark, an attacker can use xpdf to write to arbitrary files using `memcpy()` as a dispatcher gadget when there is no shadow stack. When a shadow stack is present, we are limited to a printf-oriented programming attack and we can only write files with specific extensions, which does not obviously give us ability to run arbitrary code.

7.2.6 Evaluation of mysql

When no shadow stack is present, attacks are trivial. A dispatcher gadget lets us return into `do_system()`, `do_exec()`, or `do_perl()` from within the mysql client. (For this attack we assume a vulnerable client to connects to a malicious server controlled by the attacker.) When a shadow stack is present the attacker is more limited, but we still can use printf-oriented programming to obtain arbitrary computation on memory. We could not obtain arbitrary execution with a shadow stack.

7.3 Combining attacks

As these six case studies indicate, control-flow bending is a realistic attack technique. In the five cases where CFI does not immediately stop the exploit from occurring, as it does for Apache, an attacker can use the vulnerability to achieve arbitrary writes in memory. From here, it is possible to mount traditional data-only attacks (e.g., by modifying configuration data-structures). We showed that using control-flow bending techniques, more powerful attacks are possible. We believe this attack technique is general and can be applied to other applications and vulnerabilities.

8 Related work

Control-flow integrity. Control-flow integrity was originally proposed by Abadi et al. [1, 15] a decade ago. Classical CFI instruments indirect branch target locations with equivalence-class numbers (encoded as a label in a side-effect free instruction) that are checked at branch locations before taking the branch. Many other CFI schemes have been proposed since then.

The most coarse-grained policies (e.g., Native Client [40] or PittSFIeld [20]) align valid targets to the beginning of chunks. At branches, these CFI schemes ensure that control-flow is not transferred to unaligned addresses. Fine-grained approaches use static analysis of source code to construct more accurate CFGs (e.g., WIT [2] and HyperSafe [39]). Recent work by Niu et al. [27] added support for separate compilation and dynamic loading. Binary-only CFI implementations are generally more coarse-grained: MoCFI [13] and BinCFI [44] use static binary rewriting to instrument indirect branches with additional CFI checks.

CFI evaluation metrics. Others have attempted to create methods to evaluate practical CFI implementations. The Average Indirect target Reduction (AIR) [44] metric was proposed to measure how much on average the set of indirect valid targets is reduced for a program under CFI. We argue that this metric has limited utility, as even high AIR values of 99% are insecure, allowing an attacker to perform arbitrary computation and issue arbitrary system calls. The gadget reduction metric is another way to evaluate CFI effectiveness [27], by measuring how much the set of reachable gadgets is reduced overall. Gadget finder tools like ROPgadget [34] or ropper [33] can be used to estimate this metric.

CFI security evaluations. There has recently been a significant effort to analyze the security of specific CFI schemes, both static and dynamic. Göktaş et al. [16] analyzed the security of static coarse-grained CFI schemes and found that the specific policy of requiring returns to target call-preceded locations is insufficient. Following this work, prevent-the-exploit-style coarse-grained CFI schemes with dynamic components that rely on runtime heuristics were defeated [5, 14]. The attacks relied upon the fact that the attacks could hide themselves from the dynamic heuristics, and then reduced down to attacks on coarse-grained CFI. Our evaluation of minimal programs builds on these results by showing that coarse-grained CFI schemes which have an AIR value of 99% are still vulnerable to attacks on trivially small programs.

Non-control data attacks. Attacks that target only sensitive data structures were categorized as *pure data attacks* by Pincus and Baker [32]. Typically, these attacks would overwrite application-specific sensitive variables (such as the “is authenticated” boolean which exists within many applications). This was expanded by Chen et al. [8] who demonstrated that *non-control data attacks* are practical attacks on real programs. Our work generalizes these attacks to allow modifications of control-flow data, but only in a way that follows the CFI policy.

Data-flow integrity. Nearly as old of an idea as CFI, Data-Flow Integrity (DFI) provides guarantees for the integrity of the data within a program [6]. Although the original scheme used static analysis to compute an approximate data-flow graph — what we would now call a coarse-grained approach — more refined DFI may be able to protect against our attacks. We believe security evaluation of prevent-the-corruption style defenses such as DFI is an important future direction of research.

Type- and memory-safety. Other defenses have tried to bring type-safety and memory-safety to unsafe languages like C and C++. SoftBound [22] is a compile-time defense which enforces spatial safety in C, but at a 67% performance overhead. CETS [23] extends this work with a compile-time defense that enforces temporal safety in C, by protecting against memory management errors. CCured [24] adds type-safe guarantees to C by attempting to statically determine when errors cannot occur, and dynamically adding checks when nothing can be proven statically. Cyclone [17] takes a more radical approach and re-designs C to be type- and memory-safe. Code-Pointer Integrity (CPI) [18] reduces the overhead of SoftBound by only protecting code pointers. While CPI protects the integrity of all indirect control-flow transfers, limited control-flow bending attacks using conditional jumps may be possible by using non-control-data attacks. Evaluating control-flow bending attacks on CPI would be an interesting direction for future work.

9 Conclusion

Control-flow integrity has historically been considered a strong defense against control-flow hijacking attacks and ROP attacks, if implemented to its fullest extent. Our results indicate that this is not entirely the case, and that control-flow bending allows attackers to perform meaningful attacks even against systems protected by fully-precise static CFI. When no shadow stack is in place, dispatcher functions allow powerful attacks. Consequently, CFI without return instruction integrity is not secure. However, CFI with a shadow stack does still provide value as a defense, if implemented correctly. It can significantly raise the bar for writing exploits by forcing attackers to tailor their attacks to a particular application; it limits an attacker to issue only system calls available to the application; and it can make specific vulnerabilities unexploitable under some circumstances.

Our work has several implications for design and deployment of CFI schemes. First, shadow stacks appear to be essential for the security of CFI. We also call for adversarial analysis of new CFI schemes before they are deployed, as our work indicates that many published CFI schemes have significant security weaknesses. Finally, to make control-flow bending attacks harder, deployed systems that use CFI should consider combining CFI with other defenses, such as data integrity protection to ensure that data passed to powerful functions cannot be corrupted in the presence of a memory safety violation.

More broadly, our work raises the question: just how much security can prevent-the-exploit defenses (which allow the vulnerability to be triggered and then try to prevent exploitation) provide? In the case of CFI, we argue the answer to this question is that it still provides some, but not complete, security. Evaluating other prevent-the-exploit schemes is an important area of future research.

We hope that the analyses in this paper help establish a basis for better CFI security evaluations and defenses.

10 Acknowledgments

We would like to thank Jay Patel and Michael Theodorides for assisting us with three of the case studies. We would also like to thank Scott A. Carr, Per Larsen, and the anonymous reviewers for countless discussions, feedback, and suggestions on improving the paper. This work was supported by NSF grant CNS-1513783, by the AFOSR under MURI award FA9550-12-1-0040, and by Intel through the ISTC for Secure Computing.

References

[1] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-flow integrity. In *CCS'05* (2005).

[2] Akritidis, P., Cadar, C., Raiciu, C., Costa, M., and Castro, M. Preventing memory error exploits with WIT. In *IEEE S&P'08* (2008).

[3] Bletsch, T., Jiang, X., and Freeh, V. Mitigating code-reuse attacks with control-flow locking. In *ACSAC'11* (2011).

[4] Bletsch, T., Jiang, X., Freeh, V. W., and Liang, Z. Jump-oriented programming: a new class of code-reuse attack. In *ASIACCS'11* (2011).

[5] Carlini, N., and Wagner, D. ROP is still dangerous: Breaking modern defenses. In *USENIX Security'14* (2014).

[6] Castro, M., Costa, M., and Harris, T. Securing software by enforcing data-flow integrity. In *OSDI '06* (2006).

[7] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M. Return-oriented programming without returns. In *CCS'10* (2010), pp. 559–572.

[8] Chen, S., Xu, J., Sezer, E. C., Gauriar, P., and Iyer, R. K. Non-control-data attacks are realistic threats. In *USENIX Security'05* (2005).

[9] Cheng, Y., Zhou, Z., Yu, M., Ding, X., and Deng, R. H. ROPecker: A generic and practical approach for defending against ROP attacks. In *NDSS'14* (2014).

[10] Cowan, C., Pu, C., Maier, D., Hintony, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., and Zhang, Q. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security'98* (1998).

[11] Cox, M. CVE-2006-3747: Apache web server off-by-one buffer overflow vulnerability. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3747, 2006.

[12] Criswell, J., Dautenhahn, N., and Adve, V. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *IEEE S&P'14* (2014).

[13] Davi, L., Dmitrienko, R., Egele, M., Fischer, T., Holz, T., Hund, R., Nuernberger, S., and Sadeghi, A. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *NDSS'12* (2012).

[14] Davi, L., Sadeghi, A.-R., Lehmann, D., and Monrose, F. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security'14* (2014).

[15] Erlingsson, Ú., Abadi, M., Vrable, M., Budiu, M., and Necula, G. C. XFI: Software guards for system address spaces. In *OSDI'06* (2006).

[16] Goktas, E., Athanasopoulos, E., Bos, H., and Portokalidis, G. Out of control: Overcoming control-flow integrity. In *IEEE S&P'14* (2014).

[17] Jim, T., Morrisett, J. G., Grossman, D., Hicks, M. W., Cheney, J., and Wang, Y. Cyclone: A safe dialect of C. In *ATC'02* (2002).

[18] Kuznetsov, V., Payer, M., Szekeres, L., Candea, G., Sekar, R., and Song, D. Code-pointer integrity. In *OSDI'14* (2014).

[19] MacManus, G. CVE-2013-2028: Nginx http server chunked encoding buffer overflow. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2028, 2013.

[20] McCamant, S., and Morrisett, G. Evaluating SFI for a CISC architecture. In *USENIX Security'06* (2006).

[21] Mehta, N., Riku, Antti, and Matti. The Heartbleed bug. http://heartbleed.com/, 2014.

[22] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. SoftBound: Highly compatible and complete spatial memory safety for C. In *PLDI'09* (2009).

[23] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. CETS: Compiler enforced temporal safety for C. In *ISMM'10* (2010).

[24] Necula, G., Condit, J., Harren, M., McPeak, S., and Weimer, W. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS) 27*, 3 (2005), 477–526.

[25] Nergal. The advanced return-into-lib(c) exploits. *Phrack 11*, 58 (Nov. 2007), http://phrack.com/issues.html?issue=67&id=8.

[26] Nissl, R. CVE-2009-1886: Formatstring vulnerability in smbclient. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1886, 2009.

[27] Niu, B., and Tan, G. Modular control-flow integrity. In *PLDI'14* (2014).

[28] Pappas, V., Polychronakis, M., and Keromytis, A. D. Transparent ROP exploit mitigation using indirect branch tracing. In *USENIX Security* (2013), pp. 447–462.

[29] PaX-Team. PaX ASLR (Address Space Layout Randomization). http://pax.grsecurity.net/docs/aslr.txt, 2003.

[30] Payer, M., Barresi, A., and Gross, T. R. Fine-grained control-flow integrity through binary hardening. In *DIMVA'15*.

[31] Philippaerts, P., Younan, Y., Muylle, S., Piessens, F., Lachmund, S., and Walter, T. Code pointer masking: Hardening applications against code injection attacks. In *DIMVA'11* (2011).

[32] Pincus, J., and Baker, B. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy 2* (2004), 20–27.

[33] ropper. Ropper – rop gadget finder and binary information tool. https://scoding.de/ropper/, 2014.

[34] Salwan, J. ROPgadget – Gadgets finder and auto-roper. http://shell-storm.org/project/ROPgadget/, 2011.

[35] Schwartz, E. J., Avgerinos, T., and Brumley, D. Q: Exploit hardening made easy. In *USENIX Security'11* (2011).

[36] Shacham, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS'07*.

[37] Szekeres, L., Payer, M., Wei, T., and Song, D. SoK: Eternal war in memory. In *IEEE S&P'13* (2013).

[38] van de Ven, A., and Molnar, I. Exec shield. https://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf, 2004.

[39] Wang, Z., and Jiang, X. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *IEEE S&P'10* (2010).

[40] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE S&P'09* (2009).

[41] ZENG, B., TAN, G., AND ERLINGSSON, U. Strato: A retargetable framework for low-level inlined-reference monitors. In *USENIX Security'13* (2013).

[42] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., MCCAMANT, S., AND SZEKERES, L. Protecting function pointers in binary. In *ASIACCS'13* (2013).

[43] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical control flow integrity and randomization for binary executables. In *IEEE S&P'13* (2013).

[44] ZHANG, M., AND SEKAR, R. Control flow integrity for COTS binaries. In *USENIX Security'13* (2013).

A Minimal vulnerable program for indirect jump or call hijacking

The program in Figure 6 contains a bug that allows the attacker to reliably hijack an indirect jump or indirect call target. The function `overflow()` allows an attacker to overflow a struct allocated on the stack that contains two pointers used as the targets for an indirect jump or an indirect call, respectively. The attacker can use the indirect jump or call to divert control flow to a return gadget and continue with a classic ROP attack. Alternatively, an attacker may rely on JOP or COP techniques. We also examined variations of this minimal vulnerable program, e.g., putting the struct somewhere on the heap or requiring the attacker to first perform a stack pivot to ensure that the stack pointer points to attacker-controlled data.

```
#include <stdio.h>
#include <string.h>
#define STDIN 0

void jmptarget();
void calltarget();

struct data {
 char buf[1024];
 int arg1;
 int arg2;
 int arg3;
 void (*jmpPtr)();
 void (*callPtr)(int,int,int);
};

void overflow() {
 struct data our_data;
 our_data.jmpPtr = &&label;
 our_data.callPtr = &calltarget;
 printf("%x\n", (unsigned int)&our_data.buf);
 printf("\ndata>␣");
 read(STDIN, our_data.buf, 1044);
 printf("\n");
 asm("push␣%0;push␣%1;push␣%2;call␣*%3;add␣$12,%%esp;"
 : : "r"(our_data.arg3),
     "r"(our_data.arg2),
     "r"(our_data.arg1),
     "r"(our_data.callPtr));
 asm("jmp␣*%0" : : "r"(our_data.jmpPtr));
 printf("?\n");
label:
 printf("label␣reached\n");
}

void jmptarget() {
 printf("jmptarget()␣called\n");
}

void calltarget(int arg1, int arg2, int arg3) {
 printf("calltarget()␣called␣(args:␣%x,␣%x,␣%x)\n",
 arg1, arg2, arg3);
}

int main(int argc, char* argv[]) {
 setbuf(stdout, NULL);
 overflow();
 printf("\ndone.\n");
 return 0;
}
```

Figure 6: A minimal vulnerable program that allows hijack of an indirect jump or indirect call target.

B Printf is Turing-complete

The semantics of `printf()` allow for Turing-complete computation while following the minimal CFG.

At a high level, we achieve Turing-completeness by creating logic gates out of calls to `printf()`. We show how to expand a byte to its eight bits, and how to compact the eight bits back to a byte. We will compute on values by using them in their base-1 (unary) form and we will use string concatenation as our primary method of arithmetic. That is, we represent a true value as the byte sequence 0x01 0x00, and the false value by the byte sequence 0x00 0x00, so that when treated as strings their lengths are 1 and 0 respectively.

Figure 7 contains an implementation of an OR gate using only calls to `printf()`. In the first call to `printf()`, if either of the two inputs is non-zero, the output length will be non-zero, so the output will be set to a non-zero value. The second call to `printf()` normalizes the value so if it was any non-zero value it becomes a one. Figure 7 implements a NOT gate using the fact that adding 255 is the same as subtracting one, modulo 256.

In order to operate on bytes instead of bits in our contrived format, we implement a test gate which can test if a byte is equal to a specific value. By repeating this test gate for each of the 256 potential values, we can convert a 8-bit value to its "one-hot encoding" (a 256-bit value with a single bit set, corresponding to the original value). Splitting a byte into bits does not use a pointer to a byte, but a byte itself. This requires that the byte is on the stack. Moving it there takes some effort, but can still be done with `printf()`. The easiest way to achieve this would be to interweave calls to `memcpy()` and `printf()`, copying the bytes to the stack with `memcpy()` and then operating on them with

```
void or(int* in1, int* in2, int* out) {
  printf("%s%s%n", in1, in2, out);
  printf("%s%n", out, out);
}

void not(int* in, int* out) {
  printf("%*d%s%n", 255, in, out);
  printf("%s%n", out, out);
}

void test(int in, int const, int* out) {
  printf("%*d%*d%n", in, 0, 256-const, 0, out);
  printf("%s%n", out, out);
  printf("%*d%s%n", 255, out, out);
  printf("%s%n", out, out);
}

char* pad = memalign(257, 256);
memset(pad, 1, 256);
pad[256] = 0;
void single_not(int* in, int* out) {
  printf("%*d%s%n%hhn%s%s%n", 255, in, out,
         addr_of_argument, pad, out, out);
}
```

Figure 7: Gadgets for logic gates using printf.

printf(). However, this requires more of the program CFG, so we instead developed a technique to achieve the same goal without resorting to memcpy(). When printf() is invoked, the characters are not sent directly to the stdout stream. Instead, printf() will use the FILE struct corresponding to the stdout stream to buffer the data temporarily. Since the struct is stored in a writable memory location, the attacker can invoke printf() with the “%n” format specifier to point the buffer onto the stack. Then, by reading values out of memory with “%s” the attacker can move these values onto the stack. Finally, the buffer can be moved back to its original location.

It is possible to condense multiple calls to printf() to only one. Simply concatenating the format strings is not enough, because the length of the strings is important with the “%n” modifier. That is, after executing a NOT gate, the string length will either be 255 or 256. We cannot simply insert another NOT gate, as that would make the length be one of 510, 511, or 512. We fix this by inserting a *length-repairing sequence* of “%hhn%s”, which pads the length of the string to zero modulo 256. We use it to create a NOT gate in a single call to printf() in Figure 7. Using this technique, we can condense an arbitrary number of gates into a single call to printf(). This allows bounded Turing-complete computation.

To achieve full Turing-complete computation, we need a way to loop a format string. This is possible by overwriting the pointer inside printf() that tracks which character in the format string is currently being executed. The attacker is unlucky in that at the time the “%n” format specifier is used, this value is saved in a register on our 64-bit system. However, we identify one point in time in which the attacker can always mount the attack. The printf() function makes calls to puts() for the static components of the string. When this function call is made, all registers are saved to the stack. It turns out that an attacker can overwrite this pointer from within the puts() function. By doing this, the format string can be looped.

An attacker can cause puts() to overwrite the desired pointer. Prior to printf() calling puts(), the attacker uses “%n” format specifiers to overwrite the stdout FILE object so that the temporary buffer is placed directly on top of the stack where the index pointer will be saved. Then, we print the eight bytes corresponding to the new value we want the pointer to have. Finally, we use more “%n” format specifiers to move the buffer back to some other location so that more unintended data will not be overwritten.

C Fputs-oriented programming

These printf-style attacks are not unique to printf(): many other functions can be exploited in a similar manner. We give one further attack using fputs(). For brevity, we show how an attacker can achieve a conditional write, however other computation is possible.

The FILE struct contains three char* fields to temporarily buffer character data before it is written out: a base pointer, a current pointer, and an end pointer. fputs() works by storing bytes sequentially starting from the base pointer keeping track with the current pointer. When it exceeds the end pointer, the data is written out, and the current pointer is set back to the base. Programmatically, the way this works is that if the current pointer is larger than the end pointer, fputs() flushes the buffer and then sets the current pointer to the base pointer and continues writing.

This can be used to conditionally copy from source address S to target address T if the byte address C is nonzero. Using fputs(), the attacker copies the byte at C on top of each of the 8 bytes in the end pointer. Then, the attacker sets the current pointer to T and then calls fputs() with this FILE and argument S. If the byte at C is zero, the end pointer is the NULL pointer, and no data is written. Otherwise, the data is written.

This attack requires two calls to fputs(). We initialize memory with the constant pointers that are desired. The first call to fputs() moves the C byte over the end pointer. The second call is the conditional move. The two calls can be obtained by loop injection, or by identifying an actual loop in the CFG.

Automatic Generation of Data-Oriented Exploits

Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, Zhenkai Liang
Department of Computer Science, National University of Singapore
{huhong, chuazl, sendroiu, prateeks, liangzk}@comp.nus.edu.sg

Abstract

As defense solutions against control-flow hijacking attacks gain wide deployment, control-oriented exploits from memory errors become difficult. As an alternative, attacks targeting non-control data do not require diverting the application's control flow during an attack. Although it is known that such data-oriented attacks can mount significant damage, no systematic methods to automatically construct them from memory errors have been developed. In this work, we develop a new technique called *data-flow stitching*, which systematically finds ways to join data flows in the program to generate data-oriented exploits. We build a prototype embodying our technique in a tool called FLOWSTITCH that works directly on Windows and Linux binaries. In our experiments, we find that FLOWSTITCH automatically constructs 16 previously unknown and three known data-oriented attacks from eight real-world vulnerable programs. All the automatically-crafted exploits respect fine-grained CFI and DEP constraints, and 10 out of the 19 exploits work with standard ASLR defenses enabled. The constructed exploits can cause significant damage, such as disclosure of sensitive information (e.g., passwords and encryption keys) and escalation of privilege.

1 Introduction

In a memory error exploit, attackers often seek to execute arbitrary malicious code, which gives them the ultimate freedom in perpetrating damage with the victim program's privileges. Such attacks typically hijack the program's control flow by exploiting memory errors. However, such control-oriented attacks, including code-injection and code-reuse attacks, can be thwarted by efficient defense mechanisms such as control-flow integrity (CFI) [10, 43, 44], data execution prevention (DEP) [12], and address space layout randomization (ASLR) [15, 33]. Recently, these defenses have become practical and are gaining universal adoption in commodity operating systems and compilers [8, 36], making control-oriented attacks increasingly difficult.

However, control-oriented attacks are not the only malicious consequence of memory error exploits. Memory errors also enable attacks through corrupting non-control data — a well-known result from Chen *et al.* [19]. We refer to the general class of non-control data attacks as *data-oriented attacks*, which allow attackers to tamper with the program's data or cause the program to disclose secret data inadvertently. Several recent high-profile vulnerabilities have highlighted the menace of these attacks. In a recent exploit on Internet Explorer (IE) 10, it has been shown that changing a single byte — specifically the *Safemode* flag — is sufficient to run arbitrary code in the IE process [6]. The Heartbleed vulnerability is another example wherein sensitive data in an SSL-enabled server could be leaked without hijacking the control-flow of the application [7].

If data-oriented attacks can be constructed such that the exploited program follows a legitimate control flow path, they offer a realistic attack mechanism to cause damage even in the presence of state-of-the-art control-flow defenses, such as DEP, CFI and ASLR. However, although data-oriented attacks are conceptually understood, most of the known attacks are straightforward corruption of non-control data. No systematic methods to identify and construct these exploits from memory errors have been developed yet to demonstrate the power of data-oriented attacks. In this work, we study systematic techniques for automatically constructing data-oriented exploits from given memory corruption flaws.

Based on a new concept called *data-flow stitching*, we develop a novel solution that enables us to systematize the understanding and construction of data-oriented attacks. The intuition behind this approach is that non-control data is often far more abundant than control data in a program's memory space; as a result, there exists an opportunity to reuse existing data-flow patterns in the program to do the attacker's bidding. The main idea of data-flow stitching is to "stitch" existing data-flow paths in the program to form new (unintended) data-flow paths via exploiting memory errors. Data-flow stitching can thus connect two or more data-flow paths that are disjoint in the benign execution of the program. Such a stitched execution, for instance, allows the attacker to write out a secret value (e.g., cryptographic keys) to the program's public output, which otherwise would only be used in private operations of the application.

Problem. Our goal is to check whether a program is exploitable via data-oriented attacks, and if so, to automatically generate working data-oriented exploits. We aim to develop an exploit generation toolkit that can be used in conjunction with a dynamic bug-finding tool. Specifically, from an input that triggers a memory corruption bug in the program, with the knowledge of the program, our toolkit constructs a data-oriented exploit.

Compared to control-oriented attacks, data-oriented attacks are more difficult to carry out, since attackers cannot run malicious code of their choice even after the attack. Though non-control data is abundant in a typical program's memory space, due to the large range of possibilities for memory corruption and their subtle influence on program memory states, identifying how to corrupt memory values for a successful exploit is difficult. The main challenge lies in searching through the large space of memory state configurations, such that the attack exhibits an unintended data consequence, such as information disclosure or privilege escalation. An additional practical challenge is that defenses such as ASLR randomize addresses, making it even harder since absolute address values cannot be used in exploit payloads.

Our Approach. In this work, we develop a novel solution to construct data-oriented exploits through data-flow stitching. Our approach consists of a variety of techniques that stitch data flows in a much more efficient manner compared to manual analysis or brute-force searching. We develop ways to prioritize the searching for data-flow stitches that require a single new edge or a small number of new edges in the new data-flow path. We also develop techniques to address the challenges caused by limited knowledge of memory layout. To further prune the search space, we model the path constraints along the new data-flow path using symbolic execution, and check its feasibility using SMT solvers. This can efficiently prune out memory corruptions that cause the attacker to lose control over the application's execution, like triggering exceptions, failing on compiler-inserted runtime checks, or causing the program to abort abruptly. By addressing these challenges, a data-oriented attack that causes unintended behavior can be constructed, without violating control-flow requirements in the victim program.

We build a tool called FLOWSTITCH embodying these techniques, which operates directly on x86 binaries. FLOWSTITCH takes as input a vulnerable program with a memory error, an input that exploits the memory error, as well as benign inputs to that program. It employs dynamic binary analysis to construct an information-flow graph, and efficiently searches for data flows to be stitched. FLOWSTITCH outputs a working data-oriented exploit that either leaks or tampers with sensitive data.

Results. We show that automatic data-oriented exploit generation is feasible. In our evaluation, we find that multiple data-flow exploits can often be constructed from a single vulnerability. We test FLOWSTITCH on eight real-world vulnerable applications, and FLOWSTITCH automatically constructs 19 data-oriented exploits from eight applications, 16 of which are previously unknown to be feasible from known memory errors. All constructed exploits violate memory safety, but completely respect fine-grained CFI constraints. That is, they create no new edges in the static control-flow graph. All the attacks work with the DEP protection turned on, and 10 exploits (out 19) work even when ASLR is enabled. The majority of known data-oriented attacks (c.f. Chen *et. al.* [19], Heartbleed [7], IE-Safemode [6]) are straightforward non-control data corruption attacks, requiring at most one data-flow edge. In contrast, seven exploits we have constructed are only feasible with the addition of multiple data-flow edges in the data-flow graph, showing the efficacy of our automatic construction techniques.

Contributions. This paper makes the following contributions:

- We conceptualize *data-flow stitching* and develop a new approach that systematizes the construction of data-oriented attacks, by composing the benign data flows in an application via a memory error.
- We build a prototype of our approach in an automatic data-oriented attack generation tool called FLOWSTITCH. FLOWSTITCH operates directly on Windows and Linux x86 binaries.
- We show that constructing data-oriented attacks from common memory errors is feasible, and offers a promising way to bypass many defense mechanisms to control-flow attacks. Specifically, we show that 16 previously unknown and three known data-oriented attacks are feasible from eight vulnerabilities. All our 19 constructed attacks bypass DEP and the CFI checks, and 10 of them bypass ASLR.

2 Problem Definition

2.1 Motivating Example

The following example shown in Code 1 is modeled after a web server. It loads the web site's private key from a

```
1  int server() {
2    char *userInput, *reqFile;
3    char *privKey, *result, output[BUFSIZE];
4    char fullPath[BUFSIZE] = "/path/to/root/";
5
6    privKey = loadPrivKey("/path/to/privKey");
7    /* HTTPS connection using privKey */
8    GetConnection(privKey, ...);
9    userInput = read_socket();
10   if (checkInput(userInput)) {
11     /* user input OK, parse request */
12     reqFile = getFileName(userInput);
13     /* stack buffer overflow */
14     strcat(fullPath, reqFile);
15     result = retrieve(fullPath);
16     sprintf(output,"%s:%s",reqFile,result);
17     sendOut(output);
18   }
19 }
```

Code 1: Vulnerable code snippet. String concatenation on line 14 introduces a stack buffer overflow vulnerability.

file, and uses it to establish an HTTPS connection with the client. After receiving the input — a file name, the code sanitizes the input by invoking `checkInput()` (on line 10). The code then retrieves the file content and sends the content and the file name back to the client. There is a stack buffer overflow vulnerability on line 14, through which the client can corrupt the stack memory immediately after the `fullPath` buffer.

However, there is no obvious security-sensitive non-control data [19] on the stack of the vulnerable function. To create a data-oriented attack, we analyze the data flow patterns in the program's execution under a benign input, which contains at least two data flows: the flow involving the sensitive private key pointed to by the pointer named `privKey`, and the flow involving the input file name pointed by the pointer named `reqFile`, which is written out to the program's public outputs. Note that in the benign run, these two data flows do not intersect — that is, they have no shared variables or direct data dependence between them, but we can corrupt memory in such a way that the secret private key gets written out to the public output. Specifically, the attacker crafts an attack exploiting the buffer overflow to corrupt the pointer `reqFile`, making it to point to the private key. This forces the program to copy the private key to the output buffer in the `sprintf` function on line 16, and then the program sends the output buffer to the client on line 17. Note that the attack alters no control data, and executes the same execution path as the benign run.

This example illustrates the idea of *data-flow stitching*, an exploit mechanism to manipulate the benign data flows in a program execution without changing its control flow. Though it is not difficult to manually analyze this simplified example to construct a data-oriented attack, real-world programs are much more complex and often available in binary-only form. Constructing data-oriented attacks for such programs is a challenging task we tackle in this work.

2.2 Objectives & Threat Model

In this paper, we aim to develop techniques to automatically construct data-oriented attacks by stitching data flows. The generated data-oriented attacks result in the following consequences:

G1: Information disclosure. The attacks leak sensitive data to attackers. Specifically, we target the following sources of security-sensitive data:

- **Passwords and private keys**. Leaking passwords and private keys help bypass authentication controls and break secure channels established by encryption techniques.
- **Randomized values**. Several memory protection defenses utilize randomized values generated by the program at runtime, such as stack canaries, CFI-enforcing tags, and randomized addresses. Disclosure of such information allows attackers bypass randomization-based defenses.

G2: Privilege escalation. The attacks grant attackers the access to privileged application resources. Specifically, we focus on the following kinds of program data:

- **System call parameters**. System calls are used for high-privilege operations, like `setuid()`. Corrupting system call parameters can lead to privilege escalation.
- **Configuration settings**. Program configuration data, especially for server programs (e.g., data loaded from `httpd.conf` for Apache servers) specifies critical information, such as the user's permission and the root directory of the web server. Corrupting such data directly escalates privilege.

Threat Model. We assume the execution environment has deployed defense mechanisms against control-flow hijacking attacks, such as fine-grained CFI [10,32], non-executable data [12] and state-of-the-art implementation of ASLR. Therefore attackers cannot mount control flow hijacking attacks. All non-deterministic system generated values, e.g., stack-canaries or CFI tags, are assumed to be secret and unknown to attackers.

2.3 Problem Definition

To systematically construct data-oriented exploits, we introduce a new abstraction called the *two-dimensional data-flow graph (2D-DFG)*, which represents the flows of data in a given program execution in two dimensions: memory addresses and execution time. Specifically, a

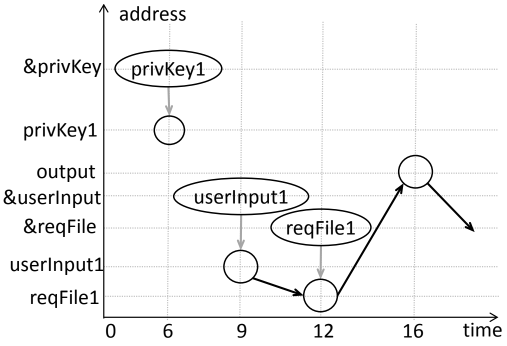


Figure 1: 2D-DFG of a concrete execution of Code 1. Black edges are data edges, while grey edges are address edges. For clarity, vertices do not strictly conform the order on address-axis (this applies to all figures). We use line number to represent the time. $var1$ means a particular value (constant) of the variable `var` in Code 1.

2D-DFG is a directed graph, represented as $G = \{V,E\}$, where V is the set of vertices, and E is the set of edges. A vertex in V is a variable instance, i.e., a point in the two dimensional address-time space, denoted as (a,t), where a is the address of the variable, and t is a representation of the execution time when the variable instance is created. The address includes both memory addresses and register names[1], and the execution time is represented as an instruction counter in the execution trace of the program. An edge (v', v) from vertex v' to vertex v denotes a data dependency created during the execution, i.e., the value of v or the address of v is derived from the value of v'. Therefore, the 2D-DFG also embodies the "points to" relation between pointer variables and pointed variables. Each vertex v has a `value` property, denoted as $v.value$.

A new vertex $v = (a,t)$ is created if an instruction writes to address a at the execution time t. A new data edge (v′, v) is created if an instruction takes v' as the source operand and takes v as a destination operand. A new address edge (v′, v) is created if an instruction takes v′ as the address of one operand v. Therefore, an instruction may create several vertices at a given point in execution if it changes more than one variables, for instance in the loop-prefixed instructions (e.g., `repmov`). Note that the 2D-DFG is a representation of the direct data dependencies created in a program execution under a concrete input, not the static data-flow graph often used in static analysis. Figure 1 shows a 2D-DFG of Code 1.

We define the core problem of *data-flow stitching* as follows. For a program with a memory error, we take the following parameters as the input: a 2D-DFG G from a benign execution of the program, a memory error influence I, and two vertices v_S (source) and v_T (target). In our example, v_S is the private key buffer, shown as (privKey1[2], 6) in Figure 1 and v_T is the public output buffer, shown as (output, 16) in Figure 1. Our goal is to generate an exploit input that exhibits a new 2D-DFG $G' = \{V',E'\}$, where V' and E' result from the memory error exploit, and that G' contains data-flow paths from v_S to v_T. Let $\overline{E} = E' - E$ be the edge-set difference and $\overline{V} = V' - V$ be the vertex-set difference. Then, $\overline{E}$ is the set of new edges we need to generate to get E' from E.

The memory error influence I is the set of memory locations which can be written to by the memory error, represented as a set of vertices. Therefore, we must select $\overline{V}$ to be a subset of vertices in I. To achieve **G1** we consider variables carrying program secrets as source vertices and variables written to public outputs as target vertices. In the development of attacks for **G2**, source vertices are attacker-controlled variables and target vertices are security-critical variables such as system call parameters. A successful data-oriented attack should additionally satisfy the following critical requirements:

- **R1.** The exploit input satisfies the program path constraints to reach the memory error, create new edges and continue the execution to reach the instruction creating v_T.
- **R2.** The instructions executed in the exploit must conform to the program's static control flow graph.

2.4 Key Technique & Challenges

The key idea in data-flow stitching is to efficiently search for the new data-flow edge set $\overline{E}$ to add in G' such that it creates new data-flow paths from v_S to v_T. For each edge $(x,y) \in \overline{E}$, x is data-dependent on v_S and v_T is data-dependent on y. We denote the sub-graph of G containing all the vertices that are data-dependent on v_S as the source flow. We also denote the sub-graph of G containing all the vertices that v_T is data-dependent on as the target flow. For each vertex pair (x, y), where x is in the source flow and y is in the target flow, we check whether (x, y) is a feasible edge of $\overline{E}$ resulting from the inclusion of vertices from I. The vertices x and y may either be contained in I directly, or be connected via a sequence of edges by corruption of their pointers which are in I. If we change the address to which x is written, or change the address from which y is read, the value of x will flow to y. If so, we call (x, y) the *stitch edge*, x the *stitch source*, and y the *stitch target*. For example, in Figure 2, we change the pointer (which is in I) of the file name from $reqFile1$ to $privKey1$. Then the flow of the private key and the flow of the file name are stitched, as we discuss in Section 2.1. In finding data-flow stitching in the 2D-DFG, we face the following challenges:

- **C1. Large search space for stitching.** A 2D-DFG from a real-world program has many data flows and a large number of vertices. For example, there are

[1] We treat the register name as a special memory address.

[2] privKey1 here means the key buffer address, a concrete value.

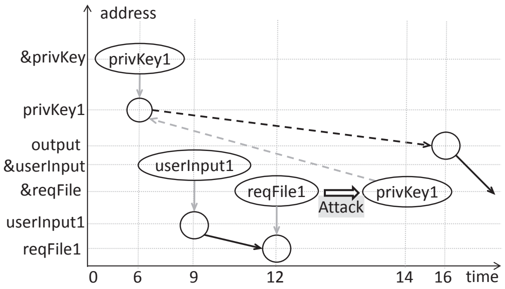


Figure 2: A data-oriented attack of Code 1. This attack connects flow of the private key and flow of the file name, with the new edges (dashed lines).

776 source vertices and 56 target vertices in one of SSHD attacks. Therefore, the search space to find a feasible path is large, for we often need heavy analysis to connect each pair of vertices.

- **C2. Limited knowledge of memory layout.** Most of the modern operating systems have enabled ASLR by default. The base addresses of data memory regions, like the stack and the heap, are randomized and thus are difficult to predict.

The 2D-DFG captures only the data dependencies in the execution, abstracting away control dependence and any conditional constraints the the program imposes along the execution path. To satisfy the requirements **R1** and **R2** completely, the following challenge must be addressed:

- **C3. Complex program path constraints.** A successful data-oriented attack causes the victim program execute to the memory error, create a stitch edge, and continue without crashing. This requires the input to satisfy all path constraints, respect the program's control flow integrity constraints, and avoid invalid memory accesses.

3 Data-flow Stitching

Data-oriented exploits can manipulate data-flow paths in a number of different ways to stitch the source and target vertices. The solution space can be categorized based on the number of new edges added by the exploit. The simplest case of data-oriented exploits is when the exploit adds a single new edge. More complex exploits that use a sequence of corrupted values can be crafted when a single-edge stitch is infeasible. We discuss these cases to solve challenge **C1** in Section 3.1 and 3.2. To overcome the challenge **C2**, we develop two methods to make data-oriented attacks work even when ASLR is deployed, discussed in Section 3.3. For each stitch candidate, we consider the path constraints and CFI requirement (**C3**) to generate input that trigger the stitch edge in Section 4.4.

```
1   struct passwd { uid_t pw_uid; ... } *pw;
2   ...
3   int uid = getuid();
4   pw->pw_uid = uid;
5   ...                  //format string error
6   void passive(void) { ...
7     seteuid(0);                 //set root uid
8     ...
9     seteuid(pw->pw_uid);  //set normal uid
10    ... }
```

Code 2: Code snippet of wu-ftpd, setting uid back to process user id.

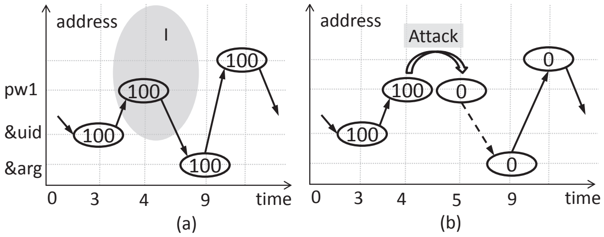


Figure 3: Target flow in the single-edge stitch of wu-ftpd. &arg is the stack address of setuid's argument. (a) is the original target flow, where the pw->pwd_uid has vale 100 and address $pw1$. Grey area stands for the memory influence I. With the stitching attack, the value at address $pw1$ is changed to 0 in (b).

3.1 Basic Stitching Technique

A basic data-oriented exploit adds one edge in the new edge set $\overline{E}$ to connect v_S with v_T. We call this case a *single-edge stitch*. For instance, attackers can create a single new vertex at the memory corruption point by overwriting a security-critical data value, causing escalation of privileges. Most of the previously known data-oriented attacks are cases of single-edge stitches, including attacks studied by Chen *et al.* [19] and the IE Safemode attack [6]. We use the example of a vulnerable web server wu-ftpd, shown in Code 2, which was used by Chen *et al.* to explain non-control data attacks [19]. In this exploit, the attackers utilizes a format string vulnerability (skipped on line 5) to overwrite the security-critical pw->pw_uid with root user's id. The subsequent setuid call on line 9, which is intended to drop the process privileges, instead makes the program retain its root user privileges. Figure 3 (a) and Figure 3 (b) show the 2D-DFG for the execution of the vulnerable code fragment under a benign and the exploit payload respectively. Numbers on time-axis are the line numbers in Code 2. The exploit aims to introduce a single edge to write a zero value from the network input to the memory allocated to the pw->pw_id. Note that the exploit is a valid path in the static control-flow graph.

Search for Single-Edge Stitch. Instead of brute-forcing all vertices in the target flow for a stitch edge, we propose a method that utilizes the influence set I of the mem-

StitchAlgo-1: Single-edge Stitch

```
Input: $G$: benign 2D-DFG,            $I$: memory influence,
       $v_T$: target vertex,          $cp$: memory error vertex,
       $X$: value to be in $V_T$.value (requirement for stitch edge)
Output: $\overline{E}$: stitch edge candidate set
$\overline{E} = \emptyset$
$TDFlow$ = dataSubgraph($G$, $v_T$)                    /* only data edges */
foreach $v \in V(TDFlow)$ do
    if $isRegister(v)$ then
        continue                                       /* Skip registers */
    if $\exists\ (v, v') \in E(TDFlow): \exists\ t : v.time < t < v'.time \wedge$
    $(v.address, t) \in I$ then
        $\overline{E} = \overline{E} \cup \{(cp, v)\}$           /* Stitch edge candidate */
```

ory error to prune the search space. The influence set I contains vertices that can be corrupted by the memory error, like the grey area shown in Figure 3. For vertices in the target flow, attackers can only affect those in the intersection of the target flow and the influence I. Other vertices do not yield a single-edge stitch and can be filtered out. Specifically, we utilize three observations here. First, register vertices can be ignored since memory error exploit cannot corrupt them. Second, the vertex must be defined (written) before the memory error and used (read) after the memory error. In Figure 3 (a), the code reads vertex (&uid, 3) before the memory error and writes vertices (&arg, 9) and the following one after the memory error. Therefore these three vertices are useless for single-edge stitches. Third, in the memory address dimension, the vertex address should belong to the memory region of the influence I. In our example, only vertex (pw1, 4) falls into the intersection of the target flow and the influence area and we select this vertex for stitch. StitchAlgo-1 shows the algorithm to identify single-edge stitch. From the given 2D-DFG, StitchAlgo-1 gets the target flow $TDFlow$ for the target vertex v_T, which only considers data edges. For each vertex v that satisfies the requirements, we add the edge from memory error vertex to v into $\overline{E}$ as one possible solution.

We consider the search space reduction due to our algorithm over a brute-force search for stitch edges. The naïve brute-force search would consider the Cartesian product of all vertices in the source flow and the target flow. In our algorithm, this search is reduced to the Cartesian product of only the live variables in the source flow at the time of corruption, and the vertices in the target flow as well as in I. In our experiments, we show that this reduction can be significant.

3.2 Advanced Stitching Technique

Single-edge stitch is a basic stitching method, creating one new edge. Advanced data-flow stitching techniques create paths with multiple edges in the new edge set $\overline{E}$. A *multi-edge stitch* can be synthesized through several ways. Attackers can use several single-edge stitches to create a multi-edge stitch. Another way is to perform *pointer stitch*, which corrupts a variable that is later used as a pointer to vertices in the source or target flow. Since the pointer determines the address of the stitch source or the stitch target, corrupting the pointer introduces two different edges: one edge for the new "points to" relationship and one edge for the changed data flow. We revisit the example of `wu-ftpd` shown earlier in Code 2, illustrating a multi-edge stitch exploit in it. Instead of directly modifying the field `pw_uid`, we change its base pointer `pw` to an address of a structure with a constant `0` at the offset corresponding to the `pw_uid`. The vulnerable code then reads `0` and uses it as the argument of `setuid`, creating a privilege escalation attack. Figure 4 shows the 2D-DFGs for the benign and attack executions. Changing the value of `pw` creates two new edges (dashed lines): the grey edge that connects the corrupted pointer to a new variable it points to, and the black edge that writes the new variable into `setuid` argument. As a result, we create a two-edge stitch.

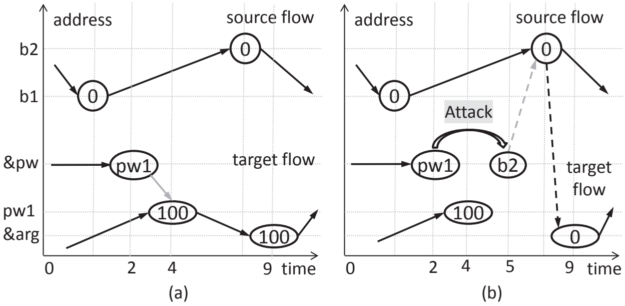


Figure 4: Two-edge stitch of `wu-ftpd`. The target flow is `pw->pw_uid`'s flow, and the source flow is the flow of a constant 0. With the attack, the variable `pw` at `&pw` is changed to $b2$. A later operation reads 0 from $b2$ and writes it to stack for `setuid`. Two edges are changed: one for pointer dereference and another for data movement.

Identifying Pointer Stitches. Our algorithm for finding multi-edge exploits using pointer stitching is shown in the StitchAlgo-2. The basic idea is to check each memory vertex in the source flow and the target flow. If it is pointed to by another vertex in the 2D-DFG, we select the pointer vertex to corrupt. The search for stitchable pointers on the target flow is different from that on the source flow. Specifically, for a vertex v in the target flow, we need to find an data edge (v', v) and a pointer vertex vp of v', and then change vp to point to a vertex vs in the source flow, so that a new edge (vs, v) will be created to stitch the data flows. For a vertex v in the source flow, we need to find an data edge (v, v') and a pointer vertex vp of v', and change vp to point to a vertex vt in the target flow, so that a new edge (v, vt) will be created to stitch the data flows. At the same time, we need to consider the liveness of the stitching vertices. For example, the source vertex

```
StitchAlgo-2: Pointer Stitch
Input: $G$: benign 2D-DFG,            $I$: memory influence,
       $v_S$: source vertex,          $v_T$: target vertex,
       $cp$: memory error vertex
Output: $\overline{E}$: stitch edge candidate set
$\overline{E} = \emptyset$
$SrcFlow$ = subgraph($G$, $v_S$)      /* both data and address edges. */
$TgtFlow$ = subgraph($G$, $v_T$)
$SDFlow$ = dataSubgraph($G$, $v_S$)        /* only data edges */
$TDFlow$ = dataSubgraph($G$, $v_T$)
foreach $v \in V(TDFlow)$ do
    if $isRegister(v)$ then continue
    if $\nexists\ (vi \in E(I) \wedge (v, v') \in TDFlow) : vi.time < v'.time$ then
    continue
    foreach $(vp, v) \in E(TgtFlow) - E(TDFlow)$ do
        /* Only consider address edges.                     */
        if $vp$ is used to write $v$ then continue
        /* Expect data flow from $v$                          */
        foreach $vs \in V(SDFlow)$ do
            if $\neg isRegister(vs) \wedge vs.isAliveAt(vp.time)$ then
                StitchAlgo-1($G$, $I$, $vp$, $cp$, $vs$.address)
foreach $v \in V(SDFlow)$ do
    if $isRegister(v)$ then continue
    if $\forall\ vi \in I: v.time < vi.time$ then continue
    foreach $(vp, v) \in E(SrcFlow) - E(SDFlow)$ do
        if $vp$ is used to read $v$ then continue
        /* Expect data flow into $v$                          */
        foreach $vt \in V(TDFlow)$ do
            if $\neg\ isRegister(vt) \wedge \exists (vt, v') \in TDFlow : vt.time$
            $< vp.time < v'.time$ then
                StitchAlgo-1($G$, $I$, $vp$, $cp$, $vt$.address)
```

should carry valid source data when it is used to write data out to the target vertex. Once we select the pointer vertex vp and its value (vt's or vs's address), the last step is to set the value into vp through the memory error exploit. StitchAlgo-2 invokes the basic stitching technique in StitchAlgo-1 to complete the last step.

Our technique uses vertex liveness and the memory error influence I to significantly reduce the search space. A naïve solution to finding pointer stitches would consider all pairs (vs, vt) where vs is in the source flow and vt is in the target flow. The search space will be the Cartesian product of the vertex set in the source flow (denoted as $V(SrcFlow)$) and the vertex set in the target flow (denoted as $V(TgtFlow)$). In contrast, in StitchAlgo-2, if the memory corruption occurs at time $t1$, the vertex used in the stitch edge from the source flow must be live at $t1$. Similarly, the vertex used in the stitch edge from the target flow should be created after $t1$. We illustrate it in Figure 5, where only the black vertices are candidates. Furthermore, we restrict our search to the set of vertices whose pointer vertices vp are inside the memory influence as well. We call the selected vertices from the source flow *R-set*. Similarly, we call the vertices selected from the target flow *W-set*. Our algorithm reduces the search space to the Cartesian product of the R-set and W-set instead.

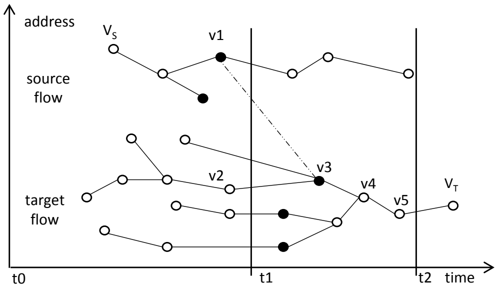


Figure 5: Stitch edge selection. The execution starts at time $t0$, and reaches memory error instructions at time $t1$. Target data is used at time $t2$, just before target vertex V_T. There are two stitch source candidates (black points in the source flow) and three stitch destination candidate (black points in the target flow). One of the stitch edge candidate is shown using the dotted line.

$$R\text{-}set = V(SrcFlow) \cap I, \quad W\text{-}set = V(TgtFlow) \cap I$$

$$|SS_{naive}| \;=\; |V(SrcFlow)| \times |V(TgtFlow)|$$

$$|SS_{pointer-stitch}| \;=\; |\ R\text{-}set\ | \times |\ W\text{-}set\ |$$

Pointer stitch constitutes a natural hierarchy of exploits, which can consist of multiple levels of dereferences of attacker-controlled pointers. For instance, in a *two-level pointer stitch* we can construct an exploit that corrupts a pointer vp_2 that points to the pointer vp. This can be achieved by treating vp as the target vertex, another pointer vp' holding the intended value (vt's or vs's address) as the source vertex and applying StitchAlgo-2 to change vp. In this case, StitchAlgo-2 is recursively used twice. Similarly, *N-level stitch* corrupts a pointer vp_N of the the pointer $vp_{(N-1)}$ to make an attack (and so on), by applying StitchAlgo-2 N times recursively. Note that for a N-level stitch to work, we need to make sure the source vertex vp'_N "aligns" with the target vertex vp_N at each level, such that the program dereferences vp_N N-1 times to get the vertex vp, and dereferences vp'_N N-1 time to get the intended value in the exploit.

Pointer stitch is one specific way to implement multi-edge stitches. In principle, it can be composed to create more powerful exploits, combining several other single-edge stitches in a "multi-step" stitch attack. In a multi-step stitch, several intermediate data flows are used to achieve data-flow stitching. Each step can be realized by pointer stitch or single-edge stitch. Multi-step stitch is useful when direct stitches of the source flow and the target flow are not feasible.

3.3 Challenges from ASLR

Address space layout randomization (ASLR) deployed by modern systems poses a strong challenge in mounting

Table 1: Deterministic memory region size of binaries on Ubuntu 12.04 x86 system. Position-independent executables have size 0. Two largest numbers are highlighted for each directory.

size (KB)	/bin	/sbin	/usr/bin	/usr/sbin	Total
0	21	22	73	18	134
1 - 8	10	**33**	**150**	**20**	**213**
8 - 16	12	17	113	11	153
16 - 32	**23**	17	**147**	14	**201**
32 - 64	**19**	**22**	103	**25**	169
64 - 128	15	8	66	8	97
128 - 256	7	2	35	4	48
256 - 512	3	2	32	3	40
> 512	2	2	32	2	38
Total	112	125	751	105	1093

successful data-oriented attacks since vertex addresses are highly unpredictable. We develop two methods in data-oriented attacks to address this challenge: stitching with deterministic addresses and stitching by address reuse. Note that attackers can use others methods developed for control flow attacks to bypass ASLR here, like disclosure of random addresses [14, 35].

3.3.1 Stitching With Deterministic Addresses

When security-critical data is stored in deterministic memory addresses, stitching data flows of such data is not affected by ASLR. Existing work [2, 34, 37] have shown that current ASLR implementations leave a large portion of program data in the deterministic memory region. For example, Linux binaries are often compiled without the "*-pie*" option, resulting in deterministic memory regions. We study deterministic memory size of Ubuntu 12.04 (x86) binaries under directories `/bin`, `/sbin`, `/usr/bin` and `/usr/sbin`, and show the results in Table 1. Among 1093 analyzed programs, more than 87.74% have deterministic memory regions. Two hundred and twenty-three programs have deterministic memory regions larger than 64KB. Inside such memory regions, there is many security-critical data, like randomized addresses in `.got.plt` and configuration structures in `.bss`. Hence we believe stitch with deterministic addresses in real-world programs is practical.

We build an information leakage attack against the `orzhttpd` web server [5] (details in Section 6.4.3) using the stitch with deterministic addresses. To respond to a page request, `orzhttpd` uses a pointer to retrieve the HTTP protocol version string. The pointer is stored in memory. If we replace the pointer value with the address of a secret data, the server will send that secret to the client. However this requires both the address of the pointer and the address of the secret to be predictable. In the `orzhttpd` example, we find that the address of the pointer is fixed (`0x8051164`) and choose the contents of the `.got.plt` section (allocated at a fixed address) as the secret to leak out. Figure 6 shows two 2D-DFGs for the benign execution and the attack, respectively. With this attack, the content of `.got.plt` is sent to the attacker, which leads to an memory address disclosure exploit useful for constructing second-stage control-hijacking attacks or stealing secret data in randomized memory region. Unlike a direct memory disclosure attack, here we use the corruption of deterministically-allocated data to leak randomized addresses.

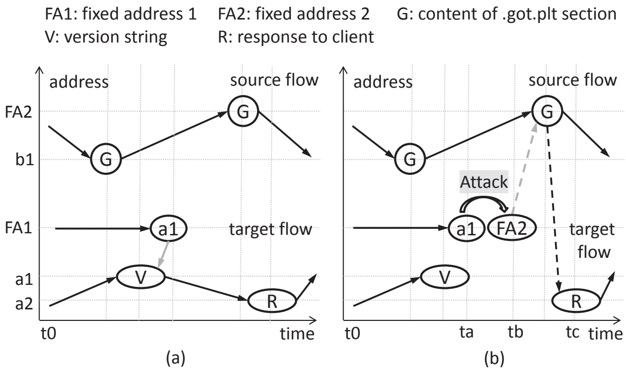


Figure 6: Stitch with deterministic memory addresses of the `orzhttpd` server. This attack is the similar to the one in Figure 4, except the address of the source vertex and the pointer's address of of the target vertex are fixed. This attack works with ASLR.

Identifying Stitch with Deterministic Addresses. We represent the deterministic memory region as a set *D*. Our algorithm considers the intersection of *D* for the vertices in the source flow and the target flow. The previously outlined stitching algorithms can then be used directly prioritizing the vertices in the intersection with *D*.

3.3.2 Stitching By Address Reuse

If the security-critical data only exists inside the randomized memory region, data-oriented attacks cannot use deterministic addresses. To bypass ASLR in such cases, we leverage the observation that a lot of randomized addresses are stored in memory. If we can reuse such real-time randomized addresses instead of providing concrete address in the exploit, the generated data-oriented attacks will be stable (agnostic to address randomization). There are two types of address reuse: partial address reuse and complete address reuse.

Partial Address Reuse. A variable's relative address, with respect to the module base address or with respect to another variable in the same module, is usually fixed. Attackers can easily calculate such relative addresses in advance. On the other hand, instructions commonly get a memory address with one base address and one relative offset (e.g., array access, switch table). If attackers control the offset variable, they can corrupt the offset with the pre-computed relative address from the selected vertex (source vertex or target vertex) and reuse the randomized base address. In this way attackers can access

```
1  struct user_details { uid_t uid; ... } ud;
2  ...                   //run with root uid
3  ud.uid = getuid();    //in get_user_info()
4  ...
5  vfprintf(...);        //in sudo_debug()
6  ...
7  setuid(ud.uid);       //in sudo_askpass()
8  ...
```

Code 3: Code snippet of sudo, setting uid to normal user id.

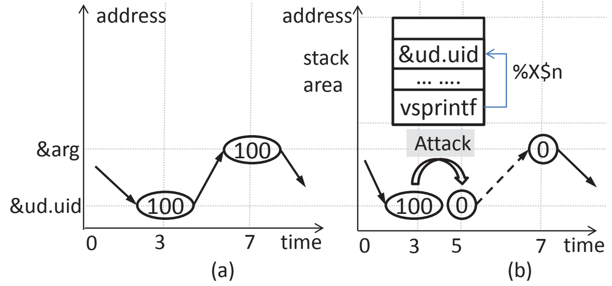


Figure 7: Stitch by complete memory address reuse of sudo. The dashed line is the new edge (single-edge stitch). An address of *ud.uid* exists on ancestor's stack frame, which is reused to overwrite *ud.uid*.

the intended data without knowing their randomized addresses. We show an example of a vulnerable instruction pattern, that allows the attacker partial ability to read a value from memory and write it out without knowing randomized addresses. If attackers control %eax, they can reuse the source base address %esi in the first instruction, and reuse the destination base address %edi in the second instruction. In fact, any memory access instruction with a corrupted offset can be used to mount partial address reuse attack.

```
//attackers control %eax
mov   (%esi,%eax,4), %ebx   //reuse %esi
mov   %ecx, (%edi,%eax,4)   //reuse %edi
```

Complete Address Reuse. We observe that a variable's address is frequently saved in memory due to the limitation of CPU registers. If the memory error allows retrieving such spilled memory address for reading or writing, attackers can reuse the randomized vertex address existing in memory to bypass ASLR. For example, in the following assembly code, if attacker controls %eax on line 1, it can load a randomized address into %ebx from memory. Then, attacker can access the target vertex pointed by %ebx without knowing the concrete randomized address. The attacker merely needs to know the right offset value to use in %eax on line 2, or may have a deterministic %esi value to gain arbitrary control over addresses loaded on line 2.

```
1  //attacker controls %eax
2  mov   (%esi, %eax, 4), %ebx
3  mov   %ecx, (%ebx) / mov (%ebx), %ecx
```

Let us consider a real example of the sudo program [9] that shows how to use such instruction patterns that permit complete address reuse meaningfully. Code 3 shows the related code of sudo, where a format string vulnerable exists in the sudo_debug function (line 5). At the time of executing vfprintf() on line 5, the address of the user identity variable (ud.uid) exists on the stack. The vfprintf() function with format string "%$X$$n" uses the Xth argument on stack for "%n". By specifying the value of X, vfprintf() can retrieve the address of ud.uid from its ancestor's stack frame and change the ud.uid to the root user ID without knowing the stack base address. Figure 7 shows the 2D-DFGs for the benign execution and the attack. This attack works even if the fine-grained ASLR is deployed.

Identifying Stitch by Address Reuse. Memory error instructions for address reuse stitch should match the patterns we discuss above. For partial address reuse, the memory error exploit corrupts variable offsets, while for complete address reuse, the memory error exploit can retrieve addresses from memory. Our approach intersects the memory error influence I with the source flow and the target flow. Then we search from the new source flow and the new target flow to identify matched instructions, from which we can build stitch by address reuse with methods discuss above.

4 The FlowStitch System

We design a system called FlowStitch to systematically generate data-oriented attacks using data-flow stitching. As shown in Figure 8, FlowStitch takes three inputs: a program with memory errors, an error-exhibiting input, and a benign input of the program. The two inputs should drive the program execution down the same execution path until the memory error instruction, with the error-exhibiting input causing a crash. FlowStitch builds data-oriented attacks using the memory errors in five steps. First, it generates the execution trace for the given program. We call the execution trace with the benign input the *benign trace*, and the execution trace with the error-exhibiting input the *error-exhibiting trace*. Second, FlowStitch identifies the influence of the memory errors from the error-exhibiting trace and generates constraints on the program input to reach memory errors. Third, FlowStitch performs data-flow analysis and security-sensitive data identification using the benign trace. Fourth, FlowStitch selects stitch candidates from the identified security-sensitive data flows with the methods discussed in Section 3. Finally, FlowStitch checks the feasibility of creating new edges with the memory errors and validates the exploit. It finally outputs the input to mount a data-oriented attack.

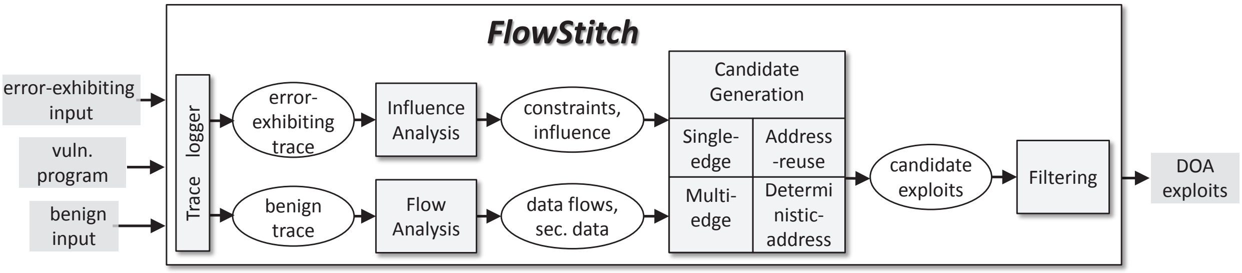


Figure 8: Overview of FLOWSTITCH. FLOWSTITCH takes a vulnerable program, an error-exhibiting input and a benign input of the program as inputs. It builds data-oriented attacks against the given program using data-flow stitching. Finally it outputs the data-oriented attack exploits.

FLOWSTITCH requires that the error-exhibiting input and the benign input make the program follow the same code path until memory error happens. Such pairs of inputs can be found by existing symbolic execution tools, like BAP [16] and SAGE [25], which explore multiple execution paths with various inputs. Before detecting one error-exhibiting execution, these tools usually have explored many matched benign executions.

4.1 Memory Error Influence Analysis

FLOWSTITCH analyzes the error-exhibiting trace to understand the influence I of the memory errors. It identifies two aspects of the memory error influence: the time when the memory errors happens during the execution (temporal influence) and the memory range that can be written to in the memory error (spatial influence). From the error-exhibiting trace, FLOWSTITCH detects instructions whose memory dereference addresses are derived from the error-exhibiting input. We call these instructions *memory error instructions*. Note that data flows ending before such instructions or starting after them cannot be affected by the memory error, therefore they are out of the temporal influence.

Attackers get access to unintended memory locations with memory error instructions. However, the program's logic limits the total memory range accessible to attackers. To identify the spatial influence of the memory error instruction, we employ dynamic symbolic execution techniques. We generate a symbolic formula from the error-exhibiting trace in which all the inputs are symbolic variables and all the path constraints are asserted true. Inputs that satisfy the formula imply that the execution to memory error instructions with an unintended address[3]. The set of addresses that satisfy these constraints and can be dereferenced at the memory error instruction constitute the spatial influence.

4.2 Security-Sensitive Data Identification

As we discuss in Section 2.3, FLOWSTITCH synthesizes flows of security-sensitive data. There are four types of data that are interesting for stitching: input data, output data, program secret and permission flags. To identify input data, FLOWSTITCH performs taint analysis at the time of trace generation, treating the given input as an external taint source. For output data, FLOWSTITCH identifies a set of program sinks that send out the program data, like `send()` and `printf()`. The parameters used in sinks are the output data. Further, we classify program secret and permission flags into two categories: the program-specific data and the generic data. FLOWSTITCH accepts user specification to find out program-specific data. For example, user can provide addresses of security flags. For the generic data, FLOWSTITCH uses the following methods to automatically infer it.

- **System call parameters.** FLOWSTITCH identifies all system calls from the trace, like `setuid`, `unlink`. Based on the system call convention, FLOWSTITCH collects the system call parameters.
- **Configuration data.** To identify configuration data, FLOWSTITCH treats the configuration file as a taint source and uses taint analysis to track the usage of the configuration data.
- **Randomized data.** FLOWSTITCH identifies stack canary based on the instructions that set and check the canary, and identifies randomized addresses if they are not inside the deterministic memory region.

Deterministic Memory Region Identification. FLOWSTITCH identifies the deterministic memory region for stitch with deterministic addresses (Section 3.3.1). It first checks the program binary to identify the memory regions that will not be randomized at runtime. If the program is not position-independent, all the data sections shown in the binary headers will be at deterministic addresses. FLOWSTITCH collects loadable sections and gets a deterministic memory set D. FLOWSTITCH further scans benign traces to find all the memory writing instructions that write data into the deterministic memory set to identify data stored in such region.

[3] This is true if the symbolic formula constructed is complete [26].

Note that based on the functionality of the security-sensitive data, we predefine goals of the attacks. For example, the attack of `setuid` parameter is to change it to the root user's id 0. For a web server's home directory string, the goal is to set it to system root directory.

4.3 Stitching Candidate Selection

For identified security-sensitive data, FLOWSTITCH generates its data flow from the 2D-DFG. FLOWSTITCH selects the source flow originated from the source vertex V_S and the target flow ended at the target vertex V_T. It then uses the stitching methods discussed in Section 3 to find stitching solutions. Although any combination of stitching methods can be used here, FLOWSTITCH uses the following policy in order to produce a successful stitching efficiently.

1. FLOWSTITCH tries the single-edge stitch technique before the multi-edge stitch technique. After the single-edge stitch's search space is exhausted, it moves to multi-edge stitch. FLOWSTITCH stops searching at four-edge stitch in our experiments.
2. FLOWSTITCH considers stitch with deterministic addresses before stitch by address reuse. After exhausting the search space of deterministic address and address reuse space, FLOWSTITCH continues searching stitches with concrete addresses shown in benign traces, for cases without ASLR.

4.4 Candidate Filtering

To overcome challenge **C3**, FLOWSTITCH checks the feasibility of each selected stitch edge candidate. We define the *stitchability constraint* to cover the following constraints.

- Path conditions to reach memory error instructions;
- Path conditions to continue to the target flow;
- Integrity of the control data;

FLOWSTITCH generates the stitchability constraint using symbolic execution tools. The constraint is sent to SMT solvers as an input. If the solver cannot find any input satisfying the constraint, FLOWSTITCH picks the next candidate stitch edge. If it exists, the input will be the witness input that is used to exercise the execution path in order to exhibit the data-oriented attack. Due to the concretization in symbolic constraint generation in the implementation, the constraints might not be complete [26], i.e., it may allow inputs that results in different paths. FLOWSTITCH concretely verifies the input generated by the SMT solver to check if it successfully mounts the data-oriented attack on the program.

5 Implementation

We prototype FLOWSTITCH on Ubuntu 12.04 32 bit system. Note that as the first step the trace generation tool can work on both Windows and Linux systems to generate traces. Although the following analysis steps are performed on Ubuntu, FLOWSTITCH works for both Windows and Linux binaries.

Trace Generation. Our trace generation is based on the Pintraces tool provided by BAP [16]. Pintraces is a Pin [28] tool that uses dynamic binary instrumentation to record the program execution status. It logs all the instructions executed by the program into the trace file, together with the operand information. In our evaluation, the traces also contain dynamic taint information to facilitate the extraction of data flows.

Data Flow Generation. For input data and configuration data, FLOWSTITCH uses the taint information to get the data flows. To generate the data flow of the security-sensitive data, FLOWSTITCH performs backward and forward slicing on the benign trace to locate all the related instructions. It is possible for one instruction to have multiple source operands. For example, in `add %eax, %ebx`, the destination operand `%ebx` is derived from `%eax` and `%ebx`. In this case, one vertex has multiple parent vertices. As a result, the generated data flow is a graph where each node may have multiple parents.

Constraint Generation and Solving. The generation of the stitchability constraint required in Section 4.4 is implemented in three parts: path constraints, influence constraints, and CFI constraints.The stitchability constraint is expressed as a logical conjunction of these three parts. We use BAP to generate formulas which capture the path conditions and influence constraints. For control flow integrity constraint, we implement a procedure to search the trace for all the indirect `jmp` or `ret` instruction. Memory locations holding the return addresses or indirect jump targets are recorded. The control flow integrity requires that at runtime, the memory location containing control data should not be corrupted by the memory errors. The stitchability constraint is checked for satisfiability using the Z3 SMT-solver [22], which produces a witness input when the constraint is satisfiable.

6 Evaluation

In this section, we evaluate the effectiveness of data-flow stitching using FLOWSTITCH, including single-edge stitch, multi-edge stitch, stitch with deterministic addresses and stitch by address reuse. We also measure the search space reduction using FLOWSTITCH and the performance of FLOWSTITCH.

6.1 Efficacy in Exploit Generation

Table 2 shows the programs used in our evaluation, as well as their running environments and vulnerabilities. The trace generation phase is performed on different

Table 2: Experiment environments and benchmarks. # of Data-Oriented Attacks gives the number of attacks generated by FlowStitch, including privilege escalation attacks and information leakage attacks. FlowStitch generates 19 data-oriented attacks from 8 vulnerable programs.

ID	Vul. Program	Vulnerability	Environment (32b)	# of Data-Oriented Attacks	
				Escalation	Leakage
CVE-2013-2028	nginx	stack buffer overflow	Ubuntu 12.04	1	1
CVE-2012-0809	sudo	format string	Ubuntu 12.04	1	0
CVE-2009-4769	httpdx	format string	Windows XP SP3	4	1
bugtraq ID: 41956	orzhttpd	format string	Ubuntu 9.10	1	1
CVE-2002-1496	null httpd	heap overflow	Ubuntu 9.10	2	0
CVE-2001-0820	ghttpd	stack buffer overflow	Ubuntu 12.04	1	0
CVE-2001-0144	SSHD	integer overflow	Ubuntu 9.10	2	1
CVE-2000-0573	wu-ftpd	format string	Ubuntu 9.10	2	1
Total	8 programs			14	5

Table 3: Evaluation of FlowStitch on generating data-oriented attacks. In the Attack Description column, L_i stands for information leakage attack, while M_i represents privilege escalation attack. The third column indicates whether the built attack can bypass ASLR or not. The "CP" column shows the number of memory error instructions. Trace size is the number of instructions inside the trace. The last four columns show the number of stitch sources and stitch targets before and after our selection. SrcFlow means source flow, while TgtFlow stands for target flow.

Vul. Apps	Attack Description	ASLR Bypass	CP	Error-exhibiting Trace Size	Benign Trace Size	# of nodes before		# of nodes after	
						SrcFlow	TgtFlow	SrcFlow	TgtFlow
nginx	L_0: private key		1	50789	411437	3	48	3	1
	M_0: http directory path				1717182	173	462	1	42
sudo	M_0: user id	✓	1	351988	854371	2083	1	1	1
httpdx	L_0: admin's password	✓	1	1197657	1361761	152	7	152	2
	M_0: admin's password	✓			1298247	78	120	1	8
	M_1: anon.'s permission	✓			1233522	78	2	1	1
	M_2: anon.'s root directory	✓			1522672	78	165	1	11
	M_3: CGI directory path	✓			1257694	78	480	1	30
orzhttpd	L_0: randomized address	✓	1	84694	131871	8	28	8	1
	M_0: directory path	✓			131871	368	95	1	19
null httpd	M_0: http directory path		2	160844	401285	3	141	2	47
	M_1: CGI directory path				335329	3	144	2	48
ghttpd	M_0: CGI directory path		1	312130	316473	3579	6	1	1
SSHD	L_0: root password hash		1	38201	3094592	776	56	97	2
	M_0: user id				674365	1	24	1	1
	M_1: authenticated flag				674365	1	2	1	1
wu-ftpd	L_0: env. variables		1	328108	1417908	88	5	88	1
	M_0: user id (single-edge)	✓			1057554	183	2	1	1
	M_1: user id (multi-edge)	✓			1057554	183	1	1	1

systems according to the tested program. All generated traces are analyzed by FlowStitch on a 32-bit Ubuntu 12.04 system. The vulnerabilities used for the experiments come from four different categories to ensure that FlowStitch can handle different vulnerabilities. Seven of the 8 vulnerable programs are server programs, including HTTP and FTP servers, which are the common targets of remote attacks. The other one is the `sudo` program, which allows users to run command as another user on Unix-like system. The last four vulnerabilities were discussed in [19], where data-oriented attacks were manually built. We apply FlowStitch on these vulnerabilities to verify the efficacy of our method.

Results. Our result demonstrates that FlowStitch can effectively generate data-oriented attacks with different vulnerabilities on different platforms. The number of generated data-oriented attacks on each program is shown in Table 2 and their details are given in Table 3. FlowStitch generates a total of 19 data-oriented attacks for eight real-world vulnerable programs, more than two attacks per program on average. Among 19 data-oriented attacks, there are five information leakage attacks and 14 privilege escalation attacks. For the vulnerable `httpdx` server, FlowStitch generates five data-oriented attacks from a format string vulnerability.

Out of the 19 data-oriented attacks, 16 are previously unknown. The three known attacks are two `uid`-corruption attacks on `SSHD` and `wu-ftpd`, and a CGI directory corruption attack on `null httpd`, discussed in [19]. FlowStitch successfully reproduces known attacks and builds new data-oriented attacks with the same vulnerabilities. Note that FlowStitch produces a different `ghttpd` CGI directory corruption attack than the one described in [19]. Details of this attack are discussed in Section 6.4.2. The results show the efficacy of our systematic approach in identifying new data-oriented attacks.

From our experiments, seven out of 19 of the data-oriented attacks are generated using multi-edge stitch. The significant number of new data-oriented attacks generated by multi-edge stitch highlights the importance of a systematic approach in managing the complexity and identifying new data-oriented attacks. As a measurement of the efficacy of ASLR on data-oriented attacks, we report that 10 of 19 attacks work even with ASLR deployed. Among 10 attacks, two attacks reuse randomized addresses on the stack and eight attacks corrupt data in the deterministic memory region. We observe that security-sensitive data such as configuration option is usually represented as a global variable in C programs and reside in the `.bss` segment. This highlights the limitation of current ASLR implementations which randomize the stack and heap addresses but not the `.bss` segment.

For three of 19 attacks, FLOWSTITCH requires the user to specify the security-sensitive data, including the private key of `nginx`, the root password hash and the authenticated flag of `SSHD`. For others, FLOWSTITCH automatically infers the security-sensitive data using techniques discussed in Section 4.2. Once such data is identified, FLOWSTITCH automatically generates data-oriented exploits.

6.2 Reduction in Search Space

Data-flow stitching has a large search space due to the large number of vertices in the flows to be stitched. Manual checking through a large search space is difficult. For example, in the root password hash leakage attack against `SSHD` server, there are 776 vertices in source flow containing the hashed root passwords. In the target flow, there are 56 vertices leading to the output data. Without considering the influence of the memory errors, there are a total of 43,456 possible stitch edges. After applying the methods described in Section 3, we get the intersection of the memory error influence I with the stitch source set R-set and the stitch target set W-set. In this way, the number of candidate edges is reduced from 43,456 to 194, obtaining a reduction ratio of 224.

The last four columns in Table 3 give the detailed information of the search space for each attack. For most of the data-oriented attacks, there is a significant reduction in the number of possible stitches. `ghttpd`-M_0 achieves the highest reduction ratio of 21,474 while `SSHD`-M_1 achieves the lowest reduction ratio of two. The median reduction ratio is 183 achieved by `wu-ftpd`-M_1(multi-edge). Given the relatively large spatial influence of the memory error, most of the reduction is achieved by the temporal influence of I.

6.3 Performance

Table 4: Performance of trace and flow generation using FLOWSTITCH. The unit used in the table is second, so 1:07 means one minute and seven seconds.

Attacks		Trace Gen		Slicing		Total
		error	benign	error	benign	
nginx	L_0	0:08	0:22	0:06	2:41	3:17
	M_0		0:36		0:12	1:02
sudo	M_0	0:35	1:07	1:17	3:34	6:33
httpdx	L_0	0:08	0:45	0:12	5:56	7:01
	M_0		0:51		4:44	5:55
	M_1		0:50		4:52	6:02
	M_2		1:03		4:45	6:08
	M_3		0:53		4:47	6:00
orzhttpd	L_0	0:17	0:20	0:12	0:24	1:13
	M_0		0:20		1:04	1:53
null httpd	M_0	0:13	1:20	0:14	6:21	8:08
	M_1		0:52		2:29	3:48
ghttpd	M_0	0:09	0:18	0:12	0:09	0:48
SSHD	L_0	2:35	9:38	1:02	21:08	34:23
	M_0		5:30		1:22	10:29
	M_1		5:30		1:00	10:07
wu-ftpd	L_0	0:12	0:50	0:19	5:42	7:03
	M_0		0:31		0:27	1:29
	M_1		0:31		0:26	1:28
Average		0:32	1:41	0:26	3:47	6:27

We measure the time FLOWSTITCH uses to generate data-oriented attacks. Table 4 shows the results, including the time of trace generation and the time of data-flow collection (slicing). Note that the trace generation time includes the time to execute instructions that are not logged (e.g., `crypto` routines and `mpz` library for `SSHD`). As we can see from Table 4, FLOWSTITCH takes an average of six minutes and 27 seconds to generate the trace and flows. Most of them are generated within 10 minutes. The information leakage attack of `SSHD` server takes the longest time, 34 minutes and 23 seconds, since crypto routines execute a large number of instructions. From the performance results, we can see that the generation of data flows through trace slicing takes up most of the generation time, from 20 percent to 87 percent. Currently, our slicer works on BAP IL file. We plan to optimize the slicer using parallel tools in the future.

6.4 Case Studies

We present five case studies to demonstrate the effectiveness of stitching methods and interesting observations.

6.4.1 Sensitive Data Lifespan

A common defense employed to reduce the effectiveness of data-oriented attacks is to limit the lifespan of security-critical data [19, 20]. This case study highlights the difficulty of doing it correctly. In the implementation of `SSHD`, the program explicitly zeros out sensitive data, such as the RSA private keys, as soon as they are not in use. For password authentication on Linux, `getspnam()` provided by glibc is often used to obtain the password hash. Rather than using the password hash directly, `SSHD` makes a local copy of the password hash on stack for its use. Although the program makes no special effort is to clear the copy on the stack, the password on stack is eventually overwritten by subsequent function frames before it can be leaked. The developer explicitly deallocates the original hash value using `endspent()` [1] in the glibc internal data structures. However, glibc does not clear the deallocated memory after `endspent()` is called and this allows FLOWSTITCH to successfully leak the hash from the copy held by glibc. Hence, this case study highlights that sensitive information should not be kept by the program after usage, and that identifying all copies of sensitive data in memory is difficult at the source level.

6.4.2 Multi-edge Stitch – ghttpd CGI Directory

The `ghttpd` application is a light-weight web server supporting CGI. A stack buffer overflow vulnerability was reported in version 1.4.0 - 1.4.3, allowing remote attackers to smash the stack of the vulnerable `Log()` function. During the security-sensitive data identification, FLOWSTITCH detects `execv()` is used to run an executable file. One of `execv()`'s arguments is the address of the program path string. Controlling it allows attackers to run arbitrary commands. FLOWSTITCH is unable to find a new data dependency edge using single-edge stitching, since there is no security-sensitive data on the stack frame to corrupt. FLOWSTITCH then proceeds to search for a multi-edge stitch. For the program path parameter of `execv()`, FLOWSTITCH identifies its flow, which includes use of a series of stack frame-base pointers saved in memory. The temporal constraints of the memory error exploit only allow the saved `%ebp` of the `Log()` function to be corrupted. Once the `Log()` function returns, the saved `%ebp` is used as a pointer, referring to all the local variables and parameters of `Log()` caller's stack frame. FLOWSTITCH corrupts the saved `%ebp` to change the variable for the CGI directory used in `execv()` system call. This attack is a four-edge stitch by composing two pointer stitches.

Chen *et al.* [19] discussed a data-oriented attack with the same vulnerability, which was in fact a two-edge stitch. However, that attack no longer works in our experiment. The `ghttpd` program compiled on our Ubuntu 12.04 platform does not store the address of command string on the stack frame of `Log()`. Only the four-edge stitching can be used to attack our `ghttpd` binary.

6.4.3 Bypassing ASLR – orzhttpd Attacks

The `orzhttpd` web server has a format string vulnerability which the attacker can exploit to control almost the whole memory space of the vulnerable program. FLOWSTITCH identifies the deterministic memory region and the randomized address on stack under `fprintf()` frame. The first attack which bypasses ASLR is a privilege escalation attack. This attack corrupts the web root directory with single-edge stitching and memory address reuse. The root directory string is stored on the heap, which is allocated at runtime. FLOWSTITCH identifies the address of the heap string from the stack and reuses it to directly change the string to `/` based on the pre-defined goal (Section 4.2). The second attack is an information leakage attack, which leaks randomized addresses in the `.got.plt` section. FLOWSTITCH identifies the deterministic memory region from the binary and performs a multi-edge stitch. The stitch involves modifying the pointer of an HTTP protocol string stored in a deterministic memory region. FLOWSTITCH changes the pointer value to the address of `.got.plt` section and a subsequent call to send the HTTP protocol string leaks the randomized addresses to attackers.

6.4.4 Privilege Escalation – Nginx Root Directory

The `Nginx` HTTP server 1.3.9-1.4.0 has a buffer overflow vulnerability [4]. FLOWSTITCH checks the local variables on the vulnerable stack and identifies two data pointers that can be used to perform arbitrary memory corruption. The memory influence of the overwriting is limited by the program logic. FLOWSTITCH identifies the web root directory string from the configuration data. It tries single-edge stitching to corrupt the root directory setting. The root directory string is inside the memory influence of the arbitrary overwriting. FLOWSTITCH overwrites the value `0x002f` into the string location, thus changing the root directory into `/`. FLOWSTITCH verifies the attack by requesting `/etc/passwd` file. As a result, the server sends the file content back to the client.

6.4.5 Information Leakage – httpdx Password

The `httpdx` server has a format string vulnerability between version 1.4 to 1.5 [3]. The vulnerable `tolog()` function records FTP commands and HTTP requests into a server-side log file. Note that direct exploitation of this vulnerability does not leak information. Using the error-exhibiting trace, FLOWSTITCH identifies the memory error instruction and figures out that there is almost no limitation on the memory range affected by attackers. From the `httpdx` binary, FLOWSTITCH manages to find a total of 102MB of deterministic memory addresses. From the benign trace, FLOWSTITCH generates data flows of the root user passwords. This is the secret to be leaked out. The FLOWSTITCH generates the necessary data flow which reaches the `send()` system call automatically.

Starting from the memory error instruction, FLOWSTITCH searches backwards in the secret data flow and identifies vertices inside the deterministic memory region. FLOWSTITCH successfully finds two such memory locations containing the “admin” password: one is a buffer containing the whole configuration file, and another only contains the password. At the same time, FLOWSTITCH searches forwards in the output flow to find the vertices that affect the buffer argument of `send()`. Our tool identifies vertices within the deterministic memory region. The solver gives one possible input that will trigger the attack. FLOWSTITCH confirms this attack by providing the attack input to the server and receiving the "admin" user password.

7 Related Work

Data-Oriented Attack. Several work [21,32,36,38,41,43,44] has been done to improve the practicality of CFI, increasing the barrier to constructing control flow hijacking attacks. Instead, data-oriented attacks are serious alternatives. Data-oriented attacks have been conceptually known for a decade. Chen *et al.* constructed non-control-data exploits to show that data-oriented attack is a realistic threat [19]. However, no systematic method to develop data-oriented attacks is known yet. In our paper, we develop a systematic way to search for possible data-oriented attacks. This method searches attacks within the candidate space efficiently and effectively.

Automatic Exploit Generation. Brumley *et al.* [17] described an automatic exploit generation technique based on program patches. The idea is to identify the difference between the patched and the unpatched binaries, and generate an input to trigger the difference. Avgerinos *et al.* [13] discussed Automatic Exploit Generation(AEG) to generate real exploits resulting in a working shell. Felmetsger *et al.* [24] discussed automatic exploit generation for web applications. The previous work focused on generating control flow hijacking exploits. FLOWSTITCH on the other hand generates data-oriented attacks that do not violate the control flow integrity. To our knowledge, FLOWSTITCH is the first tool to systematically generate data-oriented attacks.

Defenses against Data-Oriented Attacks. Data-oriented attacks can be prevented by enforcing data-flow integrity (DFI). Existing work enforces DFI through dynamic information tracking [23, 39, 40] or by legitimate memory modification instruction analysis [18,42]. However, DFI defenses are not yet practical, requiring large overheads or manual declassification. An ultimate defense is to enforce the memory safety to prevent the attacks in their first steps. Cyclone [27] and CCured [31] introduce a safe type system to the type-unsafe C languages. SoftBound [29] with CETS [30] uses bound checking with fat-pointer to force a complete memory safety. Cling [11] enforces temporal memory safety through type-safe memory reuse. Data-oriented attack prevention requires a complete memory safety.

8 Conclusion

In this paper, we present a new concept called data-flow stitching, and develop a novel solution to systematically construct data-oriented attacks. We discuss novel stitching methods, including single-edge stitch, multi-edge stitch, stitch with deterministic addresses and stitch by address reuse. We build a prototype of data-flow stitching, called FLOWSTITCH. FLOWSTITCH generates 19 data-oriented attacks from eight vulnerable programs. Sixteen attacks are previously unknown attacks. All attacks bypass DEP and the CFI checks, and 10 bypass ASLR. The result shows that automatic generation of data-oriented exploits exhibiting significant damage is practical.

Acknowledgments. We thank R. Sekar, Shweta Shinde, Yaoqi Jia, Xiaolei Li, Shruti Tople, Pratik Soni and the anonymous reviewers for their insightful comments. This research is supported in part by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate, and in part by a research grant from Symantec.

References

[1] Endspent(3C). https://docs.oracle.com/cd/E36784_01/html/E36874/endspent-3c.html.

[2] How Effective is ASLR on Linux Systems? http://securityetalii.es/2013/02/03/how-effective-is-aslr-on-linux-systems/.

[3] HTTPDX tolog() Function Format String Vulnerability. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-4769.

[4] Nginx HTTP Server 1.3.9-1.4.0 Chunked Encoding Stack Buffer Overflow. http://mailman.nginx.org/pipermail/nginx-announce/2013/000112.html.

[5] OrzHTTPd. https://code.google.com/p/orzhttpd/.

[6] Subverting without EIP. http://mallocat.com/subverting-without-eip/.

[7] The Heartbleed Bug. http://heartbleed.com/.

[8] Visual Studio 2015 Preview: Work-in-Progress Security Feature. http://blogs.msdn.com/b/vcblog/archive/2014/12/08/visual-studio-2015-preview-work-in-progress-security-feature.aspx.

[9] Sudo Format String Vulnerability. http://www.sudo.ws/sudo/alerts/sudo_debug.html, 2012.

[10] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (2005).

[11] AKRITIDIS, P. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *Proceedings of the 19th USENIX Security Symposium* (2010).

[12] ANDERSEN, S., AND ABELLA, V. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory protection technologies, Data Execution Prevention. Microsoft TechNet Library, September 2004.

[13] Avgerinos, T., Cha, S. K., Hao, B. L. T., and Brumley, D. AEG: Automatic Exploit Generation. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium* (2011).

[14] Backes, M., Holz, T., Kollenda, B., Koppe, P., Nürnberger, S., and Pewny, J. You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code. In *Proceedings of the 21st ACM Conference on Computer and Communications Security* (2014).

[15] Bhatkar, S., DuVarney, D. C., and Sekar, R. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the 12th USENIX Security Symposium* (2003).

[16] Brumley, D., Jager, I., Avgerinos, T., and Schwartz, E. J. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (2011).

[17] Brumley, D., Poosankam, P., Song, D., and Zheng, J. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *Proceedings of the 29th IEEE Symposium on Security and Privacy* (2008).

[18] Castro, M., Costa, M., and Harris, T. Securing Software by Enforcing Data-Flow Integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (2006).

[19] Chen, S., Xu, J., Sezer, E. C., Gauriar, P., and Iyer, R. K. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the 14th USENIX Security Symposium* (2005).

[20] Chow, J., Pfaff, B., Garfinkel, T., and Rosenblum, M. Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation. In *Proceedings of the 14th USENIX Security Symposium* (2005).

[21] Criswell, J., Dautenhahn, N., and Adve, V. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *Proceedings of the 35th IEEE Symposium on Security and Privacy* (2014).

[22] De Moura, L., and Bjørner, N. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008).

[23] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (2010).

[24] Felmetsger, V., Cavedon, L., Kruegel, C., and Vigna, G. Toward Automated Detection of Logic Vulnerabilities in Web Applications. In *Proceedings of the 19th USENIX Security Symposium* (2010).

[25] Godefroid, P., Levin, M. Y., and Molnar, D. A. Automated whitebox fuzz testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium* (2008), Internet Society.

[26] Godefroid, P., and Taly, A. Automated Synthesis of Symbolic Instruction Encodings from I/O Samples. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (2012).

[27] Jim, T., Morrisett, J. G., Grossman, D., Hicks, M. W., Cheney, J., and Wang, Y. Cyclone: A Safe Dialect of C. In *Proceedings of the USENIX Annual Technical Conference* (2002).

[28] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (2005).

[29] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 30th ACM SIG-PLAN Conference on Programming Language Design and Implementation* (2009).

[30] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 9th International Symposium on Memory Management* (2010).

[31] Necula, G. C., McPeak, S., and Weimer, W. CCured: Type-safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2002).

[32] Niu, B., and Tan, G. Modular Control-flow Integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014).

[33] PaX Team. PaX Address Space Layout Randomization (ASLR). `http://pax.grsecurity.net/docs/aslr.txt`, 2003.

[34] Payer, M., and Gross, T. R. String Oriented Programming: When ASLR is Not Enough. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop* (2013).

[35] Serna, F. J. The Info Leak Era on Software Exploitation. *Black Hat USA* (2012).

[36] Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, U., Lozano, L., and Pike, G. Enforcing Forward-edge Control-flow Integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium* (2014).

[37] Ubuntu. List of Programs Built with PIE, May 2012. `https://wiki.ubuntu.com/Security/Features#pie`.

[38] Wang, Z., and Jiang, X. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Proceedings of the 31st IEEE Symposium on Security and Privacy* (2010).

[39] Xu, W., Bhatkar, S., and Sekar, R. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proceedings of the 15th USENIX Security Symposium* (2006).

[40] Yip, A., Wang, X., Zeldovich, N., and Kaashoek, M. F. Improving Application Security with Data Flow Assertions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (2009).

[41] Zeng, B., Tan, G., and Erlingsson, U. Strato: A Retargetable Framework for Low-level Inlined-reference Monitors. In *Proceedings of the 22nd USENIX Security Symposium* (2013).

[42] Zeng, B., Tan, G., and Morrisett, G. Combining Control-Flow Integrity and Static Analysis for Efficient and Validated Data Sandboxing. In *Proceedings of the 18th ACM conference on Computer and Communications Security* (2011).

[43] Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., and Zou, W. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy* (2013).

[44] Zhang, M., and Sekar, R. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Security Symposium* (2013).

Protocol state fuzzing of TLS implementations

Joeri de Ruiter
School of Computer Science
University of Birmingham

Erik Poll
Institute for Computing and Information Science
Radboud University Nijmegen

Abstract

We describe a largely automated and systematic analysis of TLS implementations by what we call 'protocol state fuzzing': we use state machine learning to infer state machines from protocol implementations, using only black-box testing, and then inspect the inferred state machines to look for spurious behaviour which might be an indication of flaws in the program logic. For detecting the presence of spurious behaviour the approach is almost fully automatic: we automatically obtain state machines and any spurious behaviour is then trivial to see. Detecting whether the spurious behaviour introduces exploitable security weaknesses does require manual investigation. Still, we take the point of view that any spurious functionality in a security protocol implementation is dangerous and should be removed.

We analysed both server- and client-side implementations with a test harness that supports several key exchange algorithms and the option of client certificate authentication. We show that this approach can catch an interesting class of implementation flaws that is apparently common in security protocol implementations: in three of the TLS implementations analysed new security flaws were found (in GnuTLS, the Java Secure Socket Extension, and OpenSSL). This shows that protocol state fuzzing is a useful technique to systematically analyse security protocol implementations. As our analysis of different TLS implementations resulted in different and unique state machines for each one, the technique can also be used for fingerprinting TLS implementations.

1 Introduction

TLS, short for Transport Layer Security, is widely used to secure network connections, for example in HTTPS. Being one of the most widely used security protocols, TLS has been the subject of a lot of research and many issues have been identified. These range from cryptographic attacks (such as problems when using RC4 [4]) to serious implementation bugs (such as Heartbleed [13]) and timing attacks (for example, Lucky Thirteen and variations of the Bleichenbacher attack [3, 30, 9]).

To describe TLS, or protocols in general, a state machine can be used to specify possible sequences of messages that can be sent and received. Using automated learning techniques, it is possible to automatically extract these state machines from protocol implementations, relying only on black-box testing. In essence, this involves fuzzing different sequences of messages, which is why we call this approach *protocol state fuzzing*. By analysing these state machines, logical flaws in the protocol flow can be discovered. An example of such a flaw is accepting and processing a message to perform some security-sensitive action *before* authentication takes place. The analysis of the state machines can be done by hand or using a model checker; for the analyses discussed in this paper we simply relied on manual analysis. Both approaches require knowledge of the protocol to interpret the results or specify the requirements. However, in security protocols, every superfluous state or transition is undesirable and a reason for closer inspection. The presence of such superfluous states or transitions is typically easy to spot visually.

1.1 Related work on TLS

Various formal methods have been used to analyse different parts and properties of the TLS protocol [33, 16, 22, 32, 20, 31, 26, 24, 28]. However, these analyses look at abstract descriptions of TLS, not actual implementations, and in practice many security problems with TLS have been due to mistakes in implementation [29]. To bridge the gap between the specification and implementation, formally verified TLS implementations have been proposed [7, 8].

Existing tools to analyse TLS implementations mainly focus on fuzzing of individual messages, in particular the certificates that are used. These certificates have been the source of numerous security problems in the past. An automated approach to test for vulnerabilities in the processing of certificates is using Frankencerts as proposed by Brubaker et al. [10] or using the tool x509test[1]. Fuzzing of individual messages is orthogonal to the technique we propose as it targets different parts or aspects of the code. However, the results of our analysis could be used to guide fuzzing of messages by indicating protocol states that might be interesting places to start fuzzing messages.

Another category of tools analyses implementations by looking at the particular configuration that is used. Examples of this are the SSL Server Test[2] and sslmap[3].

Finally, closely related research on the implementation of state machines for TLS was done by Beurdouche et al. [6]. We compare their work with ours in Section 5.

1.2 Related work on state machine learning

When learning state machines, we can distinguish between a passive and active approach. In passive learning, only existing data is used and based on this a model is constructed. For example, in [14] passive learning techniques are used on observed network traffic to infer a state machine of the protocol used by a botnet. This approach has been combined with the automated learning of message formats in [23], which then also used the model obtained as a basis for fuzz-testing.

When using active automated learning techniques, as done in this paper, an implementation is actively queried by the learning algorithm and based on the responses a model is constructed. We have used this approach before to analyse implementations of security protocols in EMV bank cards [1] and handheld readers for online banking [11], and colleagues have used it to analyse electronic passports [2]. These investigations did not reveal new security vulnerabilities, but they did provide interesting insights in the implementations analysed. In particular, it showed a lot of variation in implementations of bank cards [1] – even cards implementing the same MasterCard standard – and a known attack was confirmed for the online banking device and confirmed to be fixed in a new version [11].

1.3 Overview

We first discuss the TLS protocol in more detail in Section 2. Next we present our setup for the automated learning in Section 3. The results of our analysis of nine TLS implementations are subsequently discussed in Section 4, after which we conclude in Section 5.

2 The TLS protocol

The TLS protocol was originally known as SSL (Secure Socket Layer), which was developed at Netscape. SSL 1.0 was never released and version 2.0 contained numerous security flaws [37]. This lead to the development of SSL 3.0, on which all later versions are based. After SSL 3.0, the name was changed to TLS and currently three versions are published: 1.0, 1.1 and 1.2 [17, 18, 19]. The specifications for these versions are published in RFCs issued by the Internet Engineering Task Force (IETF).

To establish a secure connection, different subprotocols are used within TLS:

- The *Handshake* protocol is used to establish session keys and parameters and to optionally authenticate the server and/or client.
- The *ChangeCipherSpec* protocol – consisting of only one message – is used to indicate the start of the use of established session keys.
- To indicate errors or notifications, the *Alert* protocol is used to send the level of the alert (either warning or fatal) and a one byte description.

In Fig. 1 a normal flow for a TLS session is given. In the ClientHello message, the client indicates the desired TLS version, supported cipher suites and optional extensions. A cipher suite is a combination of algorithms used for the key exchange, encryption, and MAC computation. During the key exchange a premaster secret is established. This premaster secret is used in combination with random values from both the client and server to derive the master secret. This master secret is then used to derive the actual keys that are used for encryption and MAC computation. Different keys are used for messages from the client to the server and for messages in the opposite direction. Optionally, the key exchange can be followed by client verification where the client proves it knows the private key corresponding to the public key in the certificate it presents to the server. After the key exchange and optional client verification, a ChangeCipherSpec message is used to indicate that from that point on the agreed keys will be used to encrypt all messages and add a MAC to them. The Finished message is finally used to conclude the handshake phase. It contains a keyed hash, computed using the master secret, of all previously exchanged handshake messages. Since it is sent after the ChangeCipherSpec message it is the first message that is encrypted and MACed. After the handshake phase, application data can be exchanged over the established secure channel.

[1] `https://github.com/yymax/x509test`

[2] `https://www.ssllabs.com/ssltest/`

[3] `https://www.thesprawl.org/projects/sslmap/`

To add additional functionality, TLS offers the possibility to add extensions to the protocol. One example of such an extension is the – due to Heartbleed [13] by now well-known – Heartbeat Extension, which can be used to keep a connection alive using HeartbeatRequest and HeartbeatResponse messages [36].

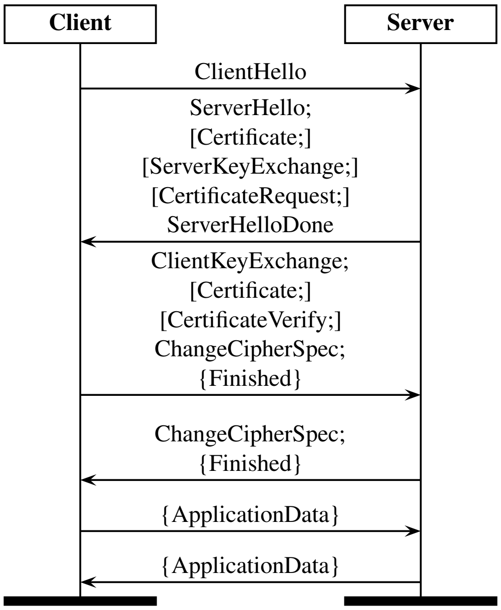


Figure 1: A regular TLS session. An encrypted message m is denoted as {m}. If message m is optional, this is indicated by [m].

3 State machine learning

To infer the state machines of implementations of the TLS protocol we used LearnLib [34], which uses a modified version of Angluin's L* algorithm [5]. An implementation that is analysed is referred to as the *System Under Test (SUT)* and is considered to be a black box. LearnLib has to be provided with a list of messages it can send to the SUT (also known as the *input alphabet*), and a command to reset the SUT to its initial state. A test harness is needed to translate abstract messages from the input alphabet to concrete messages that can be sent to the SUT. To be able to implement this test harness we need to know the messages that are used by the SUT. By sending sequences of messages and reset commands, LearnLib tries to come up with hypotheses for the state machine based on the responses it receives from the SUT. Such hypotheses are then checked for equivalence with the actual state machine. If the models are not equivalent, a counter-example is returned and LearnLib will use this to redefine its hypothesis.

As the actual state machine is not known, the equivalence check has to be approximated, with what is effectively a form of model-based testing. For this we use an improved version of Chow's W-method [12]. The W-method is guaranteed to be correct given an upper bound for the number of states. For LearnLib we can specify a depth for the equivalence checking: given a hypothesis for the state machine, the upper bound for the W-method is set to the number of found states plus the specified depth. The algorithm will only look for counterexample traces of which the lengths is at most the set upper bound, and if none can be found the current hypothesis for the state machine is assumed to be equivalent with the one implemented. This assumption is correct if the actual state machine does not have more states than the number of found states plus the specified depth. The W-method is very powerful but comes at a high cost in terms of performance. Therefore we improved the algorithm to take advantage of a property of the system we learn, namely that once a connection is closed, all outputs returned afterwards will be the same (namely *Connection closed*). So when looking for counterexamples, extending a trial trace that results in the connection being closed is pointless. The W-method, however, will still look for counterexamples by extending traces which result in a closed connection. We improved the W-method by adding a check to see if it makes sense to continue searching for counterexamples with a particular prefix, and for this we simply check if the connection has not been closed. This simple modification of the W-method greatly reduced the number of equivalence queries needed, as we will see in Section 4.

3.1 Test harness

To use LearnLib, we need to fix an input alphabet of messages that can be sent to the SUT. This alphabet is an abstraction of the actual messages sent. In our analyses we use different input alphabets depending on whether we test a client or server, and whether we perform a more limited or more extensive analysis. To test servers we support the following messages: ClientHello (RSA and DHE), Certificate (RSA and empty), ClientKeyExchange, ClientCertificateVerify, ChangeCipherSpec, Finished, ApplicationData (regular and empty), HeartbeatRequest and HeartbeatResponse. To test clients we support the following messages: ServerHello (RSA and DHE), Certificate (RSA and empty), CertificateRequest, ServerKeyExchange, ServerHelloDone, ChangeCipherSpec, Finished, ApplicationData (regular and empty), HeartbeatRequest and HeartbeatResponse.

We thus support all regular TLS messages as well as the messages for the Heartbeat Extension. The test harness supports both TLS version 1.2 and, in order to test older implementations, version 1.0. The input alphabet is not fixed, but can be configured per analysis as desired. For the output alphabet we use all the regular TLS messages as well as the messages from the Alert protocol that can be returned. This is extended with some special symbols that correspond with exceptions that can occur in the test harness:

- *Empty*, this is returned if no data is received from the SUT before a timeout occurs in the test harness.
- *Decryption failed*, this is returned if decryption fails in the test harness after a ChangeChipherSpec message was received. This could happen, for example, if not enough data is received, the padding is incorrect after decryption (e.g. because a different key was used for encryption) or the MAC verification fails.
- *Connection closed*, this is returned if a socket exception occurs or the socket is closed.

LearnLib uses these abstract inputs and outputs as labels on the transitions of the state machine. To interact with an actual TLS server or client we need a test harness that translates the abstract input messages to actual TLS packets and the responses back to abstract responses. As we make use of cryptographic operations in the protocol, we needed to introduce state in our test harness, for instance to keep track of the information used in the key exchange and the actual keys that result from this. Apart from this, the test harness also has to remember whether a ChangeCipherSpec was received or sent, as we have to encrypt and MAC all corresponding data after this message. Note that we only need a single test harness for TLS to then be able to analyse any implementation. Our test harness can be considered a 'stateless' TLS implementation.

When testing a server, the test harness is initialised by sending a ClientHello message to the SUT to retrieve the server's public key and preferred ciphersuite. When a reset command is received we set the internal variables to these values. This is done to prevent null pointer exceptions that could otherwise occur when messages are sent in the wrong order.

After sending a message the test harness waits to receive responses from the SUT. As the SUT will not always send a response, for example because it may be waiting for a next message, the test harness will generate a timeout after a fixed period. Some implementations require longer timeouts as they can be slower in responding. As the timeout has a significant impact on the total running time we varied this per implementation.

To test client implementations we need to launch a client for every test sequence. This is done automatically by the test harness upon receiving the *reset* command. The test harness then waits to receive the ClientHello message, after which the client is ready to receive a query. Because the first ClientHello is received before any query is issued, this message does not appear explicitly in the learned models.

4 Results

We analysed the nine different implementations listed in Table 1. We used demo client and server applications that came with the different implementations except with the Java Secure Socket Extension (JSSE). For JSSE we wrote simple server and client applications. For the implementations listed the models of the server-side were learned using our modified W-method for the following alphabet: ClientHello (RSA), Certificate (empty), ClientKeyExchange, ChangeCipherSpec, Finished, ApplicationData (regular and empty), HeartbeatRequest. For completeness we learned models for both TLS version 1.0 and 1.2, when available, but this always resulted in the same model.

Due to space limitations we cannot include the models for all nine implementations in this paper, but we do include the models in which we found security issues (for GnuTLS, Java Secure Socket Extension, and OpenSSL), and the model of RSA BSAFE for Java to illustrate how much simpler the state machine can be. The other models can be found in [15] as well as online, together with the code of our test harness.[4] We wrote a Python application to automatically simplify the models by combining transitions with the same responses and replacing the abstract input and output symbols with more readable names. Table 2 shows the times needed to obtain these state machines, which ranged from about 9 minutes to over 8 hours.

A comparison between our modified equivalence algorithm and the original W-method can be found in Table 3. This comparison is based on the analysis of GnuTLS 3.3.12 running a TLS server. It is clear that by taking advantage of the state of the socket our algorithm performs much better than the original W-method: the number of equivalence queries is over 15 times smaller for our method when learning a model for the server.

When analysing a model, we first manually look if there are more paths than expected that lead to a successful exchange of application data. Next we determine whether the model contains more states than necessary and identify unexpected or superfluous transitions. We also check for transitions that can indicate interesting behaviour such as, for example, a 'Bad record MAC' alert or a *Decryption failed* message. If we come across any

[4] Available at `http://www.cs.bham.ac.uk/~deruitej/`

Name	Version	URL
GnuTLS	3.3.8 3.3.12	http://www.gnutls.org/
Java Secure Socket Extension (JSSE)	1.8.0_25 1.8.0_31	http://www.oracle.com/java/
mbed TLS (previously PolarSSL)	1.3.10	https://polarssl.org/
miTLS	0.1.3	http://www.mitls.org/
RSA BSAFE for C	4.0.4	http://www.emc.com/security/rsa-bsafe.htm
RSA BSAFE for Java	6.1.1	http://www.emc.com/security/rsa-bsafe.htm
Network Security Services (NSS)	3.17.4	https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS
OpenSSL	1.0.1g 1.0.1j 1.0.1l 1.0.2	https://www.openssl.org/
nqsb-TLS	0.4.0	https://github.com/mirleft/ocaml-tls

Table 1: Tested implementations

unexpected behaviour, we perform a more in-depth analysis to determine the cause and severity.

An obvious first observation is that all the models of server-side implementations are very different. For example, note the huge difference between the models learned for RSA BSAFE for Java in Fig. 6 and for OpenSSL in Fig. 7. Because all the models are different, they provide a unique fingerprint of each implementation, which could be used to remotely identify the implementation that a particular server is using.

Most demo applications close the connection after their first response to application data. In the models there is then only one ApplicationData transition where application data is exchanged instead of the expected cycle consisting of an ApplicationData transition that allows server and client to continue exchanging application data after a successful handshake.

In the subsections below we discuss the peculiarities of models we learned, and the flaws they revealed. Correct paths leading to an exchange of application data are indicated by thick green transitions in the models. If there is any additional path leading to the exchange of application data this is a security flaw and indicated by a dashed red transition.

4.1 GnuTLS

Fig. 2 shows the model that was learned for GnuTLS 3.3.8. In this model there are two paths leading to a successful exchange of application data: the regular one without client authentication and one where an empty client certificate is sent during the handshake. As we did not require client authentication, both are acceptable paths. What is immediately clear is that there are more states than expected. Closer inspection reveals that there is a ‘shadow’ path, which is entered by sending a HeartbeatRequest message during the handshake protocol. The handshake protocol then does proceed, but eventually results in a fatal alert (‘Internal error’) in response to the Finished message (from state *8*). From every state in the handshake protocol it is possible to go to a corresponding state in the ‘shadow’ path by sending the HeartbeatRequest message. This behaviour is introduced by a security bug, which we will discuss below. Additionally there is a redundant state *5*, which is reached from states *3* and *9* when a ClientHello message is sent. From state *5* a fatal alert is given to all subsequent messages that are sent. One would expect to already receive an error message in response to the ClientHello message itself.

Forgetting the buffer in a heartbeat As mentioned above, HeartbeatRequest messages are not just ignored in the handshake protocol but cause some side effect: sending a HeartbeatRequest during the handshake protocol will cause the implementation to return an alert message in response to the Finished message that terminates the handshake. Further inspection of the code revealed the cause: the implementation uses a buffer to collect all handshake messages in order to compute a hash over these messages when the handshake is completed, but this buffer is reset upon receiving the heartbeat message. The alert is then sent because the hashes computed by server and client no longer match.

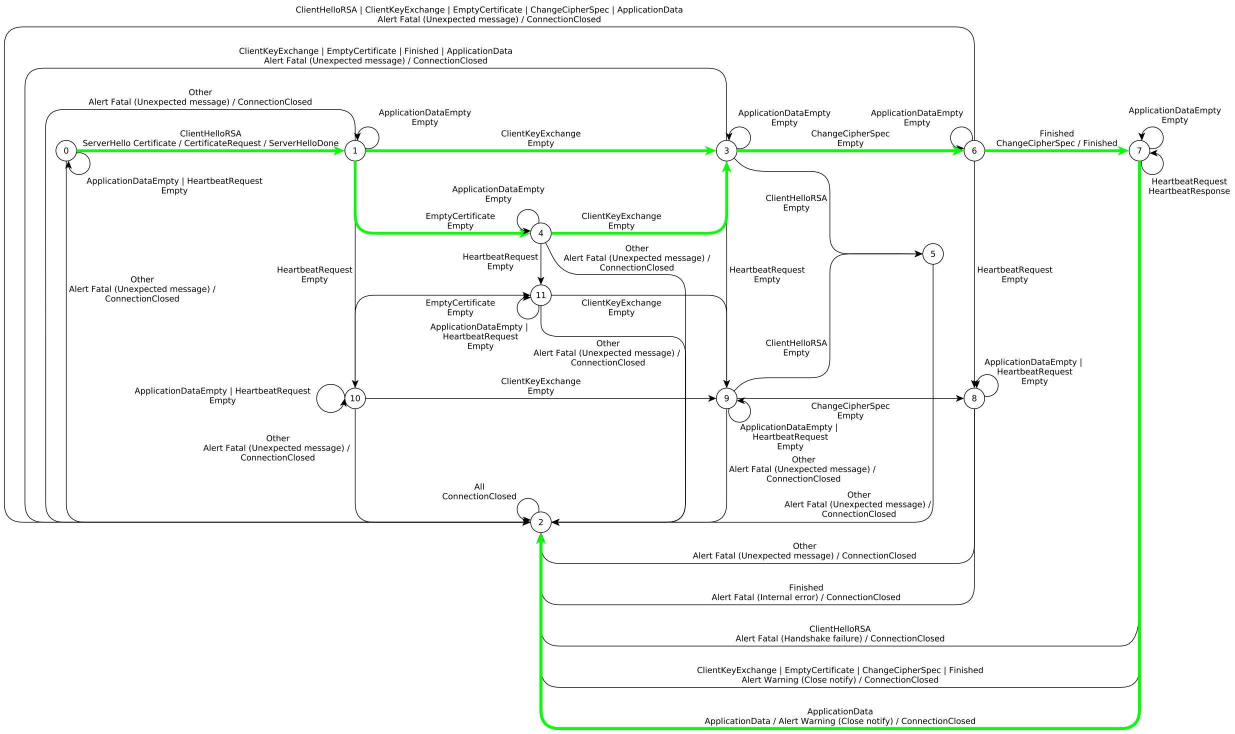

ClientHelloRSA | ClientKeyExchange | EmptyCertificate | ChangeCipherSpec | ApplicationData
Alert Fatal (Unexpected message) / ConnectionClosed
ClientKeyExchange | EmptyCertificate | Finished | ApplicationData
Alert Fatal (Unexpected message) / ConnectionClosed
Other
Alert Fatal (Unexpected message) / ConnectionClosed
ApplicationDataEmpty
Empty
ClientHelloRSA
ServerHello Certificate / CertificateRequest / ServerHelloDone
0
1
ClientKeyExchange
Empty
3
ApplicationDataEmpty
Empty
ApplicationDataEmpty
Empty
ChangeCipherSpec
Empty
6
Finished
ChangeCipherSpec / Finished
7
ApplicationDataEmpty
Empty
HeartbeatRequest
HeartbeatResponse
ApplicationDataEmpty | HeartbeatRequest
Empty
ApplicationDataEmpty
Empty
EmptyCertificate
Empty
4
ClientKeyExchange
Empty
ClientHelloRSA
Empty
5
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
HeartbeatRequest
Empty
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
HeartbeatRequest
Empty
HeartbeatRequest
Empty
HeartbeatRequest
Empty
11
EmptyCertificate
Empty
ClientKeyExchange
Empty
ApplicationDataEmpty |
HeartbeatRequest
Empty
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
ClientHelloRSA
Empty
ApplicationDataEmpty |
HeartbeatRequest
Empty
ApplicationDataEmpty | HeartbeatRequest
Empty
10
ClientKeyExchange
Empty
9
8
ChangeCipherSpec
Empty
ApplicationDataEmpty |
HeartbeatRequest
Empty
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
All
ConnectionClosed
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
2
Other
Alert Fatal (Unexpected message) / ConnectionClosed
Finished
Alert Fatal (Internal error) / ConnectionClosed
ClientHelloRSA
Alert Fatal (Handshake failure) / ConnectionClosed
ClientKeyExchange | EmptyCertificate | ChangeCipherSpec | Finished
Alert Warning (Close notify) / ConnectionClosed
ApplicationData
ApplicationData / Alert Warning (Close notify) / ConnectionClosed

Figure 2: Learned state machine model for GnuTLS 3.3.8

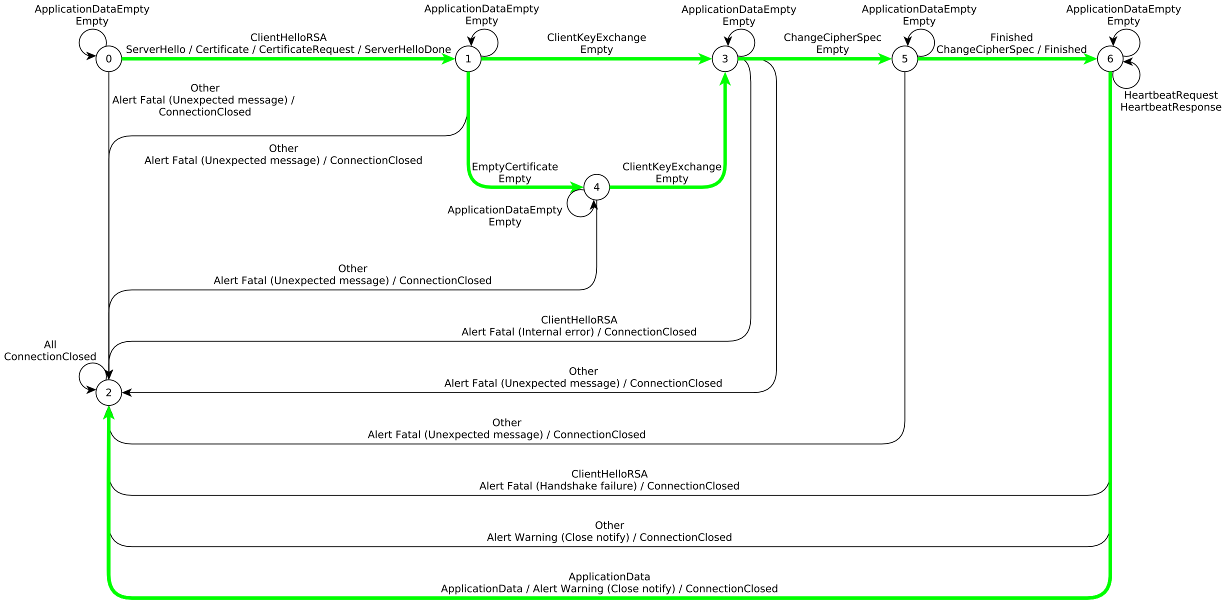

ApplicationDataEmpty
Empty
ClientHelloRSA
ServerHello / Certificate / CertificateRequest / ServerHelloDone
0
ApplicationDataEmpty
Empty
1
ClientKeyExchange
Empty
ApplicationDataEmpty
Empty
3
ChangeCipherSpec
Empty
ApplicationDataEmpty
Empty
5
Finished
ChangeCipherSpec / Finished
ApplicationDataEmpty
Empty
6
HeartbeatRequest
HeartbeatResponse
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
Other
Alert Fatal (Unexpected message) / ConnectionClosed
EmptyCertificate
Empty
4
ClientKeyExchange
Empty
ApplicationDataEmpty
Empty
Other
Alert Fatal (Unexpected message) / ConnectionClosed
ClientHelloRSA
Alert Fatal (Internal error) / ConnectionClosed
All
ConnectionClosed
2
Other
Alert Fatal (Unexpected message) / ConnectionClosed
Other
Alert Fatal (Unexpected message) / ConnectionClosed
ClientHelloRSA
Alert Fatal (Handshake failure) / ConnectionClosed
Other
Alert Warning (Close notify) / ConnectionClosed
ApplicationData
ApplicationData / Alert Warning (Close notify) / ConnectionClosed

Figure 3: Learned state machine model for GnuTLS 3.3.12. A comparison with the model for GnuTLS 3.3.8 in Fig. 2 shows that the superflous states (8, 9, 10, and 11) are now gone, confirming that the code has been improved.

	#states	Timeout	Time (h:mm)	#membership queries	#equivalence queries
GnuTLS 3.3.8	12	100ms	0:45	1370	5613
GnuTLS 3.3.12	7	100ms	0:09	456	1347
mbed TLS 1.3.10	8	100ms	0:39	520	2939
OpenSSL 1.0.1g [+]	16	100ms	0:31	1016	4171
OpenSSL 1.0.1j [+]	11	100ms	0:16	680	2348
OpenSSL 1.0.1l [+]	10	100ms	0:14	624	2249
OpenSSL 1.0.2 [+]	7	100ms	0:06	350	902
JSSE 1.8.0_25	9	200ms	0:41	584	2458
JSSE 1.8.0_31	9	200ms	0:39	584	2176
miTLS 0.1.3	6	1500ms	0:53	392	517
NSS 3.17.4	8	500ms	3:16	520	5329
RSA BSAFE for Java 6.1.1	6	500ms	0:18	392	517
RSA BSAFE for C 4.0.4	9	200ms	8:16	584	26353
nqsb-TLS 0.4.0 [+]	8	100ms	0:15	399	1835

[+] Without heartbeat extension

Table 2: Results of the automated analysis of server implementations for the regular alphabet of inputs using our modified W-method with depth 2

Alphabet	Algorithm	Time (hh:mm)	#states	Membership queries	Equivalence queries
regular	modified W-method	0:09	7	456	1347
full	modified W-method	0:27	9	1573	4126
full	original W-method	4:09	9	1573	68578

Table 3: Analysis of the GnuTLS 3.3.12 server using different alphabets and equivalence algorithms

This bug can be exploited to effectively bypass the integrity check that relies on comparing the keyed hashes of the messages in the handshake: when also resetting this buffer on the client side (i.e. our test harness) at the same time we were able to successfully complete the handshake protocol, but then no integrity guarantee is provided on the previous handshake messages that were exchanged.

By learning the state machine of a GnuTLS client we confirmed that the same problem exists when using GnuTLS as a client.

This problem was reported to the developers of GnuTLS and is fixed in version 3.3.9. By learning models of newer versions, we could confirm the issue is no longer present, as can be seen in Fig. 3.

To exploit this problem both sides would need to reset the buffer at the same time. This might be hard to achieve as at any time either one of the two parties is computing a response, at which point it will not process any incoming message. If an attacker would successfully succeed to exploit this issue no integrity would be provided on any message sent before, meaning a fallback attack would be possible, for example to an older TLS version or weaker cipher suite.

4.2 mbed TLS

For mbed TLS, previously known as PolarSSL, we tested version 1.3.10. We saw several paths leading to a successful exchange of data. Instead of sending a regular ApplicationData message, it is possible to first send one empty ApplicationData message after which it is still possible to send the regular ApplicationData message. Sending two empty ApplicationData messages directly after each other will close the connection. However, if in between these message an unexpected handshake message is sent, the connection will not be closed and only a warning is returned. After this it is also still possible to send a regular ApplicationData message. While this is strange behaviour, it does not seem to be exploitable.

4.3 Java Secure Socket Extension

For Java Secure Socket Extension we analysed Java version 1.8.0_25. The model contains several paths leading to a successful exchange of application data and contains more states than expected (see Fig. 4). This is the result of a security issue which we will discuss below.

As long as no Finished message has been sent it is apparently possible to keep renegotiating. After sending a ClientKeyExchange, other ClientHello messages are accepted as long as they are eventually followed by another ClientKeyExchange message. If no ClientKeyExchange message was sent since the last ChangeCipherSpec, a ChangeCipherSpec message will result in an error (state *7*). Otherwise it either leads to an error state if sent directly after a ClientHello (state *8*) or a successful change of keys after a ClientKeyExchange.

Accepting plaintext data More interesting is that the model contains *two* paths leading to the exchange of application data. One of these is a regular TLS protocol run, but in the second path the ChangeCipherSpec message from the client is omitted. Despite the server not receiving a ChangeCipherSpec message it still responds with a ChangeCipherSpec message to a plaintext Finished message by the client. As a result the server will send its data encrypted, but it expects data from the client to be unencrypted. A similar problem occurs when trying to negotiate new keys. By skipping the ChangeCipherSpec message and just sending the Finished message the server will start to use the new keys, whereas the client needs to continue to use its old keys.

This bug invalidates any assumption of integrity or confidentiality of data sent to the server, as it can be tricked into accepting plaintext data. To exploit this issue it is, for example, possible to include this behaviour in a rogue library. As the attack is transparent to applications using the connection, both the client and server application would think they talk on a secure connection, where in reality anyone on the line could read the client's data and tamper with it. Fig. 5 shows a protocol run where this bug is triggered. The bug was report to Oracle and is identified by CVE-2014-6593. A fix was released in their Critical Security Update in January 2015. By analysing JSSE version 1.8.0_31 we are able to confirm the issue was indeed fixed.

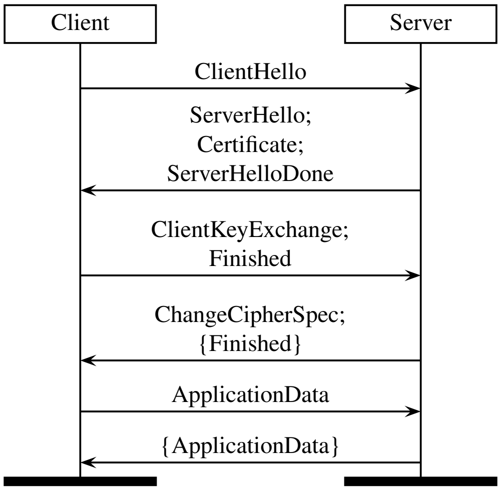


Figure 5: A protocol run triggering a bug in the JSSE, causing the server to accept plaintext application data.

This issue was identified in parallel by Beurdouche et al. [6], who also reported the same and a related issue for the client-side. By learning the client, we could confirm that the issue was also present there. Moreover, after receiving the ServerHello message, the client would accept the Finish message and start exchanging application data at any point during the handshake protocol. This makes it possible to completely circumvent both server authentication and the confidentiality and integrity of the data being exchanged.

4.4 miTLS

MiTLS is a formally verified TLS implementation written in F# [8]. For miTLS 0.1.3, initially our test harness had problems to successfully complete the handshake protocol and the responses seemed to be nondeterministic because sometimes a response was delayed and appeared to be received in return to the next message. To solve this, the timeout had to be increased considerably when waiting for incoming messages to not miss any message. This means that compared to the other implementations, miTLS was relatively slow in our setup. Additionally, miTLS requires the Secure Renegotiation extension to be enabled in the ClientHello message. The learned model looks very clean with only one path leading to an exchange of application data and does not contain more states than expected.

4.5 RSA BSAFE for C

The RSA BSAFE for C 4.0.4 library resulted in a model containing two paths leading to the exchange application data. The only difference between the paths is that an

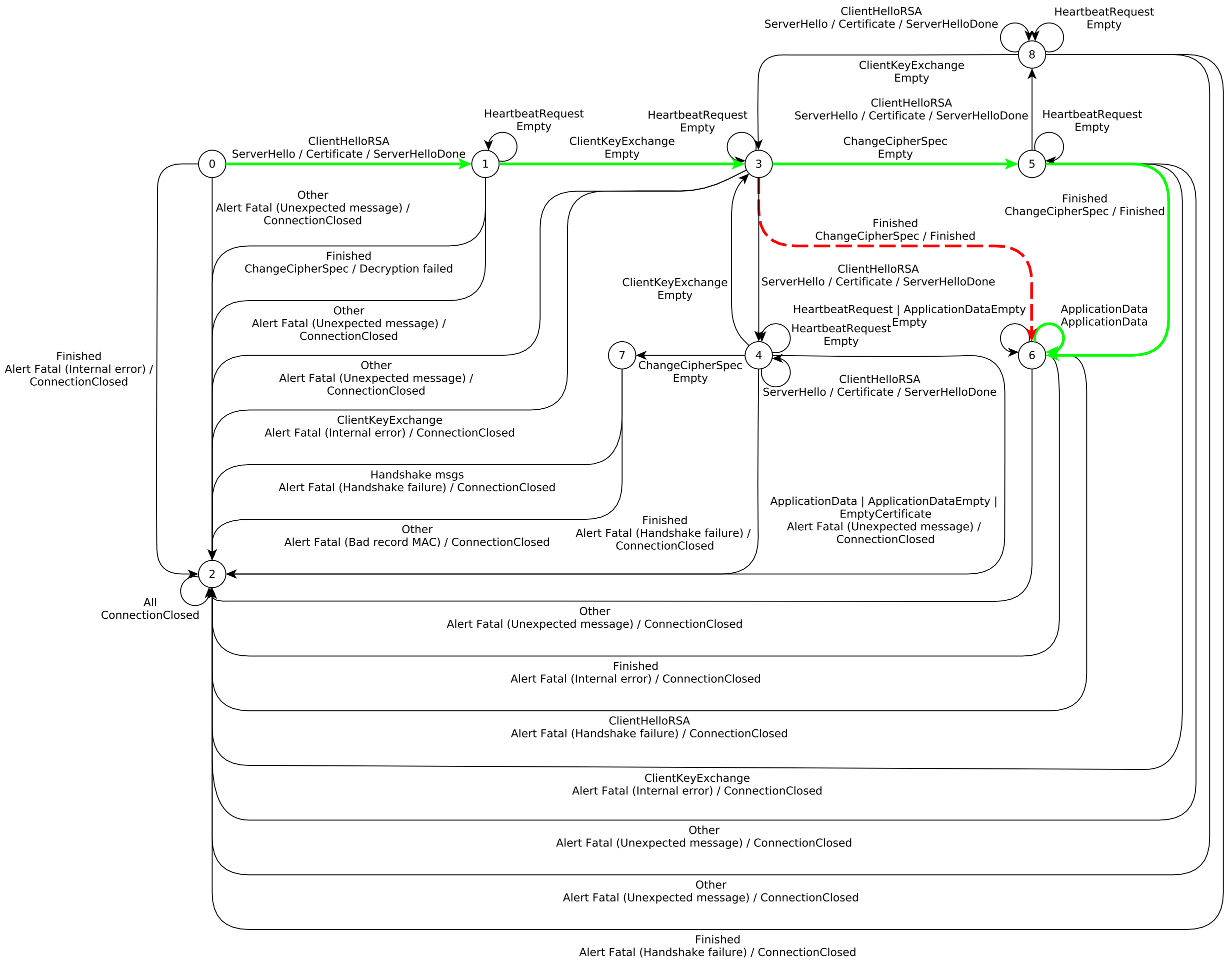

ClientHelloRSA
ServerHello / Certificate / ServerHelloDone
HeartbeatRequest
Empty
8
ClientKeyExchange
Empty
ClientHelloRSA
ServerHello / Certificate / ServerHelloDone
HeartbeatRequest
Empty
HeartbeatRequest
Empty
HeartbeatRequest
Empty
ClientHelloRSA
ServerHello / Certificate / ServerHelloDone
ClientKeyExchange
Empty
ChangeCipherSpec
Empty
0
1
3
5
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
Finished
ChangeCipherSpec / Finished
Finished
ChangeCipherSpec / Finished
Finished
ChangeCipherSpec / Decryption failed
ClientKeyExchange
Empty
ClientHelloRSA
ServerHello / Certificate / ServerHelloDone
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
HeartbeatRequest | ApplicationDataEmpty
Empty
ApplicationData
ApplicationData
HeartbeatRequest
Empty
Finished
Alert Fatal (Internal error) /
ConnectionClosed
7
4
6
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
ChangeCipherSpec
Empty
ClientHelloRSA
ServerHello / Certificate / ServerHelloDone
ClientKeyExchange
Alert Fatal (Internal error) / ConnectionClosed
Handshake msgs
Alert Fatal (Handshake failure) / ConnectionClosed
ApplicationData | ApplicationDataEmpty |
EmptyCertificate
Alert Fatal (Unexpected message) /
ConnectionClosed
Other
Alert Fatal (Bad record MAC) / ConnectionClosed
Finished
Alert Fatal (Handshake failure) /
ConnectionClosed
2
All
ConnectionClosed
Other
Alert Fatal (Unexpected message) / ConnectionClosed
Finished
Alert Fatal (Internal error) / ConnectionClosed
ClientHelloRSA
Alert Fatal (Handshake failure) / ConnectionClosed
ClientKeyExchange
Alert Fatal (Internal error) / ConnectionClosed
Other
Alert Fatal (Unexpected message) / ConnectionClosed
Other
Alert Fatal (Unexpected message) / ConnectionClosed
Finished
Alert Fatal (Handshake failure) / ConnectionClosed

Figure 4: Learned state machine model for JSSE 1.8.0_25

empty ApplicationData is sent in the second path. However, the alerts that are sent are not very consistent as they differ depending on the state and message. For example, sending a ChangeCipherSpec message after an initial ClientHello results in a fatal alert with reason ‘Illegal parameter’, whereas application data results in a fatal alert with ‘Unexpected message’ as reason. More curious however is a fatal alert ‘Bad record MAC’ that is returned to certain messages after the server received the ChangeCipherSpec in a regular handshake. As this alert is only returned in response to certain messages, while other messages are answered with an ‘Unexpected message’ alert, the server is apparently able to successfully decrypt and check the MAC on messages. Still, an error is returned that it is not able to do this. This seems to be a non-compliant usage of alert messages.

At the end of the protocol the implementation does not close the connection. This means we cannot take any advantage from a closed connection in our modified W-method and the analysis therefore takes much longer than for the other implementations.

4.6 RSA BSAFE for Java

The model for RSA BSAFE for Java 6.1.1 library looks very clean, as can be seen in Fig. 6. The model again contains only one path leading to an exchange of application data and no more states than necessary. In general all received alerts are ‘Unexpected message’. The only exception is when a ClientHello is sent after a successful handshake, in which case a ‘Handshake failure’ is given. This makes sense as the ClientHello message is not correctly formatted for secure renegotiation, which is required in this case. This model is the simplest that we learned during our research.

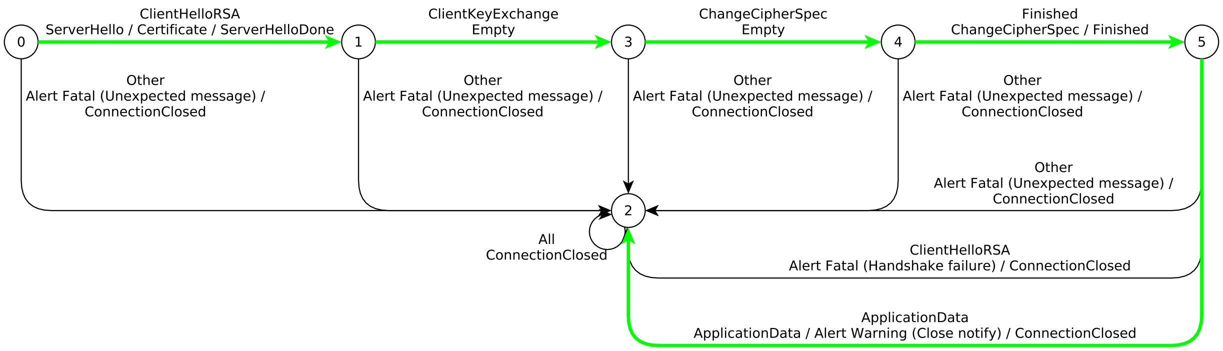

ClientHelloRSA
ServerHello / Certificate / ServerHelloDone
ClientKeyExchange
Empty
ChangeCipherSpec
Empty
Finished
ChangeCipherSpec / Finished
0
1
3
4
5
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
2
All
ConnectionClosed
ClientHelloRSA
Alert Fatal (Handshake failure) / ConnectionClosed
ApplicationData
ApplicationData / Alert Warning (Close notify) / ConnectionClosed

Figure 6: Learned state machine model for RSA BSAFE for Java 6.1.1

4.7 Network Security Services

The model for NSS that was learned for version 3.17.4 looks pretty clean, although there is one more state than one would expect. There is only one path leading to a successful exchange of application data. In general all messages received in states where they are not expected are responded to with a fatal alert ('Unexpected message'). Exceptions to this are the Finished and Heartbeat messages: these are ignored and the connection is closed without any alert. Other exceptions are non-handshake messages sent before the first ClientHello: then the server goes into a state where the connection stays open but nothing happens anymore. Although the TLS specification does not explicitly specify what to do in this case, one would expect the connection to be closed, especially since it's not possible to recover from this. Because the connection is not actually closed in this case the analysis takes longer, as we have less advantage of our modification of the W-method to decide equivalence.

4.8 OpenSSL

Fig. 7 shows the model inferred for OpenSSL 1.01j. In the first run of the analysis it turned out that HeartbeatRequest message sent during the handshake phase were 'saved up' and only responded to after the handshake phase was finished. As this results in infinite models we had to remove the heartbeat messages from the input alphabet. This model obtained contains quite a few more states than expected, but does only contain one path to successfully exchange application data.

The model shows that it is possible to start by sending two ClientHello messages, but not more. After the second ClientHello message there is no path to a successful exchange of application data in the model. This is due to the fact that OpenSSL resets the buffer containing the handshake messages every time when sending a ClientHello, whereas our test harness does this only on initialisation of the connection. Therefore, the hash computed by our test harness at the end of the handshake is not accepted and the Finished message in state *9* is responded to with an alert. Which messages are included in the hash differs per implementation: for JSSE all handshake messages since the beginning of the connection are included.

Re-using keys In state *8* we see some unexpected behaviour. After successfully completing a handshake, it is possible to send an additional ChangeCipherSpec message after which all messages are responded to with a 'Bad record MAC' alert. This usually is an indication of wrong keys being used. Closer inspection revealed that at this point OpenSSL changes the keys that the client uses to encrypt and MAC messages to the server keys. This means that in both directions the same keys are used from this point.

We observed the following behaviour after the additional ChangeCipherSpec message. First, OpenSSL expects a ClientHello message (instead of a Finished message as one would expect). This ClientHello is responded to with the ServerHello, ChangeCipherSpec and Finished messages. OpenSSL does change the server keys then, but does not use the new randoms from the ClientHello and ServerHello to compute new keys. Instead the old keys are used and the cipher is thus basically reset (i.e. the original IVs are set and the MAC counter reset to 0). After receiving the ClientHello message, the server does expect the Finished message, which contains the keyed hash over the messages since the second ClientHello and does make use of the new client and server randoms. After this, application data can be send over the connection, where the same keys are used in both directions. The issue was reported to the OpenSSL team and was fixed in version 1.0.1k.

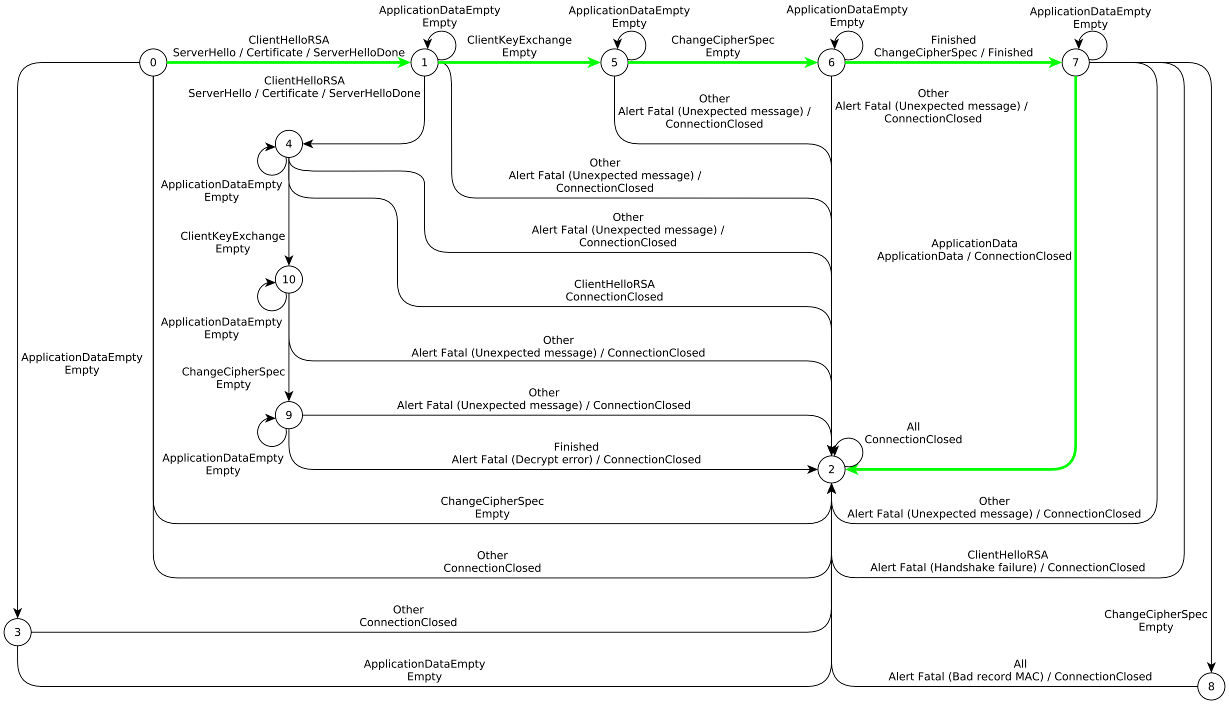

0
1
2
3
4
5
6
7
8
9
10
ApplicationDataEmpty
Empty
ClientHelloRSA
ServerHello / Certificate / ServerHelloDone
ClientKeyExchange
Empty
ChangeCipherSpec
Empty
Finished
ChangeCipherSpec / Finished
ClientHelloRSA
ServerHello / Certificate / ServerHelloDone
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
ApplicationData
ApplicationData / ConnectionClosed
ClientHelloRSA
ConnectionClosed
ClientKeyExchange
Empty
Other
Alert Fatal (Unexpected message) / ConnectionClosed
ChangeCipherSpec
Empty
Other
Alert Fatal (Unexpected message) / ConnectionClosed
All
ConnectionClosed
Finished
Alert Fatal (Decrypt error) / ConnectionClosed
ChangeCipherSpec
Empty
Other
Alert Fatal (Unexpected message) / ConnectionClosed
Other
ConnectionClosed
ClientHelloRSA
Alert Fatal (Handshake failure) / ConnectionClosed
Other
ConnectionClosed
ChangeCipherSpec
Empty
ApplicationDataEmpty
Empty
All
Alert Fatal (Bad record MAC) / ConnectionClosed

Figure 7: Learned state machine model for OpenSSL 1.0.1j

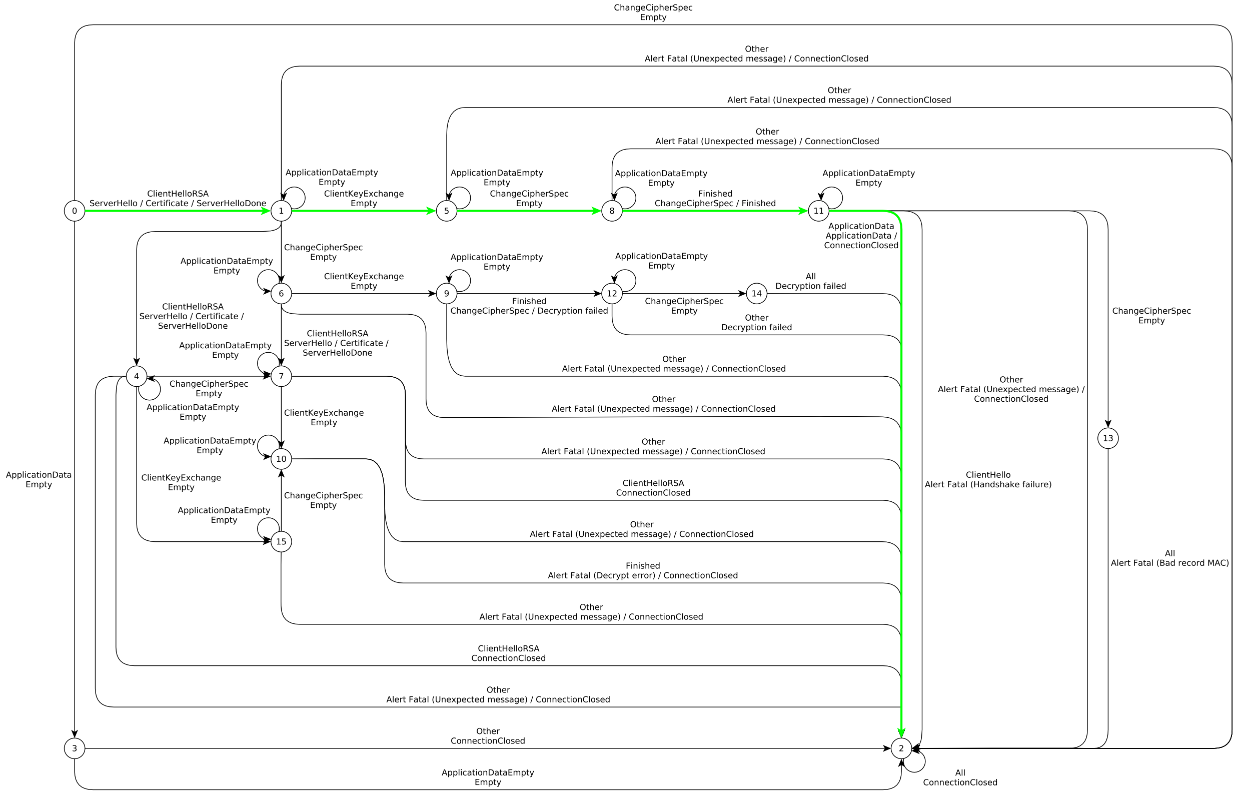

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
ChangeCipherSpec
Empty
Other
Alert Fatal (Unexpected message) / ConnectionClosed
Other
Alert Fatal (Unexpected message) / ConnectionClosed
Other
Alert Fatal (Unexpected message) / ConnectionClosed
ClientHelloRSA
ServerHello / Certificate / ServerHelloDone
ApplicationDataEmpty
Empty
ClientKeyExchange
Empty
ChangeCipherSpec
Empty
Finished
ChangeCipherSpec / Finished
ApplicationData
ApplicationData /
ConnectionClosed
ChangeCipherSpec
Empty
ClientKeyExchange
Empty
Finished
ChangeCipherSpec / Decryption failed
ChangeCipherSpec
Empty
All
Decryption failed
Other
Decryption failed
ClientHelloRSA
ServerHello / Certificate /
ServerHelloDone
Other
Alert Fatal (Unexpected message) / ConnectionClosed
Other
Alert Fatal (Unexpected message) / ConnectionClosed
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
ClientKeyExchange
Empty
ChangeCipherSpec
Empty
ApplicationData
Empty
ClientHelloRSA
ConnectionClosed
ClientHello
Alert Fatal (Handshake failure)
Other
Alert Fatal (Unexpected message) / ConnectionClosed
Finished
Alert Fatal (Decrypt error) / ConnectionClosed
All
Alert Fatal (Bad record MAC)
Other
Alert Fatal (Unexpected message) / ConnectionClosed
ClientHelloRSA
ConnectionClosed
Other
Alert Fatal (Unexpected message) / ConnectionClosed
Other
ConnectionClosed
ApplicationDataEmpty
Empty
All
ConnectionClosed

Figure 8: Learned state machine model for OpenSSL 1.0.1g, an older version of OpenSSL which had a known security flaw [27].

Early ChangeCipherSpec The state machine model of the older version OpenSSL 1.0.1g (Fig. 8) reveals a known vulnerability that was recently discovered [27], which makes it possible for an attacker to easily compute the session keys that are used in the versions up to 1.0.0l and 1.0.1g, as described below.

As soon as a ChangeCipherSpec message is received, the keys are computed. However, this also happened when no ClientKeyExchange was sent yet, in which case an empty master secret is used. This results in keys that are computed based on only public data. In version 1.0.1 it is possible to completely hijack a session by sending an early ChangeCipherSpec message to both the server and client, as in this version the empty master secret is also used in the computation of the hash in the Finished message. In the model of OpenSSL version 1.0.1g in Fig. 8 it is clear that if a ChangeCipherSpec message is received too early, the Finished message is still accepted as a ChangeCipherSpec is returned (see path *0, 1, 6, 9, 12* in the model). This is an indication of the bug and would be reason for closer inspection. The incoming messages after this path cannot be decrypted anymore however, because the corresponding keys are only computed by our test harness as soon as the ChangeCipherSpec message is received, which means that these keys are actually based on the ClientKeyExchange message. A simple modification of the test harness to change the point at which the keys are computed will even provide a successful exploitation of the bug.

An interesting observation regarding the evolution of the OpenSSL code is that for the four different versions that we analysed (1.0.1g, 1.0.1j, 1.0.1l and 1.0.2) the number of states reduces with every version. For version 1.0.2 there is still one state more than required, but this is an error state from which all messages result in a closed connection.

4.9 nqsb-TLS

A recent TLS implementation, nqsb-TLS, is intended to be both a specification and usable implementation written in OCaml [25]. For nsqb-TLS we analysed version 0.4.0. Our analysis revealed a bug in this implementation: alert messages are not encrypted even after a ChangeCipherSpec is received. This bug was reported to the nqsb-TLS developers and is fixed in a newer version. What is more interesting is a design decision with regard to the state machine: after the client sends a ChangeCipherSpec, the server immediately responds with a ChangeCipherSpec. This is different compared to all other implementations, that first wait for the client to also send a Finished message before sending a response. This is a clear example where the TLS specifications are not completely unambiguous and adding a state machine would remove room for interpretation.

5 Conclusion

We presented a thorough analysis of commonly used TLS implementations using the systematic approach we call protocol state fuzzing: we use state machine learning, which relies only on black box testing, to infer a state machine and then we perform a manual analysis of the state machines obtained. We demonstrated that this is a powerful and fast technique to reveal security flaws: in 3 out of 9 tested implementations we discovered new flaws. We applied the method on both server- and client-side implementations. By using our modified version of the W-method we are able to drastically reduce the number of equivalence queries used, which in turn results in a much lower running time of the analysis.

Our approach is able to find mistakes in the logic in the state machine of implementations. Deliberate backdoors, that are for example triggered by sending a particular message 100 times, would not be detected. Also mistakes in, for example, the parsing of messages or certificates would not be detected.

An overview of different approaches to prevent security bugs and more generally improve the security of software is given in [38] (using the Heartbleed bug as a basis). The method presented in this paper would not have detected the Heartbleed bug, but we believe it makes a useful addition to the approaches discussed in [38]. It is related to some of the approaches listed there; in particular, state machine learning involves a form of negative testing: the tests carried out during the state machine learning include many negative tests, namely those where messages are sent in unexpected orders, which one would expect to result in the closing of the connection (and which probably *should* result in closing of the connection, to be on the safe side). By sending messages in an unexpected order we get a high coverage of the code, which is different from for example full branch code coverage, as we trigger many different paths through the code.

In parallel with our research Beurdouche et al. [6] independently performed closely related research. They also analyse protocol state machines of TLS implementations and successfully find numerous security flaws. Both approaches have independently come up with the same fundamental idea, namely that protocol state machines are a great formalism to systematically analyse implementations of security protocols. Both approaches require the construction of a framework to send arbitrary TLS messages, and both approaches reveal that OpenSSL and JSSE have the most (over)complicated state machines.

The approach of Beurdouche et al. is different though: whereas we infer the state machines from the code without prior knowledge, they start with a manually constructed reference protocol state machine, and subsequently use this as a basis to test TLS implementations. Moreover, the testing they do here is not truly random, as the 'blind' learning by LearnLib is, but uses a set of test traces that is automatically generated using some heuristics.

The difference in the issues identified by Beurdouche et al. and us can partly be explained by the difference in functionality that is supported by the test frameworks used. For example, our framework supports the Heartbeat extension, whereas theirs supports Diffie-Hellman certificates and export cipher suites. Another reason is the fact that our approach has a higher coverage due to its 'blind' nature.

One advantage of our approach is that we don't have to construct a correct reference model by hand beforehand. But in the end, we do have to decide which behaviour is unwanted. Having a visual model helps here, as it is easy to see if there are states or transitions that seem redundant and don't occur in other models. Note that both approaches ultimately rely on a manual analysis to assess the security impact of any protocol behaviour that is deemed to be deviant or superfluous.

When it comes to implementing TLS, the specifications leave the developer quite some freedom as how to implement the protocol, especially in handling errors or exceptions. Indeed, many of the differences between models we infer are variations in error messages. These are not fixed in the specifications and can be freely chosen when implementing the protocol. Though this might be useful for debugging, the different error messages are probably not useful in production (especially since they differ per implementation).

This means that there is not a single 'correct' state machine for the TLS protocol and indeed every implementation we analysed resulted in a different model. However, there are some clearly wrong state machines. One would expect to see a state machine where there is clearly one correct path (or possibly more depending on the configuration) and all other paths going to one error state – preferably all with the same error code. We have seen one model that conforms to this, namely the one for RSA BSAFE for Java, shown in Fig. 6.

Of course, it would be interesting to apply the same technique we have used on TLS implementations here on implementations of other security protocols. The main effort in protocol state fuzzing is developing a test harness. But as only one test harness is needed to test all implementations for a given protocol, we believe that this is a worthwhile investment. In fact, one can argue that for any security protocol such a test harness should be provided to allow analysis of implementations.

The first manual analysis of the state machines we obtain is fairly straightforward: any superfluous strange behaviour is easy to spot visually. This step could even be automated as well by providing a correct reference state machine. A state machine that we consider to be correct would be the one that we learned for RSA BSAFE for Java.

Deciding whether any superfluous behaviour is exploitable is the hardest part of the manual analysis, but for security protocols it makes sense to simply require that there should not be any superfluous behaviour whatsoever.

The difference behaviour between the various implementations might be traced back to Postel's Law:

> 'Be conservative in what you send,
> be liberal in what you accept.'

As has been noted many times before, e.g. in [35], this is an unwanted and risky approach in security protocols: if there is any suspicion about inputs they should be discarded, connections should be closed, and no response should be given that could possibly aid an attacker. To quote [21]: 'It's time to deprecate Jon Postel's dictum and to be conservative in what you accept'.

Of course, ideally state machines would be included in the official specifications of protocols to begin with. This would provide a more fundamental solution to remove – or at least reduce – some of the implementation freedom. It would avoid each implementer having to come up with his or her own interpretation of English prose specifications, avoiding not only lots of work, but also the large variety of state machines in implementations that we observed, and the bugs that some of these introduce.

References

[1] AARTS, F., DE RUITER, J., AND POLL, E. Formal models of bank cards for free. In *Software Testing Verification and Validation Workshop, IEEE International Conference on* (2013), IEEE, pp. 461–468.

[2] AARTS, F., SCHMALTZ, J., AND VAANDRAGER, F. Inference and abstraction of the biometric passport. In *Leveraging Applications of Formal Methods, Verification, and Validation*, T. Margaria and B. Steffen, Eds., vol. 6415 of *Lecture Notes in Computer Science*. Springer, 2010, pp. 673–686.

[3] AL FARDAN, N., AND PATERSON, K. Lucky Thirteen: Breaking the TLS and DTLS record protocols. In *Security and Privacy (SP), 2013 IEEE Symposium on* (2013), IEEE, pp. 526–540.

[4] ALFARDAN, N., BERNSTEIN, D. J., PATERSON, K. G., POETTERING, B., AND SCHULDT, J. C. N. On the security of RC4 in TLS. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)* (2013), USENIX, pp. 305–320.

[5] ANGLUIN, D. Learning regular sets from queries and counterexamples. *Information and Computation 75*, 2 (1987), 87–106.

[6] BENJAMIN BEURDOUCHE, KARTHIKEYAN BHARGAVAN, A. D.-L., FOURNET, C., KOHLWEISS, M., PIRONTI, A., STRUB, P.-Y., , AND ZINZINDOHOUE, J. K. A messy state of the union: Taming the composite state machines of TLS. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015), IEEE, pp. 535–552.

[7] BHARGAVAN, K., FOURNET, C., CORIN, R., AND ZALINESCU, E. Cryptographically verified implementations for TLS. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (2008), CCS '08, ACM, pp. 459–468.

[8] BHARGAVAN, K., FOURNET, C., KOHLWEISS, M., PIRONTI, A., AND STRUB, P. Implementing TLS with verified cryptographic security. *2013 IEEE Symposium on Security and Privacy* (2013), 445–459.

[9] BLEICHENBACHER, D. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In *Advances in Cryptology – CRYPTO '98*, H. Krawczyk, Ed., vol. 1462 of *Lecture Notes in Computer Science*. Springer, 1998, pp. 1–12.

[10] BRUBAKER, C., JANA, S., RAY, B., KHURSHID, S., AND SHMATIKOV, V. Using Frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *Security and Privacy (SP), 2014 IEEE Symposium on* (2014), pp. 114–129.

[11] CHALUPAR, G., PEHERSTORFER, S., POLL, E., AND DE RUITER, J. Automated reverse engineering using Lego. In *8th USENIX Workshop on Offensive Technologies (WOOT 14)* (2014), USENIX.

[12] CHOW, T. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering 4*, 3 (1978), 178–187.

[13] CODENOMICON. Heartbleed bug. `http://heartbleed.com/`. Accessed on June 8th 2015.

[14] COMPARETTI, P., WONDRACEK, G., KRUEGEL, C., AND KIRDA, E. Prospex: Protocol specification extraction. In *Security and Privacy, 2009 30th IEEE Symposium on* (2009), IEEE, pp. 110–125.

[15] DE RUITER, J. *Lessons learned in the analysis of the EMV and TLS security protocols*. PhD thesis, Radboud University Nijmegen, 2015.

[16] DÍAZ, G., CUARTERO, F., VALERO, V., AND PELAYO, F. Automatic verification of the TLS handshake protocol. In *Proceedings of the 2004 ACM Symposium on Applied Computing* (2004), SAC '04, ACM, pp. 789–794.

[17] DIERKS, T., AND ALLEN, C. The TLS protocol version 1.0. RFC 2246, Internet Engineering Task Force, 1999.

[18] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) protocol version 1.1. RFC 4346, Internet Engineering Task Force, 2006.

[19] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) protocol version 1.2. RFC 5246, Internet Engineering Task Force, 2008.

[20] GAJEK, S., MANULIS, M., PEREIRA, O., SADEGHI, A.-R., AND SCHWENK, J. Universally composable security analysis of TLS. In *Provable Security*, J. Baek, F. Bao, K. Chen, and X. Lai, Eds., vol. 5324 of *Lecture Notes in Computer Science*. Springer, 2008, pp. 313–327.

[21] GEER, D. Vulnerable compliance. *login: The USENIX Magazine 35*, 6 (2010), 10–12.

[22] HE, C., SUNDARARAJAN, M., DATTA, A., DEREK, A., AND MITCHELL, J. C. A modular correctness proof of IEEE 802.11i and TLS. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (2005), CCS '05, ACM, pp. 2–15.

[23] HSU, Y., SHU, G., AND LEE, D. A model-based approach to security flaw detection of network protocol implementations. In *Network Protocols, 2008. ICNP 2008. IEEE International Conference on* (2008), IEEE, pp. 114–123.

[24] JAGER, T., KOHLAR, F., SCHÄGE, S., AND SCHWENK, J. On the security of TLS-DHE in the standard model. In *Advances in Cryptology – CRYPTO 2012*, R. Safavi-Naini and R. Canetti, Eds., vol. 7417 of *Lecture Notes in Computer Science*. Springer, 2012, pp. 273–293.

[25] KALOPER-MERŠINJAK, D., MEHNERT, H., MADHAVAPEDDY, A., AND SEWELL, P. Not-quite-so-broken TLS: Lessons in re-engineering a security protocol specification and implementation. In *24th USENIX Security Symposium (USENIX Security 15)* (2015), USENIX Association.

[26] KAMIL, A., AND LOWE, G. Analysing TLS in the strand spaces model. *Journal of Computer Security 19*, 5 (2011), 975–1025.

[27] KIKUCHI, M. OpenSSL #ccsinjection vulnerability. `http://ccsinjection.lepidum.co.jp/`. Access on June 8th 2015.

[28] KRAWCZYK, H., PATERSON, K., AND WEE, H. On the security of the TLS protocol: A systematic analysis. In *Advances in Cryptology – CRYPTO 2013*, vol. 8042 of *Lecture Notes in Computer Science*. Springer, 2013, pp. 429–448.

[29] MEYER, C., AND SCHWENK, J. SoK: Lessons learned from SSL/TLS attacks. In *Information Security Applications*, Y. Kim, H. Lee, and A. Perrig, Eds., Lecture Notes in Computer Science. Springer, 2014, pp. 189–209.

[30] MEYER, C., SOMOROVSKY, J., WEISS, E., SCHWENK, J., SCHINZEL, S., AND TEWS, E. Revisiting SSL/TLS implementations: New bleichenbacher side channels and attacks. In *23rd USENIX Security Symposium (USENIX Security 14)* (2014), USENIX Association, pp. 733–748.

[31] MORRISSEY, P., SMART, N., AND WARINSCHI, B. A modular security analysis of the TLS handshake protocol. In *Advances in Cryptology – ASIACRYPT 2008*, J. Pieprzyk, Ed., vol. 5350 of *Lecture Notes in Computer Science*. Springer, 2008, pp. 55–73.

[32] OGATA, K., AND FUTATSUGI, K. Equational approach to formal analysis of TLS. In *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on* (2005), IEEE, pp. 795–804.

[33] PAULSON, L. C. Inductive analysis of the internet protocol TLS. *ACM Trans. Inf. Syst. Secur. 2*, 3 (1999), 332–351.

[34] RAFFELT, H., STEFFEN, B., AND BERG, T. LearnLib: a library for automata learning and experimentation. In *Formal methods for industrial critical systems (FMICS'05)* (2005), ACM, pp. 62–71.

[35] SASSAMAN, L., PATTERSON, M. L., AND BRATUS, S. A patch for Postel's robustness principle. *Security & Privacy, IEEE 10*, 2 (2012), 87–91.

[36] SEGGELMANN, R., TUEXEN, M., AND WILLIAMS, M. Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension. RFC 6520, Internet Engineering Task Force, 2012.

[37] TURNER, S., AND POLK, T. Prohibiting Secure Sockets Layer (SSL) version 2.0. RFC 6176, Internet Engineering Task Force, 2011.

[38] WHEELER, D. Preventing Heartbleed. *Computer 47*, 8 (2014), 80–83.

Verified correctness and security of OpenSSL HMAC

Lennart Beringer
Princeton Univ.

Adam Petcher
Harvard Univ. and MIT Lincoln Laboratory

Katherine Q. Ye
Princeton Univ.

Andrew W. Appel
Princeton Univ.

Abstract

We have proved, with machine-checked proofs in Coq, that an OpenSSL implementation of HMAC with SHA-256 correctly implements its FIPS functional specification *and* that its functional specification guarantees the expected cryptographic properties. This is the first machine-checked cryptographic proof that combines a source-program implementation proof, a compiler-correctness proof, and a cryptographic-security proof, with no gaps at the specification interfaces.

The verification was done using three systems within the Coq proof assistant: the Foundational Cryptography Framework, to verify crypto properties of functional specs; the Verified Software Toolchain, to verify C programs w.r.t. functional specs; and CompCert, for verified compilation of C to assembly language.

1 Introduction

HMAC is a cryptographic authentication algorithm, the "Keyed-Hash Message Authentication Code," widely used in conjunction with the SHA-256 cryptographic hashing primitive. The sender and receiver of a message m share a secret session key k. The sender computes $s = \text{HMAC}(k,m)$ and appends s to m. The receiver computes $s' = \text{HMAC}(k,m)$ and verifies that $s' = s$. In principle, a third party will not know k and thus cannot compute s. Therefore, the receiver can infer that message m really originated with the sender.

What could go wrong?

Algorithmic/cryptographic problems. The compression function underlying SHA might fail to have the cryptographic property of being a pseudorandom function (PRF); the SHA algorithm might not be the right construction over its compression function; the HMAC algorithm might fail to have the cryptographic property of being a PRF; we might even be considering the wrong crypto properties.

Implementation problems. The SHA program (in C) might incorrectly implement the SHA algorithm; the HMAC program might incorrectly implement the HMAC algorithm; the programs might be correct but permit side channels such as power analysis, timing analysis, or fault injection.

Specification mismatch. The specification of HMAC or SHA used in the cryptographic-properties [15] proof might be subtly different from the one published as the specification of computer programs [28, 27]. The proofs about C programs might interpret the semantics of the C language differently from the C compiler.

Based on Bellare and Rogaway's probabilistic game framework [16] for cryptographic proofs, Halevi [30] advocates creating an "automated tool to help us with the mundane parts of writing and checking common arguments in [game-based] proofs." Barthe *et al.* [13] present such a tool in the form of CertiCrypt, a framework that "enables the machine-checked construction and verification" of proofs using the same game-based techniques, written in code. Barthe *et al.*'s more recent EasyCrypt system [12] is a more lightweight, user-friendly version (but not *foundational,* i.e., the implementation is not proved sound in any machine-checked general-purpose logic). In this paper we use the Foundational Cryptography Framework (FCF) of Petcher and Morrisett [38].

But the automated tools envisioned by Halevi—and built by Barthe *et al.* and Petcher—address only the "algorithmic/cryptographic problems." We also need machine-checked tools for functional correctness of C programs—not just static analysis tools that verify the absence of buffer overruns. And we need the functional-correctness tools to connect, with machine-checked proofs of equivalence, to the crypto-algorithm proofs. By 2015, proof systems for formally reasoning about crypto algorithms and C programs have come far enough that it is now possible to do this.

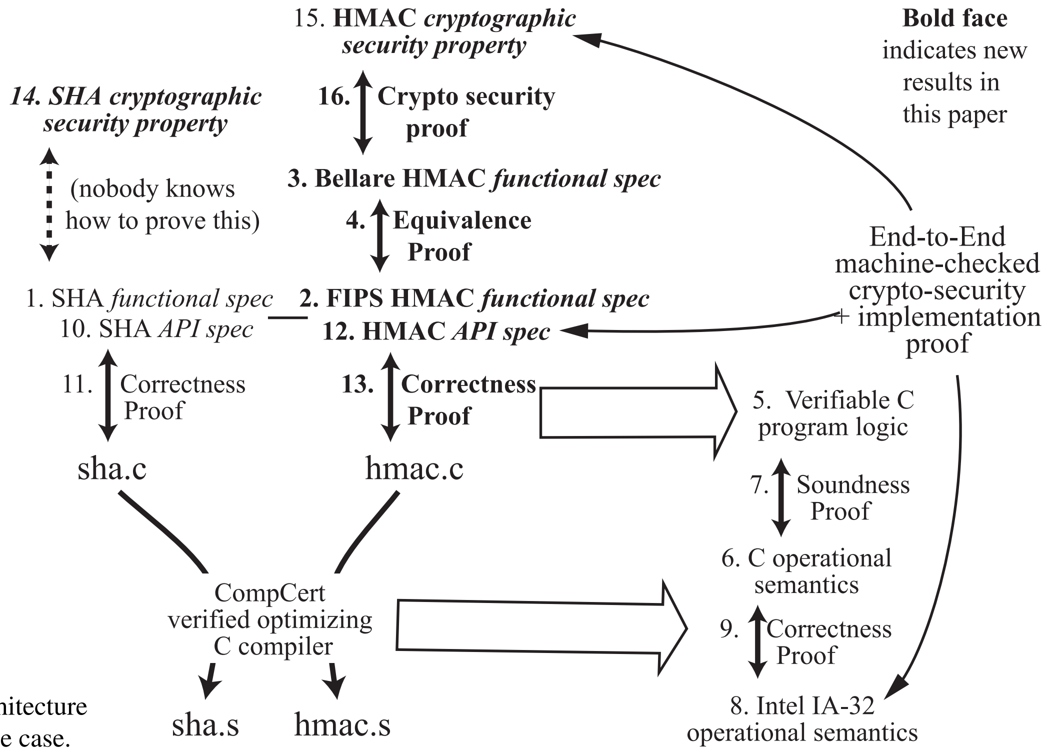

15. **HMAC** ***cryptographic security property***
Bold face indicates new results in this paper
14. SHA cryptographic security property
16. Crypto security proof
(nobody knows how to prove this)
3. Bellare HMAC *functional spec*
4. Equivalence Proof
End-to-End machine-checked crypto-security + implementation proof
1. SHA *functional spec*
10. SHA *API spec*
2. FIPS HMAC *functional spec*
12. HMAC *API spec*
11. Correctness Proof
13. Correctness Proof
5. Verifiable C program logic
sha.c
hmac.c
7. Soundness Proof
6. C operational semantics
CompCert verified optimizing C compiler
9. Correctness Proof
8. Intel IA-32 operational semantics
sha.s
hmac.s

Figure 1: Architecture of our assurance case.

Here we present machine-checked proofs, in Coq, of many components, connected and checked at their specification interfaces so that we get a truly end-to-end result: *Version 0.9.1c of OpenSSL's HMAC and SHA-256 correctly implements the FIPS 198-1 and FIPS 180-4 standards, respectively; and that same FIPS 198-1 HMAC standard is a PRF, subject to certain standard (unproved) assumptions about the SHA-256 algorithm that we state formally and explicitly.*

Software is large, complex, and always under maintenance; if we "prove" something about a real program then the proof (and its correspondence to the syntactic program) had better be checked by machine. Fortunately, as Gödel showed, checking a proof is a simple calculation. Today, proof checkers can be simple trusted (and trustworthy) kernel programs [7].

A *proof assistant* comprises a proof-checking kernel with an untrusted proof-development system. The system is typically *interactive,* relying on the user to build the overall structure of the proof and supply the important invariants and induction hypotheses, with many of the details filled in by tactical proof automation or by decision procedures such as SMT or Omega.

Coq is an open-source proof assistant under development since 1984. In the 21st century it has been used for practical applications such as Leroy's correctness proof of an optimizing C compiler [34]. But note, that compiler was not itself written in C; the proof theory of C makes life harder, and only more recently have people done proofs of substantial C programs in proof assistants [32, 29].

Our entire proof (including the algorithmic/cryptographic proofs, the implementation proofs, and the specification matches) is done in Coq, so that we avoid misunderstandings at interfaces. To prove our main theorem, we took these steps (cf. Figure 1):

1. *Formalized.*[5] We use a Coq formalization of the FIPS 180-4 Secure Hash Standard [28] as a specification of SHA-256. (Henceforth, "formalized" or "proved" implies "in the Coq proof assistant.")

2. *Formalized.** We have formalized the FIPS 198-1 Keyed-Hash Message Authentication Code [27] as a specification of HMAC. (Henceforth, the * indicates new work first reported in this paper; otherwise we provide a citation to previous work.)

3. *Formalized.** We have formalized Bellare's functional characterization of the HMAC algorithm.

4. *Proved.** We have proved the equivalence of FIPS 198-1 with Bellare's functional characterization of HMAC.

5. *Formalized.*[6] We use *Verifiable C*, a program logic (embedded in Coq) for specifying and proving functional correctness of C programs.

6. *Formalized.*[35] Leroy has formalized the operational semantics of the C programming language.

7. *Proved.*[6] Verifiable C has been proved sound. That is, if you specify and prove any input-output property of your C program using Verifiable C, then that property actually holds in Leroy's operational semantics of the C language.

8. *Formalized.*[35] Leroy has formalized the operational semantics of the Intel x86 (and PowerPC and ARM) assembly language.

9. *Proved.*[35] If the CompCert optimizing C compiler translates a C program to assembly language, then input-output property of the C program is preserved in the assembly-language program.

10. *Formalized.*[5] We rely on a formalization (in Verifiable C) of the *API interface* of the OpenSSL header file for SHA-256, including its semantic connection to the formalization of the FIPS Secure Hash Standard.

11. *Proved.*[5] The C program implementing SHA-256, lightly adapted from the OpenSSL implementation, has the input-output (API) properties specified by the formalized API spec of SHA-256.

12. *Formalized.** We have formalized the API interface of the OpenSSL header file for HMAC, including its semantic connection to our FIPS 198-1 formalization.

13. *Proved.** Our C program implementing HMAC, lightly adapted from the OpenSSL implementation, has the input-output (API) properties specified by our formalization of FIPS 198-1.

14. *Formalized.** Bellare *et al.* proved properties of HMAC [15, 14] subject to certain assumptions about the underlying cryptographic compression function (typically SHA). We have formalized those assumptions.

15. *Formalized.** Bellare *et al.* proved that HMAC implements a pseudorandom function (PRF); we have formalized what exactly that means. (Bellare's work is "formal" in the sense of rigorous mathematics and LaTeX; we formalized our work in Coq so that proofs of these properties can be machine-checked.)

16. *Proved.** We prove that, subject to these formalized assumptions about SHA, Bellare's HMAC algorithm is a PRF; this is a mechanization of a variant of the 1996 proof [15] using some ideas from the 2006 proofs [14].

Theorem. *The assembly-language program, resulting from compiling OpenSSL 0.9.1c using CompCert, correctly implements the FIPS standards for HMAC and SHA, and implements a cryptographically secure PRF subject to the usual assumptions about SHA.*

Proof. Machine-checked, in Coq, by chaining together specifications and proofs 1–16. Available open-source at https://github.com/PrincetonUniversity/VST/, subdirectories sha, fcf, hmacfcf.

The *trusted code base* (TCB) of our system is quite small, comprising only items 1, 2, 8, 12, 14, 15. Items 4, 7, 9, 11, 13, 16 need not be trusted, because they are proofs checked by the kernel of Coq. Items 3, 5, 6, 10 need not be trusted, because they are specification interfaces checked on both sides by Coq, as Appel [5, §8] explains.

One needs to trust the Coq kernel and the software that compiles it; see Appel's discussion [5, §12].

We do not analyze timing channels or other side channels. But the programs we prove correct are standard C programs for which standard timing and side-channel analysis tools and techniques can be used.

The HMAC brawl. Bernstein [19] and Koblitz and Menezes [33] argue that the security guarantees proved by Bellare *et al.* are of little value in practice, because these guarantees do not properly account for the power of precomputation by the adversary. In effect, they argue that item 15 in our enumeration is the wrong specification for desired cryptographic properties of a symmetric-key authentication algorithm. This may well be true; here we use Bellare's specification in a demonstration of end-to-end machine-checked proof. As improved specifications and proofs are developed by the theorists, we can implement them using our tools. Our proofs are sufficiently modular that only items 15 and 16 would change.

Which version of OpenSSL. We verified HMAC/SHA from OpenSSL 0.9.1c, dated March 1999, which does not include the home-brew object system "engines" of more recent versions of OpenSSL. We further simplified the code by specializing OpenSSL's use of generic "envelopes" to the specific hash function SHA-256, thus obtaining a statically linked code. Verifiable C is capable of reasoning about function pointers and home-brew object systems [6, Chapter 29]—it is entirely plausible that a formal specification of "engines" and "envelopes" could be written down—but such proofs are more complex.

2 Formalizing functional specifications

(Items 1, 2 of the architecture.) The FIPS 180-4 specification of the SHA function can be formalized in Coq as this mathematical function:

```
Definition SHA_256 (str : list Z) : list Z :=
  intlist_to_Zlist (
    hash_blocks init_registers (generate_and_pad str)).
```

where hash_blocks, init_registers, and generate_and_pad are translations of the FIPS standard. Z is Coq's type for (mathematical) integers; the (list Z) is the contents of a string of bytes, considered as their integer values. SHA-256 works internally in 32-bit unsigned modular arithmetic; intlist_to_Zlist converts a sequence of 32-bit machine ints to the mathematical contents of a byte-sequence. See Appel [5] for complete details. The functional spec of SHA-256, including definitions of all these functions, comes to 169 lines of Coq, all of which is in the trusted base for the security/correctness proof.

In this paper we show the full functional spec for HMAC256, the HMAC construction applied to hash function SHA_256:

```
Definition mkKey (l:list Z):list Z :=
  zeropad (if |l| > 64 then SHA_256 l else l).
Definition KeyPreparation (k: list Z):list byte :=
  map Byte.repr (mkKey k).
Definition HASH l m := SHA_256 (l++m)
Definition HmacCore m k :=
  HASH (opad ⊕ k) (HASH (ipad ⊕ k) m)
Definition HMAC256 (m k : list Z) : list Z :=
  HmacCore m (KeyPreparation k)
```

where zeropad right-extends[1] its argument to length 64 (i.e. to SHA256's block size, in bytes), ipad and opad are the padding constants from FIPS198-1, ⊕ denotes byte-wise XOR, and ++ denotes list concatenation.

3 API specifications of C functions

(Items 10, 12 of the architecture.) Hoare logic [31], dating from 1969, is a method of proving correctness of imperative programs using preconditions, postconditions, and loop invariants. Hoare's original logic did not handle pointer data structures well. Separation logic, introduced in 2001 [37], is a variant of Hoare logic that encapsulates "local actions" on data structures.

Verifiable C [6] is a separation logic that applies to the real C language. Verifiable C's rules are complicated in some places, to capture C's warts and corner cases.

The FIPS 180 and FIPS 198 specifications—and our definitions of SHA_256 and HMAC256—do not explain how the "mathematical" sequences of bytes are laid out in the arrays and structs passed as parameters to (and used internally by) the C functions. For this we need an *API spec.* Using Verifiable C, one specifies the API behavior of each function: the data structures it operates on, its preconditions (what it assumes about the input data structures available in parameters and global variables), and the postcondition (what it guarantees about its return value and changes to data structures). Appel [5, §7] explains how to build such API specs and shows the API spec for the SHA_256 function.

Here we show the API spec for HMAC. First we define a Coq record type,

```
Record DATA := { LEN:Z; CONT: list Z }.
```

If *key* has type DATA, then LEN(*key*) is an integer and CONT(*key*) is "contents" of the key, a sequence of integers. We do not use Coq's dependent types *here* to enforce that LEN corresponds to the length of the CONT field, but see the *has_lengthK* constraint below.

To specify the API of a C-language function in Verifiable C, one writes

DECLARE f WITH $\vec{v}$
    PRE[*params*] *Pre*   POST [*ret*] *Post*.

where f is the name of the function, *params* are the formal parameters (of various C-language types), and *ret* is the C return type. The precondition *Pre* and postcondition *Post* have the form PROP P LOCAL Q SEP R, where P is a list of pure propositions (true independent of the current program state), Q is a list of local/global variable bindings, and R is a list of *sep*aration logic predicates that describe the contents of memory. The WITH clause describes *logical* variables $\vec{v}$, abstract mathematical values that can be referred to anywhere in the precondition and postcondition.

In our HMAC256_spec, shown below, the first "abstract mathematical value" listed in this WITH clause is the key-pointer *kp*, whose "mathematical" type is "C-language value',' or val. It represents an address in memory where the HMAC session key is passed. In the LOCAL part of the PREcondition, we say that the formal parameter _key actually contains the value *kp* on entry to the function, and in the SEP part we say that there's a data_block at location *kp* containing the actual *key* bytes. In the postcondition we refer to *kp* again, saying that the data_block at address *kp* is still there, unchanged by the HMAC function.

[1] The more recent RFC4868 mandates that when HMAC is used for authentication, *a fixed key length equal to the output length of the hash functions MUST be supported, and key lengths other than the output length of the associated hash function MUST NOT be supported.* Our specification clearly separates KeyPreparation from HmacCore, but at the top level follows the more permissive standards RFC2104/FIPS198-1 as well as the implementation reality of even contemporary snapshots of OpenSSL and its clones.

```
Definition HMAC256_spec :=
 DECLARE _HMAC
  WITH kp: val, key:DATA, KV:val,
        mp: val, msg:DATA, shmd: share, md: val
  PRE [ _key OF tptr tuchar, _key_len OF tint,
        _d OF tptr tuchar, _n OF tint,
        _md OF tptr tuchar ]
   PROP(writable_share shmd;
         has_lengthK (LEN key) (CONT key);
         has_lengthD 512 (LEN msg) (CONT msg))
   LOCAL(temp _md md; temp _key kp; temp _d mp;
          temp _key_len (Vint (Int.repr (LEN key)));
          temp _n (Vint (Int.repr (LEN msg)));
          gvar _K256 KV)
   SEP(`(data_block Tsh (CONT key) kp);
       `(data_block Tsh (CONT msg) mp);
       `(K_vector KV);
       `(memory_block shmd (Int.repr 32) md))
 POST [ tvoid ]
   PROP() LOCAL()
   SEP(`(K_vector KV);
       `(data_block shmd
          (HMAC256 (CONT msg) (CONT key)) md);
       `(data_block Tsh (CONT key) kp);
       `(data_block Tsh (CONT msg) mp)).
```

The next WITH value is *key*, a DATA value, that is, a mathematical sequence of byte values along with its (supposed) length. In the PROP clause of the precondition, we enforce this supposition with *has_lengthK* (LEN *key*) (CONT *key*).

The function Int.repr injects from the mathematical integers into 32-bit signed/unsigned numbers. So temp _n (Vint (Int.repr (LEN *msg*))) means, take the mathematical integer (LEN *msg*), smash it into a 32-bit signed number, inject that into the space of C values, and assert that the parameter _n contains this value on entry to the function. This makes reasonable sense if $0 \leq$ LEN $msg < 2^{32}$, which is elsewhere enforced by *has_lengthD*. Such 32-bit range constraints are part of C's "warts and all," which are rigorously accounted for in Verifiable C. Both *has_lengthK* and *has_lengthD* are user-defined predicates within the HMAC API spec.

The precondition contains an uninitialized 32-byte memory_block at address *md*, and the _md parameter of the C function contains the value *md*. In the postcondition, we find that at address *md* the memory block has become an initialized data block containing a representation of HMAC256 (CONT *msg*) (CONT *key*).

For stating and proving these specifications, the following characteristics of separation logic are crucial:

1. The SEP lists are interpreted using the *separating conjunction* $*$ which (in contrast to ordinary conjunction $\wedge$) enforces disjointness of the memory regions specified by each conjunct. Thus, the precondition requires—and the postcondition guarantees—that keys, messages, and digests do not overlap.

2. Implicit in the semantic interpretation of a separation logic judgment is a *safety guarantee* of the absence of memory violations and other runtime errors, apart from memory exhaustion. In particular, verified code is guaranteed to respect the specified *footprint*: it will neither read from, nor modify or free any memory outside the region specified by the SEP clause of PRE. Moreover, all heap that is locally allocated is either locally freed, or is accounted for in POST. Hence, memory leaks are ruled out.

3. As a consequence of these locality principles, separation logic specifications enjoy a *frame property*: a verified judgment remains valid whenever we add an arbitrary additional separating conjunct to *both* SEP-clauses. The corresponding proof rule, the *frame rule*, is crucial for modular verification, guaranteeing, for example, that when we call SHA-256, the HMAC data structure remains unmodified.

The HMAC API spec has the 25 lines shown here plus a few more for definitions of auxiliary predicates (has_lengthK 3 lines, has_lengthD 3 lines, etc.); *plus* the API spec for SHA-256, all in the trusted base.

Incremental hashing. OpenSSL's HMAC and SHA functions are incremental. One can initialize the hasher with a key, then incrementally append message-fragments (not necessarily block-aligned) to be hashed, then finalize to produce the message digest. We fully support this incremental API in our correctness proofs. For simplicity we did not present it here, but Appel [5] presents the incremental API for SHA-256. The API spec for fully incremental SHA-256 is 247 lines of Coq; the simple (nonincremental) version has a *much* smaller API spec, similar to the 25+6 lines shown here for the nonincremental HMAC.

Once every function is specified, we use Verifiable C to prove that each function's body satisfies its specification. See Section 6.

4 Cryptographic properties of HMAC

(Items 14, 15, 16 of the architecture.) This section describes a mechanization of a cryptographic proof of security of HMAC. The final result of this proof is similar to the result of Bellare et al. [15], though the structure of the proof and some of the definitions are influenced

by Bellare's later proof [14]. This proof uses a more abstract model of HMAC (compared to the functional spec in §2) in which keys are in $\{0,1\}^b$ (the set of bit vectors of length b), inputs are in $\{0,1\}^*$ (bit lists), and outputs are in $\{0,1\}^c$ for arbitrary b and c such that $c \leq b$. An implementation of HMAC would require that b and c are multiples of some word size, and the input is an array of words, but these issues are typically not considered in cryptographic proofs.

In the context of the larger proof described in this paper, we refer to this model of HMAC in which sizes are arbitrary as the *abstract specification* of HMAC. In order to use security results related to this specification, we must show that this specification is appropriately related to the specification provided in §2. We chose to prove the security of the abstract specification, rather than directly proving the security of a more concrete specification, because there is significant value in this organization. Primarily, this organization allows us to use the exact definitions and assumptions from the cryptography literature, and we therefore gain greater assurance that the definitions are correct and the assumptions are reasonable. Also, this approach demonstrates how an existing mechanized proof of cryptographic security can be used in a verification of the security of an implementation. This organization also helps decompose the proof, and it allows us to deal with issues related to the implementation in isolation from issues related to cryptographic security.

We address the "gap" between the abstract and concrete HMAC specifications by proving that they are equivalent. Section 5 outlines the proof and states the equivalence theorem.

4.1 The Foundational Cryptography Framework

This proof of security was completed using the Foundational Cryptography Framework (FCF), a Coq library for reasoning about the security of cryptographic schemes in the computational model [38]. FCF provides a probabilistic programming language for describing all cryptographic constructions, security definitions, and problems that are assumed to be hard. Probabilistic programs are described using Gallina, the purely functional programming language of Coq, extended with a computational monad that adds sampling uniformly random bit vectors. The type of probabilistic computations that return values of type A is Comp A. The code uses {0,1}^n to describe sampling a bit vector of length n. Arrows (<-$) denote sequencing (i.e. bind) in the monad.

Listing 1 contains an example program implementing a one-time pad on bit vectors of length c (for any natural number c). The program produces a random bit vector and stores it in p, then returns the *xor* (using the standard Coq function BVxor) of p and the argument x.

```
Definition OTP c (x : Bvector c) : Comp (Bvector c)
  := p <-$ {0, 1}^c; ret (BVxor c p x)
```

Listing 1: Example Program: One-Time Pad.

The language of FCF has a denotational semantics that relates programs to discrete, finite probability distributions. A distribution on type A is modeled as a function in $A \rightarrow \mathbb{Q}$ which should be interpreted as a probability mass function. FCF provides a theory of distributions, a program logic, and a library of tactics that can be used to complete proofs without appealing directly to the semantics. We can use FCF to prove that two distributions are equivalent, that the distance between the probabilities of two events is bounded by some value, or that the probability of some event is less than some value. Such claims enable cryptographic proofs in the "sequence of games" style [16].

In some cryptographic definitions and proofs, an adversary is allowed to interact with an "oracle" that maintains state while accepting queries and providing responses. In FCF, an oracle has type S → A → Comp (B ∗ S) for types S, A, and B, of state, input, and output, respectively. The OracleComp type is provided to allow an adversary to interact with an oracle without viewing or modifying its state. By combining an OracleComp with an oracle and a value for the initial state of the oracle, we obtain a computation returning a pair of values, where the first value is produced by the OracleComp at the end of its interaction with the oracle, and the second value is the final state of the oracle.

4.2 HMAC Security

We mechanized a proof of the following fact. If h is a compression function, and h^* is a Merkle-Damgård hash function constructed from h, then HMAC based on h^* is a pseudorandom function (PRF) assuming:

1. h is a PRF.
2. h^* is weakly collision-resistant (WCR).
3. The dual family of h (denoted $\bar{h}$) is a PRF against $\oplus$-related-key attacks.

The formal definition of a PRF is shown in Listing 2. In this definition, f is a function in K → D → R that should be a PRF. That is, for a key k : K, an adversary who does not know k cannot gain much *advantage* in distinguishing f k from a random function in D → R.

The adversary A is an OracleComp that interacts with either an oracle constructed from f or with randomFunc, a random function constructed by producing random values for outputs and memoizing them so they can be repeated the next time the same input is provided. The randomFunc oracle uses a list of pairs as its state, so an empty list is provided as its initial state. The value tt is the "unit" value, where unit is a placeholder type much like "void" in the C language. This definition uses alternative arrows (such as <-$2) to construct sequences in which the first computation produces a tuple, and a name is given to each value in the tuple. The size of the tuple is provided in the arrow in order to assist the parser.

```
Definition f_oracle (k : K) (x : unit) (d : D)
                : Comp (R × unit) :=
  ret (f k d, tt).

Definition PRF_G0 : Comp bool :=
  k <-$ RndKey;
  [b, _] <-$2 A (f_oracle k) tt; ret b.

Definition PRF_G1 : Comp bool :=
  [b, _] <-$2 A (randomFunc) nil; ret b.

Definition PRF_Advantage : Rat :=
  | Pr[PRF_G0] -Pr[PRF_G1] |.
```

Listing 2: Definition of a PRF. The f_oracle function wraps the function f (closed over key k) and turns it into an oracle. A is an adversary. Comp bool is the type of probabilistic computations that produce a bool. Rat is the type of (unary, nonnegative) rational numbers.

This security definition is provided in the form of a "game" in which the adversary tries to determine whether the oracle is f (in game 0) or a random function (in game 1). After interacting with the oracle, the adversary produces a bit, and the adversary "wins" if this bit is likely to be different in the games. We define the *advantage* of the adversary to be the difference between the probability that it produces "true" in game 0 and in game 1. We can conclude that f is a PRF if this advantage is sufficiently small.

```
Definition Adv_WCR_G : Comp bool :=
  k <-$ RndKey;
  [d1, d2, _] <-$3 A (f_oracle k) tt;
  ret ((d1 != d2) && ((f k d1) ?= (f k d2))).

Definition Adv_WCR : Rat := Pr[Adv_WCR_G].
```

Listing 3: Definition of Weak Collision-Resistance.

Listing 3 defines a weakly collision-resistant function. This definition uses a single game in which the adversary is allowed to interact with an oracle defined by a keyed function f. At the end of this interaction, the adversary attempts to produce a collision, or a pair of different input values that produce the same output. In this game, we use ?= and != to denote tests for equality and inequality, respectively. The advantage of the adversary is the probability with which it is able to locate a collision.

Finally, the security proof assumes that a certain keyed function is a PRF against ⊕-related-key attacks (RKA). This definition (Listing 4) is similar to the definition of a PRF, except the adversary is also allowed to provide a value that will be xored with the unknown key before the PRF is called. Note that this assumption is applied to the *dual family* of h, in which the roles of inputs and keys are reversed. So a single input value is chosen at random and fixed, and the adversary queries the oracle by providing values which are used as keys.

```
Definition RKA_F (k: Bvector b) (s: unit)
            (p: Bvector b × Bvector c)
        : (Bvector c × unit) :=
  ret (f ((fst p) xor k) (snd p), tt).

Definition RKA_R (k: Bvector b)
          (s : list (Bvector c × Bvector c))
          (p: Bvector b × Bvector c)
          : (Bvector c × list (Bvector c × Bvector c) :=
  randomFunc s ((fst p) xor k, (snd p))

Definition RKA_G0 : Comp bool :=
  k <-$ RndKey; [b, _] <-$2 A (RKA_F k) tt; ret b.

Definition RKA_G1 : Comp bool :=
  k <-$ RndKey; [b, _] <-$2 A (RKA_R k) nil; ret b.

Definition RKA_Advantage : Rat :=
  | Pr[RKA_G0] -Pr[RKA_G1] |.
```

Listing 4: Definition of Security against ⊕ Related-Key Attacks. b is the key length of the compression function, c is the input length of the compression function; Bvector b is the type of bit-vectors of length b.

The proof of security has the same basic structure (Figure 2) as Bellare's more recent HMAC proof [14], though we simplify the proof significantly by assuming h^* is WCR. The proof makes use of a nested MAC (NMAC) construction that is similar to HMAC, but it uses h^* in a way that is not typically possible in implementations of hash functions. The proof begins by showing that NMAC is a PRF given that h is a PRF and h^* is WCR. Then we show that NMAC and HMAC are "close" (that no adversary can effectively distinguish them) under the assumption that $\bar{h}$ is a ⊕-RKA-secure PRF. Finally, we combine these two results to derive that HMAC is a PRF.

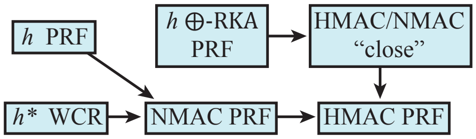


Figure 2: HMAC Security Proof Structure

We also mirror Bellare's proof by reasoning about slightly generalized forms of HMAC and NMAC (called GHMAC and GNMAC) that require the input to be a list of bit vectors of length b. The proof also makes use of a "two-key" version of HMAC that uses a bit vector of length $2b$ as the key. To simplify the development of this proof, we build HMAC on top of these intermediate constructions in the abstract specification (Listing 5).

```
Definition h_star k (m : list (Bvector b))
  := fold_left h m k.
Definition hash_words := h_star iv.

Definition GNMAC k m :=
  let (k_Out, k_In) := splitVector c c k in
  h k_Out (app_fpad (h_star k_In m)).

Definition GHMAC_2K k m :=
  let (k_Out, k_In) := splitVector b b k in
  let h_in := (hash_words (k_In :: m)) in
  hash_words (k_Out :: (app_fpad h_in) :: nil).

Definition HMAC_2K k (m : list bool) :=
  GHMAC_2K k (splitAndPad m).

Definition HMAC (k : Bvector b) :=
  HMAC_2K ((k xor opad) ++ (k xor ipad)).
```

Listing 5: HMAC Abstract Specification.

splitAndPad produces a list of bit-vectors from a list of bits (padding the last bit-vector as needed), and app_fpad is a padding function that produces a bit vector of length b from a bit vector of length c. In the HMAC function, we use constants opad and ipad to produce a key of length $2b$ from a key of length b.

The statement of security for HMAC is shown in Listing 6. We show that HMAC is a PRF by giving an expression that bounds the advantage of an arbitrary adversary A. This expression is the sum of three terms, where each term represents the advantage of some adversary against some other security definition.

The listing describes all the parameters to each of the security definitions. In all these definitions, the first parameter is the computation that produces random keys, and in PRF_Advantage and RKA_Advantage, the second parameter is the computation that produces random values in the range of the function. In all definitions, the penultimate parameter is the function of interest, and the final parameter is some constructed adversary. The descriptions of these adversaries are omitted for brevity, but only their computational complexity is relevant (e.g. all adversaries are in ZPP assuming adversary A is in ZPP).

```
Theorem HMAC_PRF:
  PRF_Advantage ({0, 1}^b) ({0, 1}^c) HMAC A <=
  PRF_Advantage ({0, 1}^c) ({0, 1}^c) h B1 +
  Adv_WCR ({0, 1}^c) h_star B2 +
  RKA_Advantage ({0, 1}^b) ({0, 1}^c)
    (BVxor b) (dual_f h) B3.
```

Listing 6: Statement of Security for HMAC.

We can view the result in Listing 6 in the asymptotic setting, in which there is a security parameter η, and parameters c and b are polynomial in η. In this setting, it is possible to conclude that the advantage of A against HMAC is negligible in η assuming that each of the other three terms is negligible in η. We can also view this result in the *concrete setting*, and use this expression to obtain *exact security* measures for HMAC when the values of b and c are fixed according the sizes used by the implementation. The latter interpretation is more informative, and probably more appropriate for reasoning about the cryptographic security of an implementation.

5 Equivalence of the two functional specs

(Item 4 of the architecture.) In §2 we described a bytes-and-words specification following FIPS198-1, suited for proving the C program; call that the *concrete specification*. In §4 we described a length-constrained bit-vector specification following Bellare *et al.*'s original papers; call that the *abstract specification*. Here we describe the proof that these two specifications are equivalent.

Proof outline. There are seven main differences between the concrete and abstract specs:

(0) The abstract spec, as its name suggests, leaves several variables as parameters to be instantiated. Thus, in order to compute with the abstract HMAC, one must pass it "converted" variables and "wrapped" functions from the concrete HMAC.

(1) The abstract spec operates on bits, whereas the concrete spec operates on bytes.

(2) The abstract spec uses the dependent type Bvector n, which is a length-constrained bit list of length n, whereas the concrete spec uses byte lists and int lists, whose lengths are unconstrained by definition.

(3) Due to its use of dependent types, the abstract spec must pad its input twice in an ad-hoc manner, whereas the concrete spec uses the SHA-256 padding function consistently.

(4) The concrete spec treats the hash function (SHA-256) as a black box, whereas the abstract spec exposes various parts of its functionality, such as its initialization vector, internal compression function, and manner of iteration. (It does this because the Bellare-style proofs rely on the Merkle-Damgård structure of the hash function.)

(5) The abstract spec pads the message and splits it into a list of blocks so that it can perform an explicit fold over the list of lists. However, the concrete spec leaves the message as a list of bytes and performs an implicit fold over the list, taking a new block at each iteration.

(6) The abstract spec defines HMAC via the HMAC_2K and GHMAC_2K structures, not directly.

Instantiating the abstract specification. The abstract HMAC spec leaves the following parameters abstract:

```
Variable c p : nat.

(* compression function *)
Variable h : Bvector c → Bvector b → Bvector c.

(* initialization vector *)
Variable iv : Bvector c.
Variable splitAndPad : Blist → list (Bvector b).
Variable fpad : Bvector c → Bvector p.
Variable opad ipad : Bvector b.
```

The abstract HMAC spec is also more general than the concrete spec, since it operates on bit vectors, not byte lists, and does not specify a block size or output size. After "replacing" the vectors with lists (see the explanation of difference (2)) and specializing $c = p = 256$ (resulting in $b = 512$), we may instantiate abstract parameters with concrete parameters or functions from SHA-256, wrapped in *bytesToBits* and/or *intlist_to_Zlist* conversion functions. For example, we instantiate the block size to 256 and the output size to 512, and define *iv* and *h* as:

```
Definition intsToBits := bytesToBits ∘ intlist_to_Zlist.
Definition sha_iv : Blist :=
  intsToBits SHA256.init_registers.
Definition sha_h (regs : Blist) (block : Blist) : Blist :=
  intsToBits (SHA256.hash_block (bitsToInts regs)
                                (bitsToInts block)).
```

The *intlist_to_Zlist* conversion function is necessary because portions of the SHA-256 spec operate on lists of *Integer*s, as specified in our bytes-and-words formalization of FIPS 180-4. (*Z* in Coq denotes arbitrary-precision mathematical integers. Our SHA-256 spec represents byte values as *Z*. An *Integer* is four byte-*Z*s packed big-endian into a 32-bit integer.)

We are essentially converting the types of the functions from functions on intlists (*intlist* → ... → *intlist*) to functions on Blists (*Blist* → ... → *Blist*) by converting their inputs and outputs.

Let us denote by HmacAbs256 the instantiation of function HMAC from Listing 5 to these parameters. Since Bellare's proof assumes that the given key is of the right length (the block size), our formal equivalence result relates HmacAbs256 to the function HmacCore from Section 2, i.e. to the part of HMAC256 that is applied after key length normalization. (Unlike Bellare, FIPS 198 includes steps to first truncate or pad the key if it is too long or short.)

Theorem. For key vector kv of type Bvector 256 and message m of type list bool satisfying $|l| \equiv 0 \ (mod\ 8)$,

$$\mathsf{HmacAbs256}\ kv\ m \approx \mathsf{HmacCore}\ \overline{m}\ (\mathsf{map\ Bytes.repr}\ \overline{kv}).$$

where $\overline{(\cdot)}$ denotes bitsToBytes conversion, and $\approx$ is equality modulo conversion between lists and vectors.

Reconciling other differences. The last difference (6) is easily resolved by unfolding the definitions of HMAC_2K and GHMAC_2K. We solve the other six problems by changing definitions and massaging the two specs toward each other, proving equality or equivalence each time.

Bridging (5) is basically the proof of correctness of a *deforestation* transformation. Consider a message m as a list of bits b_i. First, split it into 512-bit blocks B_i, then "fold" (the "reduce" operation in map-reduce) the hash operation H over it, starting with the initialization vector iv: $H(H(H(iv, B_0), B_1), \ldots, B_{n-1})$. Alternatively, express this as a recursive function on the original bit-sequence b: grab the first 512 bits, hash with H, then do a recursive call after skipping the first 512 bits:

```
Function F (r: list bool) (b: list bool)
      {measure length b} : list bool :=
match msg with
   nil ⇒ r
 | _ ⇒ F (H r (firstn 512 b)) (skipn 512 b)
end.
```

Provided that $|b|$ is a multiple of 512 (which we prove elsewhere), $F(iv, b) = H(H(H(iv, B_0), B_1), \ldots, B_{n-1})$.

We bridge (4) by using the fact that SHA-256 *is* a Merkle-Damgård construction over a compression function. This is a simple matter of matching the definition of SHA-256 to the definition of an MD hash function.

Bridging (3) is a proof that two different views of the SHA padding function are equivalent. Before iterating the compression function on the message, SHA-256 pads it in a standard, one-to-one fashion such that its length is a multiple of the block size. It pads it as such:

$$msg\,|\,[1]\,|\,[0,0,\ldots 0]\,|\,L$$

where | denotes list concatenation and L denotes the 64-bit representation of the length of the message. The number of 0s is calculated such that the length of the entire padded message is a multiple of the block size.

The abstract spec accomplishes this padding in two ways using the functions *fpad* and *splitAndPad*. *fpad* pads a message of known length of the output size c to the block size b, since c is specified to be less than b. *splitAndPad* breaks a variable-length message (of type list bool) into a list of blocks, each size b, padding it along the way. *fpad* is instantiated as a constant, since we know that the length of the message is $c < b$. *splitAndPad* is instantiated as the normal SHA padding function, but tweaked to add one block size to the length appended in $[l_1, l_2]$, since k_{in} (with a length of one block) will be prepended to the padded message later.

To eliminate these two types of ad-hoc padding, we rewrite the abstract spec to incorporate *fpad* and *splitAndPad* into a single padding function split_and_pad included in the hash function, in the style of SHA-256.

$$\mathsf{hash\_words\_padded} := \mathsf{hash\_words} \circ \mathsf{split\_and\_pad}.$$

We then remove *fpad* and *splitAndPad* from subsequent versions of the specification. We can easily prove equality by unfolding definitions.

Bridging bytes and bits. The abstract and concrete HMAC functions have different types, so we cannot prove them *equal*, only *equivalent*. $HMAC_c$ operates on (lists of) bits and $HMAC_a$ operates on (lists of) bytes. ($HMAC_c$ used to operate on vectors, but recall that we replaced them with lists earlier.) To bridge gap (1) we prove, given that the inputs are equivalent, the outputs will be equivalent:

$$k_c \approx k_a \rightarrow$$
$$m_c \approx m_a \rightarrow$$
$$HMAC_c(k_c, m_c) \approx HMAC_a(k_a, m_a).$$

The equivalence relation $\approx$ can be defined either computationally or inductively, and both definitions are useful.

To reason about the behavior of the wrapped functions with which we instantiated the abstract HMAC spec, we use the computational equivalence relation ($\approx_c$) instantiated with a generic conversion function. This allows us to build a framework for reasoning about the asymmetry of converting from bytes to bits versus from bits to bytes, as well as the behavior of repeatedly applied wrapped functions.

Bridging vectors and lists. We bridge (2) by changing all Bvector n to list bool, then proving that all functions preserve the length of the list when needed. This maintains the Bvector n invariant that its length is always n. In general, the use of lists (of bytes, or Z values) is motivated by the desire to reuse Appel [5]'s prior work on SHA literally, whereas the use of Bvector enables a more elegant proof of the proof of cryptographic properties.

Injectivity of splitAndPad. The security proof relies on the fact that splitAndPad is injective, in the sense that $b_1 = b_2$ should hold whenever splitAndPad(b_1) = splitAndPad(b_2). Indeed, this property is violated if we naively instantiate splitAndPad with the bitlists-to-bytelists roundtrip conversion of SHA-256's padding+length function, due to the non-injectivity of bitlists-to-bytelists conversion. On the other hand, as the C programs interpret all length informations as referring to lengths in bytes, attackers that attempt to send messages whose length is not divisible by 8 are effectively ruled out. To verify this property formally, we make the abstract specification (and the proof of **Theorem** HMAC_PRF) parametric in the type of messages. Instantiating the development to the case where messages are bitlists of length $8n$ allows us to establish the desired injectivity condition along the the lines of the following informal argument.

Given a message m, SHA's splitAndPad appends a 1 bit, then k zero bits, then a 64-bit integer representing the length of the message $|m|$; k is the smallest number so that $|\text{splitAndPad}(m_1)|$ is a multiple of the block size. Injective means that if $m_1 \neq m_2$ then $\text{splitAndPad}(m_1) \neq \text{splitAndPad}(m_2)$. The proof has five cases:

- $m_1 = m_2$, then by contradiction.
- $|m_1| = |m_2|$, then $\text{splitAndPad}(m_1)$ must differ from $\text{splitAndPad}(m_2)$ in their first $|m_1|$ bits.
- $|m_1| \neq |m_2|$, $|m_1| \leq 2^{64}$, $|m_2| \leq 2^{64}$, then the last 64 bits (representation of length) will differ.
- $(|m_1| - |m_2|) \bmod 2^{64} \neq 0$, then the last 64 bits (representation of length) will differ.
- $|m_1| \neq |m_2|$, and $(|m_1| - |m_2|) \bmod 2^{64} = 0$; then the lengths $|m_1|, |m_2|$ must differ by at least 2^{64}, so the variation in k_1 and k_2 (which must each be less than twice the block size) cannot make up the difference, so the padded messages will have different lengths.

Our machine-checked proof of injectivity is somewhat more comprehensive than Bellare et al.'s [15], which reads in its entirety, "Notice that a way to pad messages to an exact multiple of b bits needs to be defined, in particular, MD5 and SHA pad inputs to always include an encoding of their length."

Preservation of cryptographic security. Once the equivalence between the two functional programs has been established, and injectivity of the padding function is proved, it is straightforward to prove the applicability of **Theorem** HMAC_PRF (Listing 6) to the API spec.

6 Specifying and verifying the C program

(Items 11, 13 of the architecture.) We use Verifiable C to prove that each function's body satisfies its specification. As in a classic Hoare logic, each kind of C-language statement has one or more proof rules. Appel [6, Ch. 24-26] presents these proof rules, and explains how *tactics*—programmed in the Ltac language of Coq—apply the proof rules to the abstract syntax trees of C programs. The ASTs are obtained by applying the front-end phase of the CompCert compiler to the C program. The HMAC proof (item 13 in §1) is 2832 lines of Coq (including blanks and comments), none of which is in the trusted base because it is all machine-checked.

Just like OpenSSL's implementation of SHA-256, the C code implementing HMAC is *incremental*: the one-shot HMAC function is obtained by composing auxiliary functions hmacInit, hmacUpdate, hmacFinish, and hmacCleanup that are all exposed in the header file. They allow a client to reuse a key for the authentication of multiple messages, and also to provide each individual message in chunks, by repeatedly invoking hmacUpdate. To this end, the auxiliary functions employ the hash function's incremental interface and are formulated over a client-visible struct, HMAC_CTX. Specializing OpenSSL's original header file to the hash function SHA-256 yields the following:[2]

```
typedef struct hmac_ctx_st {
        SHA256_CTX md_ctx; // workspace
        SHA256_CTX i_ctx; // inner SHA structure
        SHA256_CTX o_ctx; // outer SHA structure
        unsigned int key_length;
        unsigned char key[64];
        } HMAC_CTX;

void HMAC_Init(HMAC_CTX *ctx,
                unsigned char *key, int len);

void HMAC_Update(HMAC_CTX *ctx,
                  const void *data, size_t len);

void HMAC_Final(HMAC_CTX *ctx,
                 unsigned char *md);

void HMAC_cleanup(HMAC_CTX *ctx);

unsigned char *HMAC(unsigned char *key,
                    int key_len,
                    unsigned char *d, int n,
                    unsigned char *md);
```

Fields i_ctx and o_ctx store partially constructed SHA data structures that are initialized during HMAC_Init to hold the ⊕ of the normalized key and ipad/opad, respectively, and are copied to the workspace md_ctx where the inner and outer hashing applications are performed.

Omitting the implementations of the other functions, the one-shot HMAC invokes the incremental functions on a freshly stack-allocated HMAC_CTX, where 32 is the digest length of SHA-256:

```
unsigned char *HMAC(unsigned char *key,
                    int key_len, unsigned char *d,
                    int n, unsigned char *md) {
  HMAC_CTX c; static unsigned char m[32];
  if (md == NULL) md=m;
  HMAC_Init(&c, key, key_len);
  HMAC_Update(&c,d,n);
  HMAC_Final(&c,md);
  HMAC_cleanup(&c);
  return(md);
}
```

In order to verify that this code satisfies the specification HMAC256_spec from Section 2, each incremental function is equipped with its individual Verifiable C specification. Each specification is formulated with reference to a suitable Coq function (or alternatively a propositional relation, as extractability is not required) that expresses the function's effect on the HMAC_CTX structure abstractly, without reference to the concrete memory layout.

More precisely, the logical counterpart of an HMAC_CTX structure is given by the Coq type

```
Inductive hmacabs :=
 HMACabs: ∀(ctx iSha oSha: s256abs)
             (keylen: Z) (key: list Z), hmacabs.
```

That is, an *HMAC abstract state* has five components: ctx, iSha, and oSha are *SHA abstract states*, keylen is an integer, and key is a list of (integer) byte values. Appel [5] defines SHA abstract states; if you initialize a SHA module and dump the first n bytes of a message into it, you get a value of type s256abs representing the abstract state of the incremental-mode SHA-256 program.

Appel also defines a relation, update_abs a c_1 c_2, saying that adding another (incremental mode) message fragment s to abstract state c_1 yields state c_2.

We define abstract states for HMAC, and the incremental-mode HMAC update relation, in terms of the SHA s256abs type and update_abs relation.

[2] During the verification, we observed that the fields key_length and key can be eliminated from hmac_ctx_st, for the price of minor alterations to the code, API specification, and proof. A similar modification has recently (and independently) been implemented in boringssl.

```
Definition hmacUpdate (data: list Z)
                        (h1 h2: hmacabs) : Prop :=
  match h1 with HMACabs ctx1 iS oS klen k
  ⇒ ∃ctx2, update_abs data ctx1 ctx2
     ∧ h2 = HMACabs ctx2 iS oS klen k
  end.
```

To connect these definitions to the *upper* parts of our verification architecture, we prove that the composition of these counterparts of the incremental functions (i.e. the counterpart of the one-shot HMAC) coincides with HMAC256 the FIPS functional specification from Section 3.

```
Definition hmacIncremental (k data dig:list Z) :=
∃hInit hUpd, hmacInit k hInit ∧
        hmacUpdate data hInit hUpd ∧
        hmacFinal hUpd dig.

Lemma hmacIncremental_sound k data dig:
       hmacIncremental k data dig →
       dig = HMAC256 data k.
Proof. ... Qed.
```

Downward, we connect hmacabs and HMAC_CTX by a separation logic representation predicate:

```
Definition hmacstate_ (h:hmacabs) (c:val): mpred:=
    EX r:hmacstate,
     !! hmac_relate h r
     && data_at Tsh t_struct_hmac_ctx_st r c.
```

where hmac_relate is a pure proposition specifying that each component of a concrete struct r has precisely the content prescribed by h.

Using these constructions, we obtain API specifications of the incremental functions such as HMAC_Update.

```
Definition HMAC_Update_spec :=
 DECLARE _HMAC_Update
 WITH h1: hmacabs, c : val, d:val, len:Z,
       data:list Z, KV:val
 PRE [ _ctx OF tptr t_struct_hmac_ctx_st,
       _data OF tptr tvoid, _len OF tuint]
   PROP(has_lengthD (s256a_len (absCtxt h1))
                        len data)
   LOCAL(temp _ctx c; temp _data d;
           temp _len (Vint (Int.repr len));
           gvar _K256 KV)
   SEP(`(K_vector KV); `(hmacstate_ h1 c);
        `(data_block Tsh data d))
 POST [ tvoid ]
   EX h2: hmacabs,
     PROP(hmacUpdate data h1 h2)
     LOCAL()
     SEP(`(K_vector KV); `(hmacstate_ h2 c);
          `(data_block Tsh data d)).
```

7 Proof effort

It is difficult to estimate the proof effort, as we used this case study to learn where to make improvements to the usability and automation of our toolset. However, we can give some numbers: size, in commented lines of code, of the specifications and proofs. Where relevant, we give the size of the corresponding C API or function.

Functional correctness proof of the C program:

C lines	Coq lines	SHA-256 component
	169	FIPS-180 functional spec of SHA
71	247	API spec of SHA-256
	1022	Lemmas about the functional spec
10	229	Proof of addlength function
81	1640	sha256_block_data_order()
10	43	SHA256_Init()
38	1682	SHA256_Update()
31	1484	SHA256_Final()
7	58	SHA256()
248	*6574*	*Total SHA-256*
	159	FIPS-198 functional spec of HMAC
25	374	API spec
25	*533*	*Total HMAC spec*
	875	Supporting lemmas
74	1530	HMAC_Init proof
7	101	HMAC_Update proof
27	196	HMAC_Final proof
5	31	HMAC_Cleanup proof
21	99	HMAC proof
134	*2832*	*Total HMAC proof*

FCF proof that HMAC is a PRF:

Coq lines	component
70	Bellare-style functional spec of HMAC
25	Statement, HMAC is a PRF
377	Proof, HMAC is a PRF
472	*Total*

Connecting Verifiable C proof to FCF proof:

Coq lines	component
3017	General equivalence proof of the two functional specs for any compression function
993	Specialization to SHA-256
4010	*Total*

8 Related work

We have presented a *foundational, end-to-end* verification. All the relevant aspects of cryptographic proofs or of the C programming language are defined and checked with respect to the foundations of logic. We say a reasoning engine for crypto is *foundational* if it is implemented in, or its implementation is proved correct in, a trustworthy general-purpose mechanized logic. We say a connection to a language implementation is *foundational* if the synthesizer or verifier is connected (with proofs in a trustworthy general-purpose mechanized logic) to the operational semantics compiled by a verified compiler.

Crypto verification. Smith and Dill [40] verify several block-cipher implementations written in Java with respect to a functional spec written either in Java or in ACL2. They compile to bytecode, then use a subset model of the JVM to generate straight-line code. This work is not end-to-end, as the JVM is unverified—and it wouldn't suffice to simply plug in a "verified" JVM, if one existed, without also knowing that the *same* specification of the JVM was used in both proofs. Their method applies only where the number of input bits is fixed and the loops can be completely unrolled. Their verifier would likely be applicable to the SHA-256 block shuffle function, but certainly not to the management code (padding, adding the length, key management, HMAC).

Cryptol [25] generates C or VHDL directly from a functional specification, where the number of input bits is fixed and the loops can be completely unrolled, i.e. the SHA-256 block shuffle, but not the SHA-256 management code or HMAC. The Cryptol synthesizer is not foundational because its semantics is not formally specified, let alone proved.

CAO is a domain specific language for crypto applications, which is compiled into C [11], and equipped with verification technology based on the FRAMA-C tool suite [4].

Libraries of arbitrary-precision arithmetic functions have been verified by Fischer [39] and Berghofer [17] using Isabelle/HOL. Bertot et al. [20] verify GMP's computation of square roots in Coq, based on Filliatre's CORRECTNESS tool for ML-style programs with imperative features [26]. Neither of these formalizations is foundatationally connected to a verified compiler.

Verified assembly implementations of arithmetic functions have been developed by Myreen and Curello [36] and Affeldt [1], who use, respectively, proof-producing (de)compilation and simulation to link assembly code and memory layout to functional specifications at (roughly) the level of our FIPS specifications. Chen et al. [24] verify the Montgomery step in Bernstein's high-speed implementation of elliptic curve 25519 [18], using a combination of SMT solving and Coq to implement a Hoare logic for Bernstein's portable assembly representation qhasm.

The abstraction techniques and representation predicates in these works are compatible with our memory layout predicates. One important future step is to condense commonalities of these libraries into an ontology for crypto-related reasoning principles, reusable across multiple language levels and realised in multiple proof assistants. Doing this is crucial for scaling our work to larger fragments of cryptographic libraries.

Formal verification of protocols is an established research area, and efforts to link abstract protocol verifications to implementations are emerging using automated techniques like model extraction or code generation [8], or interactive proof [2].

Toma and Borrione [41] use ACL2 to prove correctness of a VHDL implementation of the SHA-1 block-shuffle algorithm. There is no connection to (for example) a verified compiler for VHDL.

Backes *et al.* [10] verify mechanically (in EasyCrypt) that Merkle-Damgård constructions have certain security properties.

EasyCrypt. Almeida *et al.* [3] describe the use of their EasyCrypt tool to verify the security of an implementation of the RSA-OAEP encryption scheme. A functional specification of RSA-OAEP is written in EasyCrypt, which then verifies its security properties. An unverified Python script translates the EasyCrypt specification to (an extension of) C, then an extension of CompCert compiles it to assembly language. Finally, a leakage tool verifies that the assembly language program has no more program counter leakage than the source code, i.e. that the compiled program's trace of conditional branches is no more informative to the adversary than the source code's.

The EasyCrypt verifier is not foundational; it is an OCaml program whose correctness is not proved. The translation from C to assembly language is foundational, using CompCert. However, EasyCrypt's C code relies on bignum library functions, but provides no mechanism by which these functions can be proved correct.

CertiCrypt [13] is a system for reasoning about cryptographic algorithms in Coq; it is foundational, but (like EasyCrypt) has no foundational connection to a C semantics. **ZKCrypt**[9] is a synthesizer that generates C code for zero-knowledge proofs, implemented in CertiCrypt, also with no foundational connection to a C semantics.

Bhargavan *et al.* [21] "implement TLS with verified cryptographic security" in F# using the **F7** typechecker. F7 is not capable of reasoning about all of the required cryptographic/probabilistic relationships required to complete the proof. So parts of the proof are completed using EasyCrypt, and there is no formal relationship between the EasyCrypt proofs and the F7 proof; one must inspect the code to ensure that the things admitted in F7 are the same things that are proved in Easy-

Crypt. F7 is also not foundational, because the type checker has a large amount of trusted code and because it depends on the Z3 SMT solver. Another difference between this work and ours is that the code provides a reference implementation in F#, not an efficient implementation in C.

CryptoVerif [22] is a prover (implemented in OCaml) for security protocols in which one can, for example, extract a OCaml program from a CryptoVerif model [23]. CryptoVerif is not foundational, the extraction is not foundational, and the compiler for OCaml is not foundationally verified.

C program verification. There are many program analysis tools for C. Most of them do not address functional specification or functional correctness, and most are unsound and incomplete. They are useful in practice, but cannot be used for an end-to-end verification of the kind we have done.

Foundational formal verification of C programs has only recently been possible. The most significant such works are both operating-system kernels: seL4 [32] and CertiKOS [29]. Both proofs are refinement proofs between functional specifications and operational semantics. Both proofs are done in higher-order logics: seL4 in Isabelle/HOL and CertiKOS in Coq.

Neither of those proof frameworks uses separation logic, and neither can accommodate the use of addressable local variables in C. This means that OpenSSL's HMAC/SHA could not be proved in these frameworks, because it uses addressable local variables.

Additionally, neither of those proof frameworks can handle function pointers. OpenSSL uses function pointers in its "engines" mechanism, an object-oriented style of programming that dynamically connects components together, such as HMAC and SHA. The *Verifiable C* program logic is capable of reasoning about such object-oriented patterns in C [6, Chapter 29], although we have not done so in the work described in this paper.

9 Conclusion

Widely used open-source cryptographic libraries such as OpenSSL, operating systems kernels, and the C compilers that build them, form the backbone of the public's communication security. Since 2013 or so, it has become clear that hackers and nation-states (is there a difference anymore?) are willing to invest enormous resources in searching for vulnerabilities and exploiting them. Other authors have demonstrated that compilers [34] and OS kernels [32, 29] can be built to a *provable zero-functional-correctness-defect standard*. Here we have demonstrated the same, in a modular way, for key components of our common cryptographic infrastructure.

Functional correctness implies zero buffer-overrun defects as well. But there are side channels we have not addressed here, such as timing, fault-injection, and leaks through dead memory. Our approach does not solve these problems; but it makes them no worse. Because we can reason about standard C code, other authors' techniques for side channel analysis are applicable without obstruction.

Functional correctness (with respect to a specification) does not always guarantee that a program has abstract security properties. Here, by linking a proof of cryptographic security to a proof of program correctness, we provide that guarantee.

Acknowledgments. Funded in part by DARPA award FA8750-12-2-029 and by a grant from Google ATAP.

References

[1] AFFELDT, R. On construction of a library of formally verified low-level arithmetic functions. *Innovations in Systems and Software Engineering (ISSE) 9*, 2 (2013), 59–77.

[2] AFFELDT, R., AND SAKAGUCHI, K. An intrinsic encoding of a subset of C and its application to TLS network packet processing. *Journal of Formalized Reasoning 7*, 1 (2014), 63–104.

[3] ALMEIDA, J. B., BARBOSA, M., BARTHE, G., AND DUPRESSOIR, F. Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications security* (2013), ACM, pp. 1217–1230.

[4] ALMEIDA, J. B., BARBOSA, M., FILLIÂTRE, J., PINTO, J. S., AND VIEIRA, B. CAOVerif: An open-source deductive verification platform for cryptographic software implementations. *Sci. Comput. Program. 91* (2014), 216–233.

[5] APPEL, A. W. Verification of a cryptographic primitive: SHA-256. *ACM Trans. on Programming Languages and Systems 37*, 2 (Apr. 2015), 7:1–7:31.

[6] APPEL, A. W., DOCKINS, R., HOBOR, A., BERINGER, L., DODDS, J., STEWART, G., BLAZY, S., AND LEROY, X. *Program Logics for Certified Compilers*. Cambridge, 2014.

[7] APPEL, A. W., MICHAEL, N. G., STUMP, A., AND VIRGA, R. A trustworthy proof checker. *J. Automated Reasoning 31* (2003), 231–260.

[8] AVALLE, M., PIRONTI, A., AND SISTO, R. Formal verification of security protocol implementations: a survey. *Formal Asp. Comput. 26*, 1 (2014), 99–123.

[9] BACELAR ALMEIDA, J., BARBOSA, M., BANGERTER, E., BARTHE, G., KRENN, S., AND ZANELLA BÉGUELIN, S. Full proof cryptography: verifiable compilation of efficient zero-knowledge protocols. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 488–500.

[10] BACKES, M., BARTHE, G., BERG, M., GRÉGOIRE, B., KUNZ, C., SKORUPPA, M., AND BÉGUELIN, S. Z. Verified security of Merkle-Damgård. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th* (2012), IEEE, pp. 354–368.

[11] Barbosa, M., Castro, D., and Silva, P. F. Compiling CAO: from cryptographic specifications to C implementations. In *Principles of Security and Trust - Third International Conference, POST 2014, Proceedings* (2014), M. Abadi and S. Kremer, Eds., vol. 8414 of *Lecture Notes in Computer Science*, Springer, pp. 240–244.

[12] Barthe, G., Dupressoir, F., Grégoire, B., Kunz, C., Schmidt, B., and Strub, P.-Y. EasyCrypt: A tutorial. In *Foundations of Security Analysis and Design VII*. Springer, 2014, pp. 146–166.

[13] Barthe, G., Grégoire, B., and Zanella Béguelin, S. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2009), POPL '09, ACM, pp. 90–101.

[14] Bellare, M. New proofs for NMAC and HMAC: Security without collision-resistance. In *Advances in Cryptology-CRYPTO 2006*. Springer, 2006, pp. 602–619.

[15] Bellare, M., Canetti, R., and Krawczyk, H. Keying hash functions for message authentication. In *Advances in CryptologyCRYPTO96* (1996), Springer, pp. 1–15.

[16] Bellare, M., and Rogaway, P. Code-based game-playing proofs and the security of triple encryption. *IACR Cryptology ePrint Archive 2004* (2004), 331.

[17] Berghofer, S. Verification of dependable software using SPARK and Isabelle. In *6th International Workshop on Systems Software Verification, SSV 2011* (2011), J. Brauer, M. Roveri, and H. Tews, Eds., vol. 24 of *OASICS*, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 15–31.

[18] Bernstein, D. J. Curve25519: New Diffie-Hellman speed records. In *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, Proceedings* (2006), M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, Eds., vol. 3958 of *Lecture Notes in Computer Science*, Springer, pp. 207–228.

[19] Bernstein, D. J. The HMAC brawl. cr.yp.to/talks/2012.03.20/slides.pdf, Mar. 2012.

[20] Bertot, Y., Magaud, N., and Zimmermann, P. A proof of GMP square root. *J. Autom. Reasoning 29*, 3-4 (2002), 225–252.

[21] Bhargavan, K., Fournet, C., Kohlweiss, M., Pironti, A., and Strub, P. Implementing TLS with verified cryptographic security. In *Security and Privacy (SP), 2013 IEEE Symposium on* (2013), IEEE, pp. 445–459.

[22] Blanchet, B. A computationally sound mechanized prover for security protocols. *Dependable and Secure Computing, IEEE Transactions on 5*, 4 (2008), 193–207.

[23] Cadé, D., and Blanchet, B. Proved generation of implementations from computationally secure protocol specifications. In *Principles of Security and Trust*. Springer, 2013, pp. 63–82.

[24] Chen, Y., Hsu, C., Lin, H., Schwabe, P., Tsai, M., Wang, B., Yang, B., and Yang, S. Verifying curve25519 software. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), G. Ahn, M. Yung, and N. Li, Eds., ACM, pp. 299–309.

[25] Erkok, L., Carlsson, M., and Wick, A. Hardware/-software co-verification of cryptographic algorithms using Cryptol. In *Formal Methods in Computer-Aided Design, 2009 (FMCAD'09)* (2009), IEEE, pp. 188–191.

[26] Filliâtre, J. Verification of non-functional programs using interpretations in type theory. *J. Funct. Program. 13*, 4 (2003), 709–745.

[27] Keyed-hash message authentication code. Tech. Rep. FIPS PUB 198-1, Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD, July 2008.

[28] Secure hash standard (SHS). Tech. Rep. FIPS PUB 180-4, Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD, Mar. 2012.

[29] Gu, L., Vaynberg, A., Ford, B., Shao, Z., and Costanzo, D. CertiKOS: A certified kernel for secure cloud computing. In *Proceedings of the Second Asia-Pacific Workshop on Systems* (2011), APSys'11, ACM, pp. 3:1–3:5.

[30] Halevi, S. A plausible approach to computer-aided cryptographic proofs. http://eprint.iacr.org/2005/181, 2005.

[31] Hoare, C. A. R. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (October 1969), 578–580.

[32] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 207–220.

[33] Koblitz, N., and Menezes, A. Another look at HMAC. *Journal of Mathematical Cryptology 7*, 3 (2013), 225–251.

[34] Leroy, X. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL'06* (2006), pp. 42–54.

[35] Leroy, X. Formal verification of a realistic compiler. *Communications of the ACM 52*, 7 (2009), 107–115.

[36] Myreen, M. O., and Curello, G. Proof pearl: A verified bignum implementation in x86-64 machine code. In *Certified Programs and Proofs - Third International Conference, CPP 2013, Proceedings* (2013), G. Gonthier and M. Norrish, Eds., vol. 8307 of *Lecture Notes in Computer Science*, Springer, pp. 66–81.

[37] O'Hearn, P., Reynolds, J., and Yang, H. Local reasoning about programs that alter data structures. In *CSL'01: Annual Conference of the European Association for Computer Science Logic* (Sept. 2001), pp. 1–19. LNCS 2142.

[38] Petcher, A., and Morrisett, G. The foundational cryptography framework. In *Principles of Security and Trust - 4th International Conference, POST 2015, Proceedings* (2015), R. Focardi and A. C. Myers, Eds., vol. 9036 of *Lecture Notes in Computer Science*, Springer, pp. 53–72.

[39] Schmaltz, S. f. F. Formal verification of a big integer library including division. Master's thesis, Saarland University, 2007. busserver.cs.uni-sb.de/publikationen/Fi08DATE.pdf.

[40] Smith, E. W., and Dill, D. L. Automatic formal verification of block cipher implementations. In *Formal Methods in Computer-Aided Design (FMCAD'08)* (2008), IEEE, pp. 1–7.

[41] Toma, D., and Borrione, D. Formal verification of a SHA-1 circuit core using ACL2. In *Theorem Proving in Higher Order Logics*. Springer, 2005, pp. 326–341.

Not-quite-so-broken TLS: lessons in re-engineering a security protocol specification and implementation

David Kaloper-Meršinjak[†], Hannes Mehnert[†], Anil Madhavapeddy and Peter Sewell
University of Cambridge Computer Laboratory
first.last@cl.cam.ac.uk
[†] *These authors contributed equally to this work*

Abstract

Transport Layer Security (TLS) implementations have a history of security flaws. The immediate causes of these are often programming errors, e.g. in memory management, but the root causes are more fundamental: the challenges of interpreting the ambiguous prose specification, the complexities inherent in large APIs and code bases, inherently unsafe programming choices, and the impossibility of directly testing conformance between implementations and the specification.

We present *nqsb-TLS*, the result of our re-engineered approach to security protocol specification and implementation that addresses these root causes. The same code serves two roles: it is both a specification of TLS, executable as a test oracle to check conformance of traces from arbitrary implementations, and a usable implementation of TLS; a modular and declarative programming style provides clean separation between its components. Many security flaws are thus excluded by construction.

nqsb-TLS can be used in standalone Unix applications, which we demonstrate with a messaging client, and can also be compiled into Xen unikernels (specialised virtual machine image) with a trusted computing base (TCB) that is 4% of a standalone system running a standard Linux/OpenSSL stack, with all network traffic being handled in a memory-safe language; this supports applications including HTTPS, IMAP, Git, and Websocket clients and servers. Despite the dual-role design, the high-level implementation style, and the functional programming language we still achieve reasonable performance, with the same handshake performance as OpenSSL and 73% – 84% for bulk throughput.

1 Introduction

Current mainstream engineering practices for specifying and implementing security protocols are not fit for purpose: as one can see from many recent compromises of sensitive services, they are not providing the security we need. Transport Layer Security (TLS) is the most widely deployed security protocol on the Internet, used for authentication and confidentiality, but a long history of exploits shows that its implementations have failed to guarantee either property. Analysis of these exploits typically focusses on their immediate causes, e.g. errors in memory management or control flow, but we believe their root causes are more fundamental:

Error-prone languages: historical choices of programming language and programming style that tend to lead to such errors rather than protecting against them.

Lack of separation: the complexities inherent in working with large code bases, exacerbated by lack of emphasis on clean separation of concerns and modularity, and by poor language support for those.

Ambiguous and untestable specifications: the challenges of writing and interpreting the large and ambiguous prose specifications, and the impossibility of directly testing conformance between implementations and a prose specification.

In this paper we report on an experiment in developing a practical and usable TLS stack, *nqsb-TLS*, using a new approach designed to address each of these root-cause problems. This re-engineering, of the development process and of our concrete stack, aims to build in improved security from the ground up.

We demonstrate the practicality of the result in several ways: we show on-the-wire interoperability with existing stacks; we show reasonable performance, in both bulk transfer and handshakes; we use it in a test oracle, validating recorded packet traces which contain TLS sessions between other implementations; and we use it as part of a standalone instant-messaging client. In addition to use in such traditional executables, nqsb-TLS is usable in applications compiled into unikernels – type-safe, single-address-space VMs with TCBs that run directly on a hypervisor [32]. This integration into a unikernel stack lets us demonstrate a wide range of working systems, including HTTPS, IMAP, Git, and Websocket clients and servers, while sidestepping a further difficulty with radical solutions in this area: the large body of legacy code (in applications, operating systems, and libraries) that existing TLS stacks are intertwined with.

We assess the security of nqsb-TLS also in several ways: for each of the root causes above, we discuss why our approach rules out certain classes of associated flaws, with reference to an analysis of flaws found in previous TLS implementations; and we test our authentication logic with a large corpus of certificate chains generated by using the Frankencert fuzzer [8], which found flaws in several previous implementations. We have also made the system publically available for penetration testing, as a *Bitcoin Piñata*, an example unikernel using nqsb-TLS. This has a TCB size roughly 4% of that of a similar system using OpenSSL on Linux.

We describe our overall approach in the remainder of the introduction. We then briefly describe the TLS protocol (§2), analyse flaws previously found in TLS implementations (§3), and the result of applying our approach, dubbed nqsb-TLS (§4). We demonstrate the duality of nqsb-TLS next by using its specification to validate recorded sessions (§5) and executing its implementation to provide concrete services (§6). We evaluate the interoperability, performance, and security (§7) of nqsb-TLS, describe related work (§8), and conclude (§9).

nqsb-TLS is freely available under a BSD license (`https://nqsb.io`), and the data used in this paper is openly accessible [27].

1.1 Approach

A precise and testable specification for TLS In principle, a protocol specification should unambiguously define the set of all implementation behaviour that it allows, and hence also what it does not allow: it should be *precise*. This should not be confused with the question of whether a specification is loose or tight: a precise specification might well allow a wide range of implementation behaviour. It is also highly desirable for specifications to be *executable as test oracles*: given an implementation behaviour (perhaps a trace captured from a particular execution), the specification should let one compute whether it is in the allowed set or not.

In practice, the TLS specification is neither, but rather a series of RFCs written in prose [13, 14, 15]. An explicit and precise description of the TLS state machine is lacking, as are some security-critical preconditions of its transitions, and there are ambiguities in various semi-formal grammars. There is no way such prose documents can be executed as a test oracle to directly test whether implementation behaviour conforms to the specification. TLS is not unique in this, of course, and many other specifications are expressed in the same traditional prose style, but its disadvantages are especially serious for security protocols.

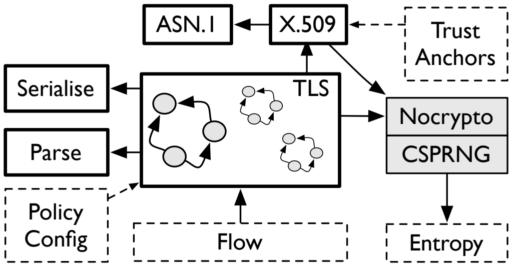


Figure 1: nqsb-TLS is broken down into strongly separated modules. The main part, in bold boxes, has pure value-passing interfaces and internals. The PRNG maintains internal state, while Nocrypto includes C code but has a pure effect-free interface. Arrows indicate depends-on relationships.

For nqsb-TLS, we specify TLS as a collection of pure functions over abstract datatypes. By avoiding I/O and shared mutable state, these functions can be considered in isolation and each is deterministic, with errors returned as explicit values. The top-level function takes an abstract protocol state and an incoming message, and calculates the next state and any response messages. To do so, it invokes subsidiary functions to parse the message, drive the state machine, perform cryptographic operations, and construct the response. This top-level function can be executed as a trace-checker, on traces both from our implementation and from others, such as OpenSSL, to decide whether they are allowed by our specification or not. In building our specification, to resolve the RFC ambiguities, we read other implementations and tested interoperability with them; we thereby capture the practical de facto standard.

Reuse between specification and implementation The same functions form the main part of our implementation, coupled with code for I/O and to provide entropy. Note that this is not an "executable specification" in the conventional sense: our specification is necessarily somewhat loose, as the server must have the freedom to choose a protocol version and cipher suite, and the trace checker must admit that, while our implementation makes particular choices.

Each version of the implementation (Unix, unikernel) has a top-level `Flow` module that repeatedly performs I/O and invokes the pure functional core; the trace-checker has a top-level module of the same type that reads in a trace to be checked offline.

Separation and modular structure This focus on pure functional descriptions also enables a decomposition of the system (both implementation and specification) into strongly separated modules, with typed interfaces, that interact only by exchanging pure values, as shown in Fig. 1. These modules and their interfaces are arranged to ensure that localised concerns such as binary protocol formats, ASN.1 grammars and certificate validation are not spread throughout the stack, with no implicit dependencies via shared memory.

External resources are explicitly represented as modules, instead of being implicitly accessed, and each satisfies a module type that describes collections of operations over an abstract type, and that can be instantiated with any of several implementations. These include the `Nocrypto` cryptography layer and our PRNG, which depends on an external `Entropy` module type.

Communication with the outside world is factored out into an I/O component, `Flow`, that passes a byte sequence to the pure core, then transmits responses and handles timeouts, and is used by the top-level but not by the TLS engine itself. The pure TLS engine depends on some external data, such as the policy config and trust anchors.

Choice of language and style The structure we describe above could be implemented in many different programming languages, but guarantees of memory and type safety are desirable to exclude many common security flaws (lack of memory safety was the largest single source of vulnerabilities in various TLS stacks throughout 2014, as shown in our §3 vulnerability analysis), and expressive statically checked type and module systems help maintain the strongly separated structure that we outlined. Our implementation of nqsb-TLS uses OCaml, a memory-safe, statically typed programming language that compiles to fast native code with a lightweight, embeddable runtime. OCaml supports (but does not mandate) a pure programming style, and has a module system which supports large-scale abstraction via ML functors – modules that can depend on other modules' types. In OCaml, we can encode complex state machines (§4), with lightweight invariants statically enforced by the type checker (state machine problems were the second largest source of vulnerabilities). Merely using OCaml does not guarantee all the properties we need, of course (one can write imperative and convoluted code in any language); our specification and programming styles are equally important.

This is a significant departure from normal practice, in which systems software is typically written in C, but we believe our evaluation shows that it is viable in at least some compelling scenarios (§7).

Non-goals For nqsb-TLS we are focussed on the engineering of TLS specifications and implementations, not on the security protocol itself (as we recall in §3, some vulnerabilities have been found there). We are also not attempting to advance the state of the art in side-channel defence, though we do follow current best practice. We are focussed on making a stack that is usable in practice and on security improvements achievable with better engineering processes, rather than trying to prove that a specification or implementation is correct or secure (see §8 for related work in that direction).

Current state The entire set of TLS RFCs [13, 14, 15] are implemented in nqsb-TLS, apart from minor rarely used features, such as DSS certificates and *anon* and pre-shared keys ciphersuites. As we demonstrate in §7.1, nqsb-TLS can interoperate with many contemporary TLS implementations, but we are not attempting to support legacy options or those of doubtful utility. We neither support SSLv3 [1], nor use RC4 in the default configuration [39]. The crypto wars are over: we have not implemented ciphersuites to adhere to export restrictions, which gave rise to the FREAK and Logjam attacks.

nqsb-TLS is strict (see §7.2), which results in roughly 10% failing connections from legacy clients. But since our main goal is to provide security, we are not willing to make compromises for insecure implementations. In addition to TLS itself, we also implemented ASN.1, X.509 and crypto primitives. From a practical point of view, the largest missing part is elliptic curve cryptography.

2 TLS Background

TLS provides the twin features of authentication and confidentiality. Clients typically verify the server's identity, the server can optionally verify the client's identity, while the two endpoints establish an encrypted communication channel. This channel should be immune from eavesdropping, tampering and message forgery.

There have been three standardised versions of TLS, 1.0, 1.1 and 1.2, while the last SSL (version 3) is still in wide usage. A key feature of TLS is algorithmic agility: it allows the two endpoints to negotiate the key exchange method, symmetric cipher and the message authentication mode upon connecting. This triple is called a cipher suite, and there are around 160 cipher suites standardised and widely supported. Together with a number of standardised extensions to the protocol that can be negotiated, this creates a large possible space of session parameters. This large variation in configuration options is a marked characteristic of TLS, significantly contributing to the complexity of its state machine.

Only a handful implementations of TLS are in wide use. The three major free or open-source implementations are OpenSSL, GnuTLS and Mozilla's NSS. Microsoft supplies SChannel with their operating systems, while Apple supplies Secure Transport with theirs, and Oracle Java runtime comes bundled with JSSE.

Structurally, TLS is a two-layered protocol. The outer layer preserves message boundaries and provides framing. It encapsulates one of five sub-protocols: handshake, change cipher spec, alert, application data or heartbeat. Both layers can contain fragmentation.

A TLS session is initiated by the client, which uses the handshake protocol to signal the highest protocol version, possible extensions, and a set of ciphersuites it supports. The server picks the highest protocol version it shares with the client and a mutually supported ciphersuite, or fails the handshake. The ciphersuite determines whether the server authenticates itself, and depending on the server configuration it requests the client to authenticate itself. After the security parameters for the authenticated encryption scheme are negotiated, the Change Cipher Spec message activates these, and the last handshake message authenticates the handshake. Either party can renegotiate the session over the established channel by initiating another handshake.

The handshake sub-protocol contains a complex state machine, which must be successfully traversed at least once. Handshake messages are independent of other sub-protocols, but some other sub-protocols are dependent of a successful handshake. For instance, it is not possible to exchange application data before a session is established, and it is impossible to affect the use of negotiated session parameters while the negotiation is still in progress.

Server and client authentication is performed by means of X.509 certificates. Usually path validation is used: after one party presents a sequence of certificates called the certificate chain, the other party needs to verify that a) each certificate in the chain is signed by the next certificate; b) the last certificate is signed by one of the trust anchors independent of connection; and c) that the first party owns the private key associated with the first certificate in the chain by transferring a signed message containing session-specific data. For correct authentication, the authenticating party also needs to verify general semantic well-formedness of the involved certificates, and be able to deal with three version of X.509 and a number of extensions.

X.509 certificates are described through ASN.1, a notation for describing the abstract syntax of data, and encoded using Distinguished Encoding Rules (DER), one of the several standard encodings ASN.1 defines. A particular description in the ASN.1 language coupled with a choice of encoding defines both the shape the the data-structures and their wire-level encoding. ASN.1 provides a rich language for describing structure, with a number of primitive elements, like `INTEGER` and `BIT STRING`, and combining constructs, like `SEQUENCE` (a record of sub-grammars) and `CHOICE` (a node joining alternative grammars). The ASN.1 formalism can be used with a compiler that derives parsing and serialisation code for the target language, but TLS implementations more typically contain custom parsing code for dealing with X.509 certificates. As X.509 exercises much of ASN.1, this parsing layer is non-trivial and significantly adds to the implementation complexity.

3 Vulnerability Analysis

In the past 13 months (January 2014 to January 2015), 54 CVE security advisories have been published for 6 widely used TLS implementations (see Table 1): 22 for OpenSSL, 6 for GnuTLS, 7 for NSS, 2 for SChannel, 2 for Secure Transport, 5 for JSSE, and 10 related to errors in their usage in the client software (excluding vulnerabilities related to DTLS – TLS over UDP).

These vulnerabilities have a wide range of causes. We classify them into broad families below, identifying root causes for each and discussing how nqsb-TLS avoids flaws of each kind.

General memory safety violations Most of these bugs, 15 in total, are memory safety issues: out-of-bounds reads, out-of-bounds writes and `NULL` pointer dereferences. A large group has only been demonstrated to crash the hosting process, ending in denial-of-service, but some lead to disclosure of sensitive information.

A now-notorious example of this class of bugs is Heartbleed in OpenSSL (CVE-2014-0160). Upon receiving a heartbeat record, a TLS endpoint should respond by sending back the payload of the record. The record contains the payload and its length. In Heartbleed, the TLS implementation did not check if the length of the received heartbeat matched the length encoded in the record, and responded by sending back as many bytes as were requested on the record level. This resulted in an out-of-bounds read, which lets a malicious client discover parts of server's memory. In April 2014, Cloudflare posed a challenge of exploiting this bug to compromise the private RSA key, which has been accomplished by at least four independent researchers.

nqsb-TLS avoids this class of issues entirely by the choice of a programming language with automated memory management and memory safety guarantees: in OCaml, array bounds are always checked and it is not possible to access raw memory; and our pure functional programming style rules out reuse of mutable buffers.

Certificate parsing TLS implementations need to parse ASN.1, primarily for decoding X.509 certificates. While ASN.1 is a large and fairly complex standard, for the purposes of TLS, it is sufficient to implement one of its encodings (DER), and only some of the primitives. Some TLS implementations contain an ad-hoc ASN.1 parser,

Product	CVE ID	Issue source
OpenSSL	2013-4353, 2015-0206, 2014-[3567, 3512, 3569, 3508, 3470, 0198, 0160]	Memory management
	2015-0205, 2015-0204, 2014-3572, 2014-0224, 2014-3568, 2014-3511	State machine
	2014-8275	Certificate parsing
	2014-2234	Certificate validation
	2014-3509, 2010-5298	Shared mutable state
	2014-0076	Timing side-channel
	2014-3570	Wrong sqrt
GnuTLS	2014-8564, 2014-3465, 2014-3466	Memory management
	2014-1959, 2014-0092, 2009-5138	Certificate validation
NSS	2014-1544	Memory management
	2013-1740	State machine
	2014-1490	Shared mutable state
	2014-1569, 2014-1568	Certificate parsing
	2014-1492	Certificate validation
	2014-1491	DH param validation
SChannel	2014-6321	Memory management
Secure Transport	2014-1266	State machine
JSSE	2014-6593, 2014-0626	State machine
	2014-0625	Memory exhaustion
	2014-0411	Timing side-channel
Applications	2014-2734	Memory management
	2014-3694, 2014-0139, 2014-2522, 2014-8151, 2014-1263	Certificate validation
	2013-7373, 2014-0016, 2014-0017, 2013-7295	RNG seeding
Protocol-level	2014-1771, 2014-1295, 2014-6457	Triple handshake
	2014-3566	POODLE

Table 1: Vulnerabilities in TLS implementations in 2014.

combining the core ASN.1 parsing task with the definitions of ASN.1 grammars, and this code operates as a part of certificate validation.

Unsurprisingly, ASN.1 parsing is a recurrent source of vulnerabilities in TLS and related software, dating back at least to 2004 (MS04-007, a remote code execution vulnerability), and 3 vulnerabilities in 2014 (CVEs 2014-8275, 2014-1568 and 2014-1569). Two examples are CVE-2015-1182, the use of uninitialised memory during parsing in PolarSSL, which could lead to remote code execution, and CVE-2014-1568, a case of insufficiently selective parsing in NSS, which allowed the attacker to construct a fake signed certificate from a large space of byte sequences interpreted as the same certificate.

This class of errors is due to ambiguity in the specification, and ad-hoc parsers in most TLS implementations. nqsb-TLS avoids this class of issues entirely by separating parsing from the grammar description (§4.4).

Certificate validation Closely related to ASN.1 parsing is certificate validation. X.509 certificates are nested data structures standardised in three versions and with various optional extensions, so validation involves parsing, traversing, and extracting information from complex compound data. This opens up the potential for errors both in the control-flow logic of this task and in the interpretation of certificates (multiple GnuTLS vulnerabilities are related to lax interpretation of the structures).

In 2014, there were 5 issues related to certificate validation. A prominent example in the control-flow logic is GnuTLS (CVE-2014-0092), where a misplaced `goto` statement lead to certificate validation being skipped if any intermediate certificate was of X.509 version 1.

Many implementations interleave the complicated X.509 certificate validation with parsing the ASN.1 grammar, leading to a complex control flow with subtle call chains. This illustrates another way in which the choice of programming language and style can lead to errors: the normal C idiom for error handling uses `goto` and negative return values, while in nqsb-TLS we return errors explicitly as values and have to handle all possible variants. OCaml's typechecker and pattern-match exhaustiveness checker ensures this at compile time (§4.3).

State machine errors TLS consists of several subprotocols that are multiplexed at the record level: *(i)* the handshake that initially establishes the secure connection and subsequently renegotiates it; *(ii)* alerts that signal out-of-band conditions; *(iii)* cipher spec activation notifications; *(iv)* heartbeats; and *(v)* application data. The majority of the TLS protocol specification covers the handshake state machine. The path to a successful negotiation is determined during the handshake and depends on the ciphersuite, protocol version, negotiated options, and configuration, such as client authentication. Errors in the handshake logic often lead to a security breach, allowing attackers to perform active man-in-the-middle (MITM) insertion, or to passively gain knowledge over the negotiated security parameters.

There were 10 vulnerabilities in this class. Some led to denial-of-service conditions caused (for example) by `NULL`-pointer dereferences on receipt of an unexpected message, while others lead to a breakdown of the TLS security guarantees. An extensive study of problems in TLS state machine implementations has been done in the literature [2, 11].

A prominent example is Apple's "goto fail" (CVE-2014-1266), caused by a repetition of a `goto` statement targeting the cleanup block of the procedure responsible for verifying the digital signature of the `ServerKeyExchange` message. This caused the procedure to skip the subsequent logic and return the value registered in the output variable. As this variable was initialised to "success", the signature was never verified.

Another typical example is the CCS Injection in OpenSSL (CVE-2014-0224). `ChangeCipherSpec` is the message signalling that the just negotiated security parameters are activated. In the TLS state machine, it is legal only as the penultimate message in the handshake sequence. However, both OpenSSL (CVE-2014-0224) and JSSE (CVE-2014-6593) allowed a CCS message before the actual key exchange took place, which activated predictable initial security parameters. A MITM attacker can exploit this by sending a CCS during handshake, causing two parties to establish a deterministic session key and defeating encryption.

Some of these errors are due to missing preconditions of state machine transitions in the specification. In nqsb-TLS, our code structure (§4.1) makes the need to consider each of these clear. We encode the state machine explicitly, while state transitions default to failure.

Protocol bugs In 2014, two separate issues in the protocol itself were described: POODLE and triple handshakes. POODLE is an attack on SSL version 3, which does not specify the value of padding bytes in CBC mode. Triple handshake [3] is a MITM attack where one negotiates sessions with the same security parameters and resumes. We do not claim to prevent nor solve those protocol bugs in nqsb-TLS, we mitigate triple handshake by resuming sessions only if the extended master secret [4] was used. Furthermore, we focus on a modern subset of the protocol, not including SSL version 3, so neither attack is applicable.

Timing side-channel leaks Two vulnerabilities were related to timing side-channel leaks, where the observable duration of cryptographic operations depended on cryptographic secrets. These were implementation issues, related to the use of variable-duration arithmetic operations. The PKCS1.5 padding of the premaster secret is transmitted during an RSA key exchange. If the unpadding fails, there is computationally no need to decrypt the received secret material. But omitting this step leaks the information on whether the padding was correct through the time signature, and this can be used to obtain the secret. A similar issue was discovered in 2014 in various TLS implementations [34].

nqsb-TLS mitigates this attack by always computing the RSA operation, on padding failure with a fake value. To mitigate timing side-channels, which a memory managed programming language might further expose, we use C implementations of the low level primitives (§4.2).

Usage of the libraries Of the examined bugs, 10 were not in TLS implementations themselves, but in the way the client software used them. These included the high-profile anonymisation software Tor [16], the instant messenger Pidgin and the widely used multi-protocol data transfer tool cURL.

TLS libraries typically have complicated APIs due to implementing a protocol with a large parameter space. For example, OpenSSL 1.0.2 documents 243 symbols in its protocol alone, not counting the cryptographic parts of the API. Parts of its API are used by registering callbacks with the library that get invoked upon certain events. A well-documented example of the difficulty in correctly using these APIs is the OpenSSL certificate validation callback. The library does not implement the full logic that is commonly needed (it omits name validation), so the client needs to construct a function to perform certificate validation using a mix of custom code and calls to OpenSSL, and supply it to the library. This step is a common pitfall: a recent survey [23] showed that it is common for OpenSSL clients in the wild to do this incorrectly. We counted 6 individual advisories stemming from improper usage of certificate validation API, which is a large number given that improper certificate validation undermines the authentication property of TLS and completely undermines its security.

The root cause of this error class is the large and complex legacy APIs of contemporary TLS stacks. nqsb-TLS does not mirror those APIs, but provides a minimal API with strict validation by default. This small API is sufficient for various applications we developed. OpenBSD uses a similar approach with their `libtls` API.

4 The *nqsb-TLS* stack

We now describe how we structure and develop the nqsb-TLS stack, following the approach outlined in the introduction to avoid a range of security pitfalls.

4.1 TLS Core

The heart of our TLS stack is the core protocol implementation. By using pure, composable functions to express the protocol handling, we deal with TLS as a data-transformation pipeline, independent of how the data is obtained or transmitted.

Accordingly, our core revolves around two functions. One (handle_tls) takes the sequence of bytes seen on the wire and a value denoting the previous state, and produces, as new values, the bytes to reply with or to transfer to the application, and the subsequent state. Our state type encapsulates all the information about a TLS session in progress, including the state of the handshake, the cryptographic state for both directions of communication, and the incomplete frames previously received, as an immutable value. The other one (send_application_data) takes a sequence of bytes that the application wishes to send and the previous state, and produces the sequence ready to be sent and the subsequent state. Coupled with a few operations to extract session information from the state, these form the entire interface to the core protocol implementation.

Below the entry points, we segment the records, decrypt and authenticate them, and dispatch to the appropriate protocol handler. One of the places where OCaml helps most prominently is in handling of the combined state machine of handshake and its interdependent subprotocols. We use algebraic data types to encode each possible handshake state as a distinct type variant, that symbolically denotes the state it represents and contains all of the data accumulated so far. The overall state type is simply the discriminated union of these variants. Every operation that extracts information from state needs to scrutinise its value through a form of multi-way branching known as pattern match. This syntactic construct combines branching on the particular variant of the state present with extraction of components. The resulting dispatch leads to equation-like code: branches that deal with distinct states follow directly from the values representing them, process the state data locally, and remain fully independent in the sense of control flow and access to values they operate on. Finally, each separately materialises the output and subsequent state.

This construction and the explicit encoding of state-machine is central to maintaining the state-machine invariants and preserving the coherence of state representation. It is impossible to enter a branch dedicated to a particular transition without the pair of values representing the appropriate state and appropriate input message, and, as intermediate data is directly obtained from the state value, it is impossible to process it without at the same time requiring that the state-machine is in the appropriate state. It is also impossible to manufacture a state-representation ahead of time, as it needs to contain all of the relevant data.

The benefit of this encoding is most clearly seen in CCS-injection-like vulnerabilities. They depend on session parameters being stored in locations visible throughout the handshake code, which are activated on receipt of the appropriate message. In the OpenSSL case (CVE-2014-0224), the dispatch code failed to verify whether all of these locations were populated, which implies that the handshake progressed to the appropriate phase. In our case, the only way to refer to the session parameters is to deconstruct a state-value containing them, and it is impossible to create this value without having collected the appropriate session parameters.

All of core's inner workings adhere to a predictable, restricted coding style. Information is always communicated through parameters and result values. Error propagation is achieved exclusively through results, without the use of exceptions. We explicitly encode errors distinct from successful results, instead of overloading the result's domain to mean error in some parts of its range. The type checker verifies both that each code path is dealing with exactly one possibility, and – through the exhaustiveness checker – that both forms have been accounted for. The repetitive logic of testing for error results and deciding whether to propagate the error or proceed is then abstracted away in a few higher-order functions and does not re-appear throughout the code.

This approach has also proven convenient when maintaining a growing code-base: when we had to add significant new capabilities, e.g. extending the TLS version support to versions 1.1 and 1.2 or implementing client authentication, the scope of changes was localised and the effects they had on other modules were flagged by the type checker.

4.2 Nocrypto

TLS cryptography is provided by *Nocrypto*, a separate library we developed for that purpose. It supports basic modular public-key primitives like RSA, DSA and DH; the two most commonly used symmetric block ciphers, AES and 3DES; the most important hash functions, MD5, SHA and the SHA2 family; and an implementation of the cryptographically strong pseudorandom number generator, Fortuna [20].

One of the fundamental design decisions was to use block-level symmetric encryption and hash cores written in C. For hashing, DES, and the portable version of AES, we use widely available public domain code. In addition, we wrote our own AES core using the Intel AES-NI instructions.

There are two reasons for using C at this level. Firstly, symmetric encryption and hashing are the most CPU-intensive operations in TLS. Therefore, performance concerns motivate the use of C. Secondly, the security impact of writing cryptography in a garbage-collected environment is unclear. Performing computations over secret material in this context is a potential attack vector. The garbage collector pauses might act as an amplifier to any existing timing side-channel leaks, revealing information about the allocation rate. We side-step this issue by leaving the secret material used by symmetric encryption opaque to the OCaml runtime.

Such treatment creates a potential safety problem in turn: even if we manage to prevent certain classes of bugs in OCaml, they could occur in our C code. Our strategy to contain this is to restrict the scope of C code: we employ simple control flow and never manage memory in the C layer. C functions receive pre-allocated buffers, tracked by the runtime, and write their results there. The most complex control flow in these are driving loops that call the compression function (in the case of hashes), or the block transform (in the case of ciphers), over the contents of the input buffer. AES-NI instructions are particularly simplifying in this respect, as the code consists of a sequence of calls to compiler intrinsics.

Presently, only the AES-NI implementation of AES is protected from timing side-channel leaks, since the bulk of the cipher is implemented via constant-time dedicated instructions. The generic code path is yet to be augmented with code to pre-load substitution tables in a non-data-dependent manner.

More complex cryptographic constructions, like cipher modes (CBC, CTR, GCM and CCM) and HMAC are implemented in OCaml on top of C-level primitives. We benefit from OCaml's safety and expressive power in these more complex parts of the code, but at the same time preserve the property that secret material is not directly exposed to the managed runtime.

Public key cryptography is treated differently. It is not block-oriented and is not easily expressed in straight-line code, while the numeric operations it relies on are less amenable to C-level optimisation. At the same time, there are known techniques for mitigating timing leaks at the algorithmic level [28], unlike in the symmetric case. We therefore implement these directly in OCaml using GMP as our bignum backend and employ the standard blinding countermeasures to compensate for potential sources of timing side-channels.

Our Fortuna CSPRNG uses AES-CTR with a self-rekeying regime and a system of entropy accumulators. Instead of entropy estimation, it employs exponential lagging of accumulators, a scheme that has been shown to asymptotically optimally recover from state compromise under a constant input of entropy of unknown quality [17]. To retain purity of the system and facilitate deterministic runs, entropy itself is required from the system as an external service, as shown later in §6.

For the sake of reducing complexity in the upper layers, the API of *Nocrypto* is concise and retains the applicative style, mapping inputs to outputs. We did make two concessions to further simplify it: first, we use OCaml exceptions to signal programming errors of applying cyptographic operations to malformed input (such as buffers which are not a multiple of the block size in CBC mode, or the use of RSA keys unsuitably small for a message). Secondly, we employ a global and changing RNG state, because operations involving it are pervasive throughout interactions with the library and the style of explicit passing would complicate the dependent code.

4.3 X.509

X.509 certificates are rich tree-like data structures whose semantics changes with the presence of several optional extensions. Although the core of the path-validation process is checking of the signature, a cryptographic operation, the correct validation required by the standard includes extensive checking of the entire data structure.

For example, each extension must be present at most once, the key usage extension can further constrain which exact operations a certificate is authorised for, and a certificate can specify the maximal chain length which is allowed to follow. There are several ways in which a certificate can express its own identity and the identity of its signing certificate. After parsing, a correct validation procedure must take all these possibilities into account.

The ground encoding of certificates again benefits from algebraic data types, as the control flow of functions that navigate this structure is directed by the type-checker. On a level above, we separate the validation process into a series of functions computing individual predicates, such as the certificate being self-signed, its validity period matching the provided time or conformance of the present extensions to the certificate version. The conjunction of these is clearly grouped into single top-level functions validating certificates in different roles, which describe the high-level constraints we impose upon the certificates. The entire validation logic amounts to 314 lines of easily reviewable code.

This is in contrast to 7 000 lines of text in the RFC [9], which go into detail to explain extensions – such as policies and name constraints – that are rarely seen in the wild. For the typical HTTPS setting, the RFC fails to clarify how to search for a trust anchor, and assumes instead the presence of exactly one. Due to cross signing there can be multiple chains with different properties which are not covered by the RFC.

nqsb-TLS initially strictly followed the RFC, but was not able to validate many HTTPS endpoints on the Internet. It currently follows the RFC augmented with

Mozilla's guidelines and provides a self-contained condensation of these which can be used to clarify, or even supplant, the specification. We created an extensive test suite with full code coverage, the code has been evaluated (see §7.2) with the Frankencert tool, and it successfully parses most of ZMap's certificate repositories. In addition, we also support signing and serialising to PEM.

The interface to this logic is deterministic (it is made so by requiring the current time as an input). Our X.509 library provides operations to construct a full *authenticator*, by combining the validation logic with the current time at the moment of construction, which the TLS core can be parametrised with. We do not leave validation to the user of the library, unlike other TLS libraries [23]. Instead, we have full implementations of path validation with name checking [42] and fingerprint-based validation, and we use the type system to force the user to instantiate one of them and provide it to the TLS layer.

4.4 ASN.1

ASN.1 parsing creates a tension in TLS implementations: TLS critically relies on ASN.1, but it requires only a subset of DER encoding, and, since certificates are usually pre-generated, needs very little in the way of writing. For the purposes of TLS, it is therefore sufficient to implement just a partial parser.

When implementing ASN.1, a decision has to be made on how to encode the actual abstract grammar that will drive the parsing process, given by various TLS and X.509-related standards. OpenSSL, PolarSSL, JSSE and others, with the notable exception of GnuTLS, do not make any attempts to separate the grammar definition from the parsing process. The leaf rules of ASN.1 are implemented as subroutines, which are exercised in the order required by the grammar in every routine that acts as parser. In other words, they implement the parsers as ad-hoc procedures that interleave the code that performs the actual parsing with the encoding of the grammar to be parsed. Therefore the code that describes the high-level structure of data also contains details of invocation of low-lever parsers and, in the case of C, memory management. Unsurprisingly, ASN.1 parsers provide a steady stream of exploits in popular TLS implementations.

We retain the full separation of the abstract syntax representation from the parsing code, avoiding the complexity of the code that fuses the two. At the same time, we avoid parser generators which output source code that is hard to understand.

Instead, we created a library for declaratively describing ASN.1 grammars in OCaml, using a functional technique known as combinatory parsing [21]. It exposes an opaque data type that describes ASN.1 grammar instances and provides a set of constants (corresponding to terminals) and functions over them (corresponding to productions). Nested applications of these functions to create data that describes ASN.1 grammars follow the shape of the actual ASN.1 grammar definitions. Internally, this tree-like type is traversed at initialisation-time to construct the parsing and serialisation functions.

This approach allows us to create "grammar expressions" which encode ASN.1 grammars, and derive parsers and serialisers. As the ASN.1-language we create is a fragment of OCaml, we retain all the benefits of its static type checking. Types of functions over grammar representations correspond to restrictions in the production rules, so type-checking grammar expressions amounts to checking their well-formedness without writing a separate parser for the grammar formalism. Moreover, type inference automatically derives the OCaml representation of the types defined by ASN.1 grammars.

Such an approach also makes testing much easier. The grammar type is traversed to generate random inhabitants of the particular grammar, which can be serialised and parsed back to check that the two directions match in their interpretation of the underlying ASN.1 encoding and to exercise all of the code paths in both.

A derived parsing function does not interpret the grammar data, but as its connections to component parsing functions are known only when synthesis takes place at run-time, we do not retain the benefit of inlining and inter-function optimisation a truly compiled parser would have. Nonetheless, given that it parses roughly 50 000 certificates per second, this approach does not create a major performance bottleneck. The result is a significant reduction in code complexity: the ASN.1 parsing logic amounts to 620 lines of OCaml, and the ASN.1 grammar code for X.509 certificates and signatures is around 1 000 lines. For comparison, PolarSSL 1.3.7 needs around 7 500 lines to parse ASN.1, while OpenSSL 1.0.1h has around 25 000 in its ASN.1 parser.

5 Using nqsb-TLS as a test oracle

One use of nqsb-TLS is as an *executable test oracle*, an application which reads a recorded TLS session trace and checks whether it (together with some configuration information) adheres to the specification that nqsb-TLS embodies. This recorded session can be a packet capture (using tcpdump) of a TLS session between various implementations (e.g. OpenSSL, PolarSSL, nqsb-TLS), or, for basic testing, a trace generated by nqsb-TLS itself.

To do this we must deal with the looseness of the TLS specification: a TLS client chooses its random nonce, set of ciphersuites, protocol version, and handshake extensions, while a TLS server picks its random nonce, the protocol version, the ciphersuite, possibly the DH group, and possibly extensions. Our test oracle does not make

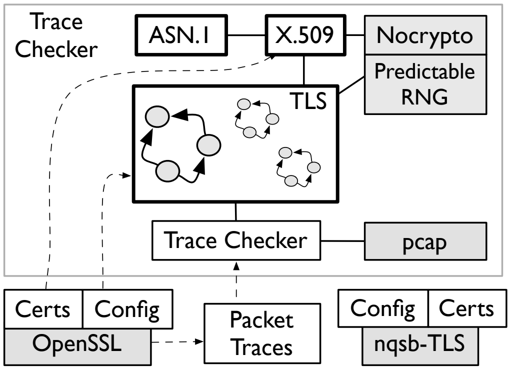


Figure 2: nqsb-TLS acts as a trace checker: the RNG is predictable, configuration and certificates are inputs, driven by packet traces from OpenSSL (or other stacks).

those decisions, but rather takes the parameters recorded in the given session. To make this possible, given the on-the-wire encryption, some configuration information has to be provided to the trace checker, including private key material. In addition, both records and sub-protocols can be fragmented; our test oracle normalises the records to not contain any fragmentation for comparison.

Figure 2 shows how nqsb-TLS can be used to build such a test oracle (note that it does not instantiate the entropy source for this usage). The test oracle produces its initial protocol state from the given session. It calculates `handle_tls` with its state and the record input of the given session, together with the particular selection of protocol version, etc., resulting in an output state, potentially an output record, and potentially decrypted application data. It then compares equality of the output record and the given session. If successful, it uses the output state and next recorded input of the given session to evaluate `handle_tls` again, and repeats to the end of the trace. It thus terminates either when the entire trace has been accepted, which means success; or with a discrepancy between the nqsb-TLS specification and the recorded session, which means failure and needs further investigation. Such a discrepancy might indicate an error in the TLS stack being tested, an error in the nqsb-TLS specification, or an ambiguity in what TLS actually is.

A first test of this infrastructure was to use a recorded session of the change cipher spec injection (CVE-2014-0224): our test oracle correctly denied this session, identifying an unexpected message. We ran our test oracle and validated our 30 000 interoperability traces (see §7.1) and our Piñata traces (see §7.2), and also validated recorded TLS sessions between various implementations (OpenSSL, PolarSSL, nqsb-TLS) using tcpdump.

While running the test oracle we discovered interestingly varied choices in fragmentation of messages among existing stacks, which may be useful in fingerprinting.

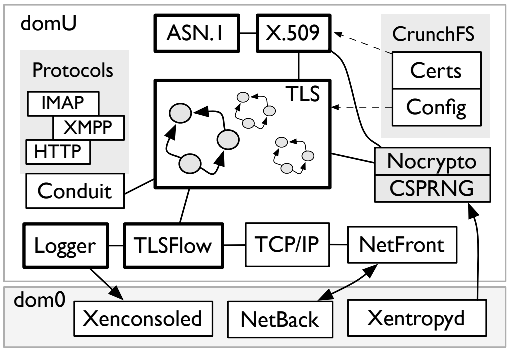


Figure 3: nqsb-TLS as a unikernel domU VM on Xen: a dom0 Xentropyd proxies host entropy, config and certificates are compiled in, various protocols run over TLS.

The test oracle opens up the prospect of extensive testing of the behaviour of different TLS implementations, especially if combined with automated test generation.

6 Using nqsb-TLS in applications

Another use of nqsb-TLS is as a TLS implementation in applications. We ported nqsb-TLS to two distinct environments and developed a series of applications, some for demonstration purposes, others for regular use.

6.1 Porting nqsb-TLS

To use nqsb-TLS as an executable implementation, we have to provide it with implementations of entropy and flow (see Figure 1), and an effectful piece of code that communicates via the network and drives the core.

We pay special attention to prevent common client bugs which arise from complexity of configuring TLS stacks and correspondingly large APIs. In each instance, there is only one function to construct a TLS configuration which can be turned into an I/O interface, the function does extensive validation of the requested parameters, and the resulting configuration object is immutable. This restricts potentially error-prone interactions that configure TLS to a single API point.

Unix Porting nqsb-TLS to Unix was straightforward; we use the POSIX sockets API to build a flow and `/dev/urandom` as the entropy source. The exposed interface provides convenience functions to read certificates and private keys from files, and analogues of `listen`, `connect`, `accept`, `read`, `write`, and `close` for communication.

MirageOS The MirageOS variant allows nqsb-TLS to be compiled into a unikernel VM (see Figure 3). It uses the MirageOS OCaml TCP/IP library [31] to provide the

I/O flow, which is in turn coupled to Xen device drivers that communicate with the backend physical network device via shared memory rings [44]. The logger outputs directly to the VM console, and the certificates and the secret keys are compiled into OCaml data structures at build time and become part of the VM image. A key challenge when running in a virtualised environment is providing a suitable entropy source [18], especially in the common case of a VM having no access to physical hardware. Since specialised unikernels have very deterministic boot-sequences that make sources of entropy even scarcer, we had to extend MirageOS and Xen to avoid cryptographic weaknesses [25].

One way in which we solve this is by relying on dom0 to provide cross-domain entropy injection. We developed *Xentropyd*, a dom0 daemon which reads bytes from `/dev/urandom` and makes them available to VMs via an inter-VM shared memory channel. The entropy device is plugged in as a standard Xen device driver via Xenstore [22], and MirageOS has a frontend library that periodically injects entropy into the nqsb-TLS CSPRNG.

To avoid being fully reliant on dom0, we implement additional entropy harvesting within the unikernel itself. We do this by trapping the MirageOS event loop and using `RDTSCP` instruction to read the Time Stamp Counter (TSC) register on each external event. This provides us with the unpredictability inherent in the ambient events. This source is augmented with readings from the CPU RNG where available: we feed the results of `RDSEED` (or `RDRAND`) instruction into the entropy pool on each event.

To make the RNG more resilient, we do extra entropy harvesting at boot time. Following Whirlwind RNG [18], we employ a timing loop early in the boot phase, designed to take advantage of nondeterminism inherent in the CPU by way of internal races in the CPU state. This provides an initial entropy boost in the absence of *Xentropyd* and helps mitigate resumption-based attacks [18].

In an ideal scenario the entropy would be provided through both mechanisms, but we expect the usage to rely on one or the other, depending on deployment: on an ARM board lacking high-resolution timing and CPU RNG, the user is likely to have control over the hypervisor and be able to install *Xentropyd*. Conversely, in commercial hosting scenarios where the assistance of dom0 might not be available but the extra CPU features are, we expect the user to rely on the internal entropy harvesting.

6.2 Applications

An example application using the Unix interface is the terminal-based instant messaging client *jackline* using XMPP. The XMPP protocol negotiates features, such as TLS, over a plaintext TCP connection. Jackline performs an upgrade to TLS via the `STARTTLS` mechanism before authentication credentials are exchanged. The Unix port of nqsb-TLS contains an API that supports upgrading an already established TCP connection to TLS. Jackline can use either of the authentication APIs (path and fingerprint validation) depending on user configuration.

tlstunnel also runs on Unix and accepts a TLS connection, forwards the application data to another service via TCP, similar to stud and stunnel. This has been deployed since months on some websites.

The Unix application *certify* generates RSA private keys, self-signed certificates, and certificate signing requests in PEM format. It uses nocrypto and X.509.

The OCaml Conduit library (also illustrated in Fig. 3) supports communication transports that include TCP, inter-VM shared memory rings. It provides a high-level API that maps URIs into specific transport mechanisms. We added nqsb-TLS support to Conduit so that any application that links to it can choose between the use of nqsb-TLS or OpenSSL, depending on an environment variable. As of February 2015, 42 different libraries (both client and server) use Conduit and its provided API and can thus indirectly use nqsb-TLS for secure connections. The OPAM package manager uses nqsb-TLS as part of its mirror infrastructure to fetch 2 500 distribution files, with no HTTPS-related regressions encountered.

7 Evaluation

We now assess the interoperability, security, and performance of nqsb-TLS.

7.1 Interoperability

We assess the interoperability of nqsb-TLS in several ways: testing against OpenSSL and PolarSSL on every commit; successfully connecting to most of the Fortune 500 web sites; testing X.509 certificates from ZMap; and by running a web server.

This web server, running since mid 2014, displays the live sequence diagram of a successful TLS session established via HTTPS. A user can press a button on the website which let the server initiate a renegotiation. The server configuration includes all three TLS protocol versions and eight different ciphersuites, picking a protocol version and ciphersuite at random. Roughly 30 000 traces were recorded from roughly 350 different client stacks (6230 unique user agent identifiers).

Of these, around 27% resulted in a connection establishment failure. Our implementation is strict, and does not allow e.g. duplicated advertised ciphersuites. Also, several accesses came from automated tools which evaluate the quality of a TLS server by trying each defined ciphersuite separately.

Roughly 50% of the failed connections did not share a ciphersuite with nqsb-TLS. Another 20% started with bytes which were not interpretable by nqsb-TLS. 12% of the failed connections did not contain the secure renegotiation extension, which our server requires. 5% of the failed traces were attempts to send an early change cipher spec. Another 4% tried to negotiate SSL version 3. 2.5% contained a ciphersuite with null (iOS6).

We parse more than 99% of ZMap's HTTPS (20150615) and IMAP (20150604) certificate repository. The remaining failures are RSASSA-PSS signatures, requiring an explicit NULL as parameter, and unknown and outdated algorithm identifiers.

This four-fold evaluation shows that our TLS implementation is broadly interoperable with a large number of other TLS implementations, which also indicates that we are capturing the de facto standard reasonably well.

Specification mismatches While evaluating nqsb-TLS we discovered several inconsistencies between the RFC and other TLS implementations:

- Apple's SecureTransport and Microsoft's SChannel deny application data records while a renegotiation is in process, while the RFC allows interleaving.
- OpenSSL (1.0.1i) accepts any X.509v3 certificate which contains either digitalSignature or keyEncipherment in keyUsage. RFC [15] mandates digitalSignature for DHE, keyEncipherment for RSA.
- Some unknown TLS implementation starts the padding data [29] (must be 0) with 16 bit length.
- A TLS 1.1 stack sends the unregistered alert 0x80.

7.2 Security

We assess the security of nqsb-TLS in several ways: the discussion of the root causes of many classic vulnerabilities and how we avoid them; mitigation of other specific issues; our state machine was tested [11]; random testing with the Frankencert [8] fuzzing tool; a public integrated system protecting a bitcoin reward; and analysis of the TCB size of that compared with a similar system built using a conventional stack.

Avoidance of classic vulnerability root causes In Sections 3 and 4 we described how the nqsb-TLS structure and development process exclude the root causes of many vulnerabilities that have been found in previous TLS implementations.

Additional mitigations The TLS RFC [15] includes a section on implementation pitfalls, which contains a list of known protocol issues and common failures when implementing cryptographic operations. nqsb-TLS mitigates all of these.

Further issues that nqsb-TLS addresses include:

- Interleaving of sub-protocols, except between change of cipher spec and finished.
- Each TLS 1.0 application data is prepended by an empty fragment to randomise the IV (BEAST).
- Secure renegotiation [40] is required.
- SCSV extension [35] is supported.
- Best practices against attacks arising from mac-then-encrypt in CBC mode are followed (no mitigation of Lucky13 [19])
- No support for export restricted ciphersuites, thus no downgrade to weak RSA keys and small DH groups (FREAK and Logjam).
- Requiring extended master secret [4] to resume a session.

State machine fuzzing Researchers fuzzed [11] nqsb-TLS and found a minor issue: alerts we send are not encrypted. This issue was fixed within a day after discovery, and it is unlikely that it was security-relevant.

Frankencert Frankencert is a fuzzing tool which generates syntactically valid X.509 certificate chains by randomly mixing valid certificates and random data. We generated 10 000 X.509 certificate chains, and compared the verification result of OpenSSL (1.0.1i) and nqsb-TLS The result is that nqsb-TLS accepted 120 certificates, a strict subset of the 192 OpenSSL accepted.

Of these 72 accepted by OpenSSL but not by nqsb-TLS, 57 certificate chains contain arbitrary data in X.509v3 extensions where our implementation allows only restricted values. An example is the key usage extension, which specifies a sequence of OIDs. In the RFC, 9 different OIDs are defined. Our X.509v3 grammar restricts the value of the key usage extension to those 9 OIDs. 12 certificate chains included an X.509v3 extension marked critical but not supported by nqsb-TLS.

Two server certificates are certificate authority certificates. While not required by the path validation, best practices from Mozilla recommend to not accept a server certificate which can act as certificate authority. The last certificate is valid for a Diffie-Hellman key exchange, but not for RSA. Our experimental setup used RSA, thus nqsb-TLS denied the certificate appropriately.

Exposure to new vulnerabilities Building nqsb-TLS in a managed language potentially opens us up to vulnerabilities that would not affect stacks written in C. Algorithmic complexity attacks are a low-bandwidth class of denial-of-service attacks that exploit deficiencies in many common default data structure implementations [10]. The modular structure of nqsb-TLS makes it easy to audit the implementations used within each component. The French computer security governmental office [37] assessed the security of the OCaml runtime in

2013, which lead to several changes (such as distinction between immutable strings and mutable byte arrays).

The Bitcoin Piñata To demonstrate the use of nqsb-TLS in an integrated system based on MirageOS, and to encourage external code-review and penetration testing, we set up a public bounty, the *Bitcoin Piñata*. This is a standalone MirageOS unikernel containing the secret key to a bitcoin address, which it transmits upon establishing a successfully authenticated TLS connection. The service exposes both TLS client and server on different ports, and it is possible to bridge the traffic between the two and observe a successful handshake and the encrypted exchange of the secret.

The attack surface encompasses the entire system, from the the underlying operating system and its TCP/IP stack, to TLS and the cryptographic level. The system will only accept connections authenticated by the custom certificate authority that we set up for this purpose. Reward is public and automated, because if an attacker manages to access the private bitcoin key, they can transfer the bitcoins to an address of their choosing, which is attestable through the blockchain.

While this setup cannot prove the absence of security issues in our stack, it motivated several people to read through our code and experiment with the service.

At the end of June 2015, there were 230 000 accesses to the website from more than 50 000 unique IP addresses. More than 9 600 failed and 12 000 successful TLS connections from 1000 unique IPs were present. Although we cannot directly verify that all successful connection resulted from the service being short-circuited to connect to itself, there have been no outgoing transactions registered in the blockchain. The breakdown of failed connections is similar to §7.1. We collected 42 certificates which were tried for authentication, but failed (not well formatted, not signed, not signed by our trust anchor, private key not present). A detailed analysis of the captured traces showed that most of the flaws in other stacks have been attempted against the Piñata.

Trusted computing base The TCB size is a rough quantification of the attack surface of a system. We assess the TCB of our Piñata, compared to a similar traditional system using Linux and OpenSSL. Both systems are executed on the same hardware and the Xen hypervisor, which we do not consider here. The TCB sizes of the two systems are shown in Table 2 (using `cloc`).

The traditional system contains the Linux kernel (excluding device drivers and assembly code), glibc, and OpenSSL. In comparison, our Piñata uses a minimal operating system, the OCaml runtime, and several OCaml libraries (including GMP). While the traditional system uses a C compiler, our Piñata additionally uses the OCaml compiler (roughly 40 000 lines of code).

	Linux/OpenSSL	Unikernel/nqsb-TLS	
Kernel	1600	48	(36)
Runtime	689	25	(6)
Crypto	230	23	(14)
TLS	41	6	(0)
Total	2560	102	(56)

Table 2: TCB (in kloc); portion of C code in parens

	nqsb-TLS	OpenSSL	PolarSSL
RSA	698 hs/s	723 hs/s	672 hs/s
DHE-RSA	601 hs/s	515 hs/s	367 hs/s

Table 3: Handshake performance of nqsb-TLS, OpenSSL and PolarSSL, using 1024-bit RSA certificate and 1024-bit DH group.

The trusted computing base of the traditional system is 25 times larger than ours. Both systems provide the same service to the outside world and are hardly distinguishable for an external observer.

7.3 Performance

We evaluate the performance of nqsb-TLS, comparing it to OpenSSL 1.0.2c and PolarSSL 1.3.11. We use a single machine to avoid network effects. In the case of nqsb-TLS, we compile the test application as a Unix binary to limit the comparison to TLS itself.

The test machine has an Intel i7-5600 Broadwell CPU and runs Linux 4.0.5 and glibc 2.21. Throughput is measured by connecting the command line tool *socat*, linked against OpenSSL, to a server running the tested implementation, and transferring 100 MB of data from the client to the server. This test is repeated for various transmission block sizes. Handshakes are measured by running 20 parallel processes in a continuous connecting loop and measuring the maximum number of successful connection within 1 second; the purpose of parallelism is to negate the network latency.

Throughput rates are summarized in Figure 4. With 16 byte blocks, processing is dominated by the protocol overhead. This helps us gauge the performance impact of using OCaml relative to C, as nqsb-TLS implements protocol logic entirely in OCaml. At this size, we run at about 78% of OpenSSL's speed.

At 8196 bytes, performance becomes almost entirely dominated by cryptographic processing. All three implementations use AES-NI, giving them roughly comparable speed. OpenSSL's performance lead is likely due to its extensive use of assembly, and in particular the technique of stitching, combining parts the of the cipher mode of operation and hashing function to saturate the CPU pipeline. PolarSSL's performance drop compared

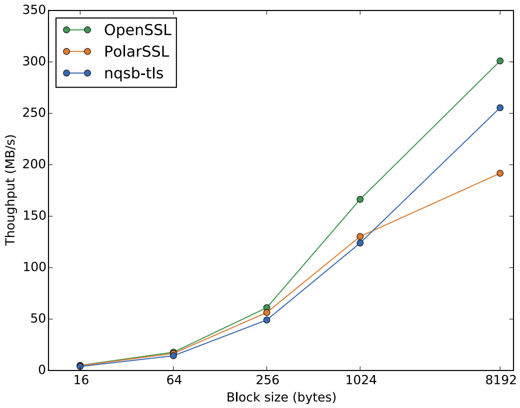


Figure 4: Scaling of throughput with application data size for nqsb-TLS, OpenSSL and PolarSSL, using AES_256_CBC_SHA.

to nqsb-TLS is likely a consequence of our usage of simple software pipelining in the AES-NI code. nqsb-TLS reaches about 84% of OpenSSL's speed in this scenario.

Handshake performance, summarized in Table 3, is roughly similar. We attribute OpenSSL's advantage to their use of C in the protocol handling, and PolarSSL's disadvantage to our use of faster bignum routines provided by GMP. The comparatively smaller cost nqsb-TLS pays for DH is a result of picking shorter exponents, matched to the security strength of the group modulus.

We ran miTLS 0.8.1 through Mono 4.0.1.44 on the same test machine. Using the bundled HTTP server, we achieve a peak throughput of 19 MB/s for a file transfer using the same cipher suite. As the Mono cryptography provider only contains C# AES implementations, we exclude this implementation from further analysis. We do note, however, that the throughput ratio between miTLS and OpenSSL is similar to the one its authors report [5].

The exact numbers are likely to vary with the choice of cipher suite, which places different weights on hashing and cipher performance, the CPU generation, which is utilised to a fuller extent by OpenSSL, and the testing scenario. The broad picture is that our usage of OCaml for all but the lowest-level cryptographic primitives is, in itself, not taking a prohibitive toll on performance.

8 Related Work

Security proofs Several research groups [36, 26, 24, 12, 38] have modelled and formally verified security properties of TLS. Because TLS is a complex protocol, most models use a simplified core, and formalising even these subsets is challenging work which is not very accessible to an implementer audience. Additionally, these models need to be validated with actual implementations, to relate to the de facto standard, but this is rarely done (to do so, some kind of trace checker or executable needs to be developed). Some of these models formalised the handshake protocol, but omitted renegotiation, in which a security flaw was present until discovered in 2009.

miTLS The miTLS [5] stack is developed in F7 with the possibility to extract executable F# code. It is both a formalisation of the TLS protocol and a runnable implementation. This formalisation allowed its developers to discover two protocol-level issues: alert fragmentation and triple handshake. As an implementation, it depends on the Common Language Runtime for execution, and uses its services for cryptographic operations and X.509 treatment (including ASN.1 parsing). In contrast, nqsb-TLS cannot be used for verifying security properties of TLS, but provides a test oracle and a fast runnable implementation which is easily deployable. It compiles to native code and implements the entire stack from scratch, making it self-contained. It can be used e.g. in MirageOS, which only provides the bare TCP/IP interfaces and has no POSIX layer or cryptographic services.

Language-oriented approach to security Mettler et al. propose Joe-E [33], a subset of Java designed to support the development of secure software systems. They strongly argue in favour of a particular programming style to facilitate ease of security reviews.

Our approach shares some of the ideas in Joe-E. By disallowing mutable static fields of Java classes, they effectively prohibit globally visible mutable state and enforce explicit propagation of references to objects, which serve as object capabilities. They also emphasise immutability to restrict the flows of data and achieve better modularity and separation of concerns.

The difference in our approach is that we use immutability and explicit data-passing not only on the module (or class) boundaries but pervasively throughout the code, aiming to facilitate code-review and reasoning on all levels. A further difference is that Joe-E focusses the proposed changes in style on security reviews only, aiming to help the reader of the code ascertain that the code does not contain unforeseen interactions and faithfully implements the desired logic. In contrast, we employ a fully declarative style. Our goals go beyond code review, as large portions of our implementation are accessible as a clarification to the specification, and we have an executable test oracle.

Finally, there is the difference between host languages. Java lacks some of the features we found to be most significant in simplifying the implementation, chiefly the ability to encode deeply nested data structures and traverse them via pattern-matching, and to express local operations in a pure fashion.

Brittle implementations of cryptography systems Schneier et al.'s work [43] discovered several root causes for software implementing cryptographic systems, which explicitly mentions incorrect error handling and flawed API usage. We agree with their principles for software engineering for cryptography, and extend this further by proposing our approach: immutable data, value-passing interfaces, explicit error handling, small API footprint.

TLS implementations in high-level languages Several high-level languages contain their own TLS stack. Oracle Java ships with JSEE, a memory-safe implementation. However its overall structure closely resembles the C implementations. For example, the state machine is built around accumulating state by mutations of shared memory locations, the parsing and validation of certificates are not clearly separated, and the certificate validation logic includes non-trivial control flow. This resulted in high-level vulnerabilities similar in nature to the ones found in C implementations, such as CCS Injection (CVE-2014-0626), and its unmanaged exception system led to several further vulnerabilities [34].

There are at least two more TLS implementations in functional languages, one in Isabelle [30] and one in Haskell. Interestingly, both implementations experiment with their respective languages' expressivity to give the implementations an essentially imperative formulation. The Isabelle development uses a coroutine-like monad to directly interleave I/O operations with the TLS processing, while the Haskell development uses a monad stack to both interleave I/O and to implicitly propagate the session state through the code. In this way both implementations lose the clear description of data-dependencies and strong separation of layers nqsb-TLS has.

Protocol specification and testing There is an extensive literature on protocol specification and testing in general (not tied to a security context). We build in particular on ideas from Bishop et al.'s work on TCP [6, 41], in which they developed a precise specification for TCP and the Sockets API in a form that could be used as a trace-checker, characterising the de facto standard. TCP has a great deal of internal nondeterminism, and so Bishop et al. resorted to a general-purpose higher-order logic for their specification and symbolic evaluation over that for their trace-checker. In contrast, the internal nondeterminism needed for TLS can be bounded as we describe in §5, and so we have been able to use simple pure functional programming, and to arrange the specification so that it is simultaneously usable as an implementation. We differ also in focussing on an on-the-wire specification rather than the endpoint-behaviour or end-to-end API behaviour specifications of that work. In contrast to the Sockets API specified in POSIX, there is no API for TLS. Every implementation defines its custom API, and many have a compatibility layer for the OpenSSL API.

9 Conclusion

We have described an experiment in engineering critical security-protocol software using what may be perceived as a radical approach. We focus throughout on structuring the system into modules and pure functions that can each be understood in isolation, serving dual roles as test-oracle specification and as implementation, rather than traditional prose specifications and code driven entirely by implementation concerns.

Our evaluation suggests that it is a successful experiment: nqsb-TLS is usable in multiple contexts, as test oracle and in Unix and unikernel applications, it has reasonable performance, and it is a very concise body of code. Our security assessment suggests that, while it is by no means guaranteed secure, it does not suffer from several classes of flaws that have been important in previous TLS implementations. In this sense, it is at least not quite so broken as some secure software has been.

In turn, this indicates that our *approach* has value. As further evidence of that, we applied the same approach to the *off-the-record* [7] security protocol, used for end-to-end encryption in instant messaging protocols. We engineered a usable implementation and reported several inconsistencies in the prose specification. The XMPP client mentioned earlier uses nqsb-TLS for transport layer encryption, and our OTR implementation for end-to-end encryption.

The approach cannot be applied everywhere. The two obvious limitations are (1) that we rely on a language runtime to remove the need for manual memory management, and (2) that our specification and implementation style, while precise and concise, is relatively unusual in the wider engineering community. But the benefits suggest that, where it can be applied, it will be well worth doing so.

Acknowledgements Parts of this work were supported by EPSRC Programme Grant EP/K008528/1 (REMS: Rigorous Engineering for Mainstream Systems) and by the European Unions Seventh Framework Programme FP7/20072013 under the User Centric Networking project (no. 611001). We also thank IPredator (`https://ipredator.se`) for lending bitcoins for our Piñata and hosting it, and the MirageOS team and the anonymous reviewers for their valuable feedback.

References

[1] Barnes, R., Thomson, M., Pironti, A., and Langley, A. Deprecating secure sockets layer version 3.0. RFC 7568, 2015.

[2] Beurdouche, B., Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P.-Y.,

and Zinzindohoue, J. K. A messy state of the union: Taming the composite state machines of TLS. In *Security and Privacy* (2015), IEEE.

[3] Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Pironti, A., and Strub, P.-Y. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *Security and Privacy* (2014), IEEE.

[4] Bhargavan, K., Delignat-Lavaud, A., Pironti, A., Langley, A., and Ray, M. Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension, Apr. 2015.

[5] Bhargavan, K., Fournet, C., Kohlweiss, M., Pironti, A., and Strub, P.-Y. Implementing TLS with verified cryptographic security. In *Security and Privacy* (2013).

[6] Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M., and Wansbrough, K. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and Sockets. In *SIGCOMM* (Aug. 2005).

[7] Borisov, N., Goldberg, I., and Brewer, E. Off-the-record communication, or, why not to use PGP. In *WPES* (2004), ACM.

[8] Brubaker, C., Jana, S., Ray, B., Khurshid, S., and Shmatikov, V. Using Frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *Security and Privacy* (2014), IEEE.

[9] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and Polk, W. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, May 2008.

[10] Crosby, S. A., and Wallach, D. S. Denial of service via algorithmic complexity attacks. In *USENIX Security* (2003).

[11] de Ruiter, J., and Poll, E. Protocol state fuzzing of TLS implementations. In *USENIX Security* (2015).

[12] Díaz, G., Cuartero, F., Valero, V., and Pelayo, F. Automatic verification of the TLS handshake protocol. In *Symposium on Applied Computing* (2004), ACM.

[13] Dierks, T., and Allen, C. The TLS Protocol Version 1.0. RFC 2246, Jan. 1999. Obsoleted by RFC 4346.

[14] Dierks, T., and Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346, Apr. 2006. Obsoleted by RFC 5246.

[15] Dierks, T., and Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, Aug. 2008.

[16] Dingledine, R., Mathewson, N., and Syverson, P. Tor: The second-generation onion router. In *USENIX Security* (2004).

[17] Dodis, Y., Shamir, A., Stephens-Davidowitz, N., and Wichs, D. How to eat your entropy and have it too – optimal recovery strategies for compromised RNGs. Cryptology ePrint Archive, Report 2014/167, 2014.

[18] Everspaugh, A., Zhai, Y., Jellinek, R., Ristenpart, T., and Swift, M. Not-so-random numbers in virtualized Linux and the Whirlwind RNG. In *Security and Privacy* (2014), IEEE.

[19] Fardan, N. J. A., and Paterson, K. G. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *Security and Privacy* (2013), IEEE.

[20] Ferguson, N., and Schneier, B. *Practical Cryptography*, 1 ed. John Wiley & Sons, Inc., 2003.

[21] Frost, R., and Launchbury, J. Constructing natural language interpreters in a lazy functional language. *The Computer Journal* (Apr. 1989).

[22] Gazagnaire, T., and Hanquez, V. OXenstored: An efficient hierarchical and transactional database using functional programming with reference cell comparisons. In *ICFP* (2009), ACM.

[23] Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D., and Shmatikov, V. The most dangerous code in the world: Validating SSL certificates in non-browser software. In *CCS* (2012), ACM.

[24] He, C., Sundararajan, M., Datta, A., Derek, A., and Mitchell, J. C. A modular correctness proof of IEEE 802.11I and TLS. In *CCS* (2005), ACM.

[25] Heninger, N., Durumeric, Z., Wustrow, E., and Halderman, J. A. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *USENIX Security* (2012).

[26] Jager, T., Kohlar, F., Schäge, S., and Schwenk, J. On the security of TLS-DHE in the standard model. In *CRYPTO* (2012).

[27] Kaloper-Meršinjak, D., Mehnert, H., Madhavapeddy, A., and Sewell, P. Supplementary material doi: 10.5281/zenodo.19160, June 2015.

[28] Kocher, P. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*. 1996.

[29] Langley, A. A TLS ClientHello padding extension, Feb. 2015.

[30] Lochbihler, A., and Züst, M. Programming TLS in Isabelle/HOL. In *Isabelle* (2014).

[31] Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., and Crowcroft, J. Unikernels: Library operating systems for the cloud. In *ASPLOS* (2013), ACM.

[32] Madhavapeddy, A., and Scott, D. J. Unikernels: The rise of the virtual library operating system. *Commun. ACM 57*, 1 (Jan. 2014), 61–69.

[33] Mettler, A., Wagner, D., and Close, T. Joe-e: A security-oriented subset of Java. In *NDSS* (2010).

[34] Meyer, C., Somorovsky, J., Weiss, E., Schwenk, J., Schinzel, S., and Tews, E. Revisiting SSL/TLS implementations: New Bleichenbacher side channels and attacks. In *USENIX Security* (2014).

[35] Moeller, B., and Langley, A. TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks. RFC 7507, Apr. 2015.

[36] Morrissey, P., Smart, N. P., and Warinschi, B. A modular security analysis of the TLS handshake protocol. In *ASIACRYPT* (2008).

[37] nationale de la sécurité des systèmes d'information, A. Étude de la sécurité intrinséque des langages fonctionnels, 2013.

[38] Paulson, L. C. Inductive analysis of the internet protocol TLS. *ACM Transactions on Information and System Security 2* (1999).

[39] Popov, A. Prohibiting RC4 Cipher Suites. RFC 7465, Feb. 2015.

[40] Rescorla, E., Ray, M., Dispensa, S., and Oskov, N. Transport Layer Security (TLS) Renegotiation Indication Extension. RFC 5746, Feb. 2010.

[41] Ridge, T., Norrish, M., and Sewell, P. A rigorous approach to networking: TCP, from implementation to protocol to service. In *FM* (May 2008).

[42] Saint-Andre, P., and Hodges, J. Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS). RFC 6125, Mar. 2011.

[43] Schneier, B., Fredrikson, M., Kohno, T., and Ristenpart, T. Surreptitiously weakening cryptographic systems. Cryptology ePrint Archive, Report 2015/097, 2015.

[44] Warfield, A., Hand, S., Fraser, K., and Deegan, T. Facilitating the development of soft devices. In *USENIX Annual Technical Conference* (2005).

To Pin or Not to Pin
Helping App Developers Bullet Proof Their TLS Connections

Marten Oltrogge, Yasemin Acar
DCSEC, Leibniz University Hannover
oltrogge,acar@dcsec.uni-hannover.de

Sergej Dechand, Matthew Smith
USECAP, University Bonn
dechand, smith@cs.uni-bonn.de

Sascha Fahl
FKIE, Fraunhofer
fahl@fkie.fraunhofer.de

Abstract

For increased security during TLS certificate validation, a common recommendation is to use a variation of pinning. Especially non-browser software developers are encouraged to limit the number of trusted certificates to a minimum, since the default CA-based approach is known to be vulnerable to serious security threats.

The decision for or against pinning is always a trade-off between increasing security and keeping maintenance efforts at an acceptable level. In this paper, we present an extensive study on the applicability of pinning for non-browser software by analyzing 639,283 Android apps. Conservatively, we propose pinning as an appropriate strategy for 11,547 (1.8%) apps or for 45,247 TLS connections (4.25%) in our sample set. With a more optimistic classification of borderline cases, we propose pinning for consideration for 58,817 (9.1%) apps or for 140,020 (3.8%[1]) TLS connections. This weakens the assumption that pinning is a widely usable strategy for TLS security in non-browser software. However, in a nominal-actual comparison, we find that only 45 apps actually implement pinning. We collected developer feedback from 45 respondents and learned that only a quarter of them grasp the concept of pinning, but still find pinning too complex to use. Based on their feedback, we built an easy-to-use web-application that supports developers in the decision process and guides them through the correct deployment of a pinning-protected TLS implementation.

1 Introduction

Android is the major platform for mobile users and mobile app developers. Many apps handle sensitive information and deploy the transport layer security protocol (TLS) to protect data in transit. Previous research uncovered security issues with TLS in mobile apps [7, 8, 9, 2, 22] that highlight that developers have problems with implementing correct certificate validation while users are challenged by TLS interstitials. Furthermore, the default TLS implementation on Android receives criticism [24, 18]: Adopted from web-browsers, default TLS certificate validation relies on a huge number of root CAs pre-installed on all Android devices [24]. Hence, all Android apps suffer from the same issues as web-browsers: A single malicious CA is able to conduct Man-In-The-Middle attacks (MITMAs) against all apps trusting the respective certificate. To make things even worse, Fahl et al. [8] uncovered that in 97% of all cases where developers implemented their own certificate validation strategy, they turned off validation entirely and left their apps vulnerable to MITMAs with arbitrary certificates, i.e. every active network attacker was able to attack successfully.

Pinning is often recommended as a general countermeasure to tackle the weakest link in the CA-based infrastructure [1, 14, 17, 8]. We use the term *pinning* in this paper to include both pinning the complete X.509 certificate or only the certificate's public key. Instead of trusting a large set of root CAs that come pre-installed with the operating system, software limits the set of certificates it trusts to *pins*, which can be single leaf certificates, single root CA certificates or a set of certificates. Pinning is a straightforward mechanism and its deployment does not require changes to the current CA infrastructure. However, pinning has not found widespread adoption yet. While limiting the number of trusted certificates drastically increases security, pinning doesn't come for free: Embedding trusted certificates into an app requires app updates whenever the pins change. Hence, the decision whether

[1] This smaller percentage in the optimistic case is caused by a different prevalence of third party library use.

pinning is applicable for a TLS connection is always a trade-off between increased security and maintenance effort that is not entirely under the control of an app developer: Whenever users do not update their app although the pins have changed, the app stops working and users might uninstall the broken app. Therefore, to pin or not to pin is a critical decision for app developers, which requires in-depth understanding of the mechanisms behind pinning and its implications on their apps. This is where our work comes in: To the best of our knowledge, we are the first to explore the applicability of pinning as an appropriate alternative certificate validation strategy for non-browser software. In this paper, we make the following contributions:

Status Quo We evaluate the status quo and analyze 639,283 Android apps to find that only 45 apps implement pinning.

Formalization Instead of intuitively recommending pinning, we formalize criteria that must be considered when the decision is made whether to pin or not to pin.

Implementation We apply static code analysis and program slicing to automatically assess those criteria and obtain an overview of the situation for single apps.

Evaluation We evaluate our criteria against a set of 639,283 Android apps for an overview of the applicability of pinning in the Android universe. We find that 223,655 apps establish TLS-secured connections to remote origins; 11,481 (5.13%) of these 223,655 apps are eligible candidates to implement pinning for one or more of their TLS connections.

App Updates Since new certificate pins need to be updated on the users' devices, the update speed is crucial. Therefore, we instrument telemetry data from a popular anti-virus software provider. We evaluate the update behaviour of 871,911 unique users from January 2014 to December 2014 and find that only 50% of the users update to a new app version within the first week after release.

Developer View Although pinning is only applicable in relatively few cases, the nominal-actual comparison leaves room for improvement. We therefore collected feedback from 45 developers of apps for which we would recommend pinning. We identified the developers' major issues with pinning and used their feedback as the foundation to build an easy-to-use web application that assists developers with securing their apps' TLS connections. We offer help on the decision whether to pin or not to pin and support the implementation of pinning with concrete suggestions and code examples.

Take-aways We formulate lessons learned during our evaluation to share them with the research community.

2 Background and Motivation

To establish secure TLS connections, *certificate validation* is important in order for communication partners to authenticate remote endpoints. Therefore, Android and most other non-browser software adopted the in-browser certificate validation strategy, i.e. applications trust a predefined set of *root certificate authorities* (root CAs). When attempting to establish a secure connection, the server provides the client with a *certificate chain*.

2.1 Default Certificate Validation

Certificate chain validation works as follows:

Chain of Trust Based on the certificate chain sent by the server, the client tries to build the *chain of trust* beginning at the server's leaf certificate up to one of the root CA certificates. Every certificate in the chain is checked for validity – i.e. it is not expired and it is signed by its immediate successor in the chain. The second last certificate in the chain must be signed by one of the root CAs installed on the user's device [3]; the last certificate in the chain belongs to the signing root CA.

Hostname Verification A certificate is bound to a certain identity – in this case a particular hostname or a set of hostnames. During *hostname verification*, a client verifies this identity by checking whether the hostname can be matched to one of the identities (i.e. CommonName, SubjectAltName [20]) for which the certificate was issued.

Further checks Complete certificate validation may include further checks, e. g. certificates have to be checked for *revocation* which is done via a certificate revocation list (CRL)[3] or Online Certificate Status Protocol (OCSP)[21].

2.2 MITM Attack

In a Man-In-The-Middle attack (MITMA), the attacker is in a position to intercept network communication. A passive MITMA can only eavesdrop on the communication, while an active MITMA can also tamper with the communication. Correctly configured TLS together with proper certificate validation is fundamentally capable of preventing both passive and active attackers from executing their attacks.

2.3 Alternative Validation Strategies

Over the years, alternative certificate validation strategies have come up [12, 26, 16, 15, 10, 17]. While they all provide approaches to check a certificate chain's validity, our paper focuses on *pinning* [17] as one of the most recommended alternative strategies for non-browser software.

2.3.1 Pinning

Pinning is a notion of certificate validation that uses existing knowledge about the network origin or the certification data presented to protect the TLS connection [17]. Required parameters to be pinned need to be available in an application before the TLS handshake happens. Pinning can be achieved based on different parameters (e.g. public key, whole certificate).
Pinning can also be applied to different subjects:
Leaf Pinning Leaf pinning includes pinning a single leaf certificate or its public key or a set of leaf certificates or their public keys and is the most rigorous way of pinning.
CA Pinning Certificate authority pinning effectively limits the number of trusted certificate authorities and allows for more flexible reconfiguration of deployed TLS certificates. This includes both intermediate and root CAs.

2.3.2 Trust on First Use (TOFU)

Another notion of pinning is *trust on first use* (TOFU). Instead of knowing the information to be pinned in advance, the first certificate (leaf or CA) seen for a TLS connection is stored locally on the client side and used to validate certificates for further connections. Hence, TOFU can be seen as a mix of conventional pinning (cf. Section 2.3.1) and the default validation strategy. TOFU is used to secure SSH connections and is an opt-in feature for HTTPS webservers for which it is called HTTP Public Key Pinning (HPKP [5]). As of today, HPKP has not found widespread adoption on the web [11].

2.4 Flaws in client implementations

The Android platform provides built-in functionalities for handling TLS and certificate validation based on the PKI without further configuration. It comes pre-loaded with an extensive truststore featuring 140+ root CAs [18]. Additionally, the Android framework enables developers to provide custom implementations for handling certificate validation. There are several reasons for developers to use custom implementations:

- Application developers might want to use a self-signed certificate either for testing, effort- or economical reasons;
- When a root CA is not in the system-wide list of trusted root CAs, a company might have an internal CA that issues certificates for use in intranet applications;
- Security can be enhanced by restricting reliance on the PKI to mitigate the exposure to weaknesses in the PKI
- and custom implementations are required to implement leaf or CA pinning.

2.5 Related Work

Recently, TLS has been subject to urgent flaws. Studies by Fahl et al. [7] and Georgiev et al. [9] uncovered a disastrous state of TLS-Code. Georgiev et al. [9] show numerous flaws in non-browser TLS-Code. Fahl et al. [7] identify numerous applications featuring erroneous custom TLS-code that potentially renders applications vulnerable to MITMAs. This is caused by incorrect implementations of custom certificate handling. Investigations on the reasons illustrate that especially developers without a security focus are unaware of the correct use of the APIs that Android provides: They carelessly incorporate code snippets from platforms such as `stackoverflow.com` [8]. In particular, self-signed certificates used during development lead to erroneous code that makes applications vulnerable by deactivating certificate validation completely.

Both of the above investigations [7, 9] focused on the difficulties experienced by developers with the implementation of correct certificate validation. However, they discuss possible countermeasures rather sketchily and do not evaluate the applicability of pinning for non-browser software.

To improve the state of erroneous certificate handling and to mitigate the threats, Fahl et al. [8] suggest to completely disallow custom code. Instead, they propose a framework that allows to realize common use-cases (e.g. self-signed certificates for testing, pinning, default warning messages, etc.) by configuration without the developer writing any TLS-related code. Tendulkar and Enck [23] suggest a similar approach. However, although both approaches try to improve the usability of certificate handling for software developers, they do not support developers in the decision process of whether pinning is a recommendable certificate validation strategy.

3 An Exit Strategy

Previous work shows that TLS is complex and error-prone. Studies imply that the implementation of client code for correct certificate validation is hard for software developers. The general trust model received a lot of criticism over the last years and alternative solutions have not found widespread adoption.

As a general solution for both problems, pinning is often advocated as a secure and reliable alternative to the default trust model for non-browser software [17, 14, 1, 8]. Pinning can serve two purposes. First, it can mitigate the risk of MITMAs as it strengthens the validation process. Secondly, pinning allows to overcome limitations of the current CA infrastructure, e.g. by allowing self-signed certificates.

We challenge the recommendation to use pinning for non-browser software and conduct a deep analysis on the root causes that hinder its widespread adoption in non-browser software.

3.1 Pinning in Android Apps

Before evaluating the status quo, we give a brief overview of how TLS pinning can be implemented in Android apps. In general, Android apps can establish low-level TLS connections via an `SSLSocket` or an `HTTPSUrlConnection` object. Using pinning in these cases requires the developer to implement a custom version of Android's `TrustManager` interface with an appropriate `checkServerTrusted` method. This method has to check whether the certificate sent by the remote server matches one of the given pins. Another option is to use a custom `KeyStore` in which developers can store their own certificates. Many Android applications do not make use of such lower level APIs but use a `WebView` to display HTML directly. Sadly Android does not provide an API to implement pinning for `WebViews`[2]. As a workaround, developers can download all HTML/-JavaScript via the low-level `HTTPSUrlConnection`, store it locally and only use the `WebView` for rendering. However, this is a clear shortcoming of Android's API and makes the implementation of pinning unnecessarily hard.

3.2 Status Quo

For a better understanding of the current adoption of TLS pinning in Android apps, we evaluated custom certificate verification strategies for 639,283 Android apps. Therefore we used `MalloDroid` [7] and extended the classification feature of custom verification implementations. We focused on both customized implementations of the `TrustManager` interface and the usage of a custom `KeyStore` for TLS certificates.

Whenever we found that a `KeyStore` object was created, we conducted a reachability analysis [13] for this object. For objects that were reachable from an app's entry point, we assume that this app uses pinning. Next, we extracted the keystore file that was loaded to check whether leaf or CA certificates were pinned. We found 21 apps that implement pinning using the keystore method. 13 of these apps pin a leaf certificate, while 8 of them pin a custom root CA certificate.

Whenever we found a `TrustManager` implementation, we checked whether the `chain` parameter of the `checkServerTrusted` method was accessed by the implementation. Implementations that do not use this parameter do not verify the remote origin's certificate chain and hence were removed from further analyses. In a second step we conducted a reachability analysis for implemented `TrustManager` objects and removed all implementations that were not reachable from an app's entry point. We found custom `TrustManager` implementations in 42,902 apps and could remove 42,042 apps from the list since they either implemented bypassing `TrustManagers` or were not reachable. The remaining 858 apps implemented 189 different `TrustManagers`. We compared these implementations with the list provided by Fahl et al. [8] and could filter out 124 implementations that basically add logging for certificate validation. We manually reviewed the remaining 65 implementations and found that 13 implemented pinning. Overall, these implementations were used by 24 apps.

Altogether, of the 639,283 apps in our data-set, 45 implement pinning. These numbers confirm findings already reported by Fahl et al. [8].

4 Classification Strategy

The decision whether or not a TLS connection should be secured by pinning depends on multiple factors and is not trivial in many cases.

Furthermore, whenever we cannot reliably identify the origin string for a TLS connection endpoint, we cannot assess whether pinning would be a reasonable validation strategy. Therefore, we report our results for two different scenarios:

[2] `https://code.google.com/p/android/issues/detail?id=76501`

Conservative We report numbers for a conservative scenario. Whenever we cannot identify the origin string for a TLS connection endpoint, we assume that pinning is unfeasible. This covers most of our results.

Optimistic For some of the cases where we could not successfully identify origin strings, pinning could be applied under certain circumstances. These cases are treated differently for the optimistic scenario as detailed in Section 6.

For our classification, we consider the following properties as high level indicators:

Prior Knowledge of the Target Origin Prior knowledge about the target origin is vital: Pinning is only feasible if the developer of an app is able to hard-code target origins into their app. This includes adding target origins at compile time as well as via configuration files before or at run-time. Whenever target origins depend on user- or external app input – e.g. via an `Intent` – at run-time, we consider pinning as not feasible, since web-browsers do not automatically pin certificates for websites for the same reasons.[3]. Automatically pinning previously unknown origins would increase the danger of failed TLS handshakes due to substituted certificates and would decrease acceptance of pinning for both developers and app users.

In the *conservative* scenario, we recommend the default validation strategy for all connections where the origin depends on external input. However, some of these connections can be pinned in the *optimistic* scenario (cf. Section 6).

Ownership of Relevant API Calls App development consists of writing one's own code and embedding external libraries. All source code that was written by the developer or the developer's company is considered owned by the developer. API calls required for certificate validation during the establishment of TLS-secured network connections are *relevant* for this. Relevant API calls might be a `HTTPSUrlConnection` or an `SSLSocket`. Whenever ownership of relevant API calls is given, pinning might be feasible. Library developers do not own their code when it is included in other apps. Therefore, we do not recommend pinning for library code. Library developers cannot control when app developers update their libraries, while app developers can hardly influence whether library developers keep their certificate pins up to date.

For both the *conservative* and the *optimistic* scenario pinning is not recommended in case API ownership is not given.

TLS Certificate Configuration Responsibility Being responsible for the TLS certificate configuration as well as being the owner of relevant API calls eases the coordinated deployment of pinning in apps. In case developers or their companies have control over the TLS certificate configuration of origins used in apps, both the certificate pins in apps and the corresponding server configurations can be coordinated. In these cases, pinning is feasible. Whenever apps communicate with public origins, such as public API interfaces or websites, pinning cannot be recommended. Certificate configurations can change frequently and the responsible administrators only rarely announce them in advance. Unplanned certificate changes can lead to failing TLS handshakes and are therefore unacceptable.

For both the *conservative* and the *optimistic* scenario pinning is not recommended in case the developer is not responsible for TLS certificate configuration.

4.1 Possible Recommendations

The above criteria build the foundation for the decision whether pinning is applicable for a TLS connection in a given app. The classification algorithm recommends one of the following strategies:

Leaf Pinning Leaf pinning limits the number of trusted certificates to the server's leaf certificate/public key (cf. Section 2.3.1).

CA Pinning CA pinning effectively limits the number of trusted CAs (cf. Section 2.3.1). We treat conventional pinning (cf. Section 2.3.1) and TOFU (cf. Section 2.3.2) equally, since both provide a similar level of security (cf. Section 2.3.1) and require the same maintenance overhead from an app developer's point of view.

Default Whenever none of the above criteria applies, pinning is not a recommended strategy and should not be implemented. This also accounts for cases where external input – e.g. user input in an address bar or `Intent` input – influences a TLS connection.

[3] Administrators can configure HTTP Public Key Pinning (HPKP) to pin TLS certificates in modern web-browsers. However, this involves heavy manual configuration work on the server side and does not happen automatically (cf. Section 2).

4.2 Classification Details

Algorithm 1 illustrates the classification process we apply to decide whether pinning is an advisable verification strategy. In the initial state, the default strategy is assumed to be the right choice for a TLS connection, since we do not know yet if pinning is recommendable. First, we check whether the TLS connection is established within a third party library. In this case classification ends with the recommendation to use the default strategy, since ownership of the relevant API is not guaranteed.

Second, we check whether the remote origin depends on user input, other external sources such as `Intents`, or may be configured via a configuration file. In these cases we recommend the default strategy, since control over the remote origin is not guaranteed.

Third, we check whether the TLS connection's remote endpoint is a popular origin; in this case, classification ends with recommending the default strategy, since the TLS configuration of the remote origin is probably not under the developer's control. If we assume that a remote origin is probably under control of the developer, as no other app accesses it, the classification continues: We check whether the origin's certificate was issued by a valid CA or is self-signed. For self-signed certificates and certificates that were issued by a valid CA, we recommend leaf certificate pinning. For certificates issued by an untrusted CA, we recommend CA pinning, since the custom CA is probably under control of the developer.

4.3 Challenges

For our strategy classification, we apply static code analysis on a large set of Android apps. To work as efficiently as possible, we identified multiple challenges:

4.3.1 Relevant API Calls

First, we identify relevant API calls, which means taking remote origins as parameters and establishing a TLS-secured connection between the app and the origin. The official Android API documentation identifies relevant API calls in the packages presented in Table 1.

These API calls are the most low-level calls in the API and they implicitly include higher level APIs such as the `HTTPSUrlConnection`.

Package name
org.apache.http.client.methods.*
org.apache.http.HttpHost
android.webkit.WebView.loadUrl
android.webkit.WebView.loadDataWithBaseURL
android.webkit.WebView.loadData
android.webkit.WebView.postUrl
android.net.http.AndroidHttpClient
java.net.Url

Table 1: Relevant API Calls.

4.3.2 Embedded Static TLS Origins

As described above, knowing remote origins in advance is crucial for pinning. At this point, we focus on extracting whether a remote origin is embedded in an app or depends on user input or is injected via an external interface such as an `Intent`. This information is supplied via parameters to relevant API calls. Although these mainly refer to `String` values, the object-oriented and Java-based nature of the Android platform introduces complexity:

- `Strings` may not be constant values but can be composed of numerous substrings. We identify concatenation of Strings, formatting of `Strings` and platform-specific APIs for building URIs as relevant. Therefore, we statically backtrack these and reproduce `String` values.
- Values can stem from variables or may be return values of method calls. Therefore, we account for intra-application method calls as well.
- Values that stem from `Resources`, `Properties` or `Preferences` hint at configuration parameters.
- Origins can be input parameters for application entrypoints. Entrypoints are parts of an application that allow either other app components, external apps or users to interact with an app. Android application entrypoints are `Activities`, `Services`, `Receivers`, `Intents` and `Bundles`. UI-Components in `Activities` hint at user input.

4.3.3 API Call Ownership

API call ownership is a requirement for pinning. To identify whether an app developer holds ownership of relevant API calls, we must distinguish relevant API calls that are accessed by third party libraries and relevant API calls accessed by code that was

Algorithm 1: The Classification Process.

```
for $r \in results$ do
    $dependencies \leftarrow dependencies(r)$;
    set strategy as default;
    if $dependencies \neq \emptyset$ then
        if $\exists d \in dependencies | d \in \{exposed\,intent, UI-Component\}$ then
            continue;
        else if $\exists d \in dependencies | d \in \{unexposed\,intent, variable, configuration\}$ then
            continue;
    if not callInLibrary(r) and not isPublicHost(r) then
        $host \leftarrow host(r)$;
        $schema \leftarrow schema(r)$;
        $cert \leftarrow cert(r)$;
        $dependencies \leftarrow dependencies(r)$;
        if $isUnderControl(host)$ then
            if $signedByUntrustedCA(cert)$ then
                mark for CA pinning;
            else if $validCertChain(cert)$ or $isSelfSigned(cert)$ then
                mark for leaf pinning;
```

written by the app developer (we call this *custom code*). Whenever we find relevant API calls in code that is shared between multiple apps by different authors, we assume a library that is not under control of an app's developer.

4.3.4 Origin Exclusivity

Whenever the TLS configuration of a remote origin is not in the range of influence of the developer, pinning is not advisable. We classify origins that are shared between multiple apps' authors and connections that access public origins as not under control of the developer.

5 Implementation Details

To decide whether pinning for a TLS connection is advisable and to address the above challenges (cf. Section 4.3), we implement our classification strategy in a multi-step process. We extend[4] the MalloDroid tool [7] and execute the following steps:

1. Disassemble a given Android application to gain access to the application's code and call graph (cf. Section 5.1).
2. Identify relevant API calls – i.e. API calls that implement TLS connections in apps (cf. Sections 5.2 and 4.3.1).
3. Extract information for remote origins applying program slicing (cf. Sections 5.3 and 5.3.1).
4. Determine whether API and/or origin ownership for relevant API calls is given and decide which certificate validation strategy suits best (cf. Sections 4.2 and 5.4).

5.1 Disassembly

In Step 1, we use androguard [4] to disassemble apps and construct call graphs for further processing.

5.2 Relevant API Calls

In Step 2, the call graphs were used to identify apps that make use of API calls in which origin information for establishing TLS-enabled connections is specified (cf. Section 4.3).

We consider API calls as *relevant* if they are used during the process of TLS connection establishment (cf. Table 1).

5.3 Program Slicing

In Step 3, we apply backwards program slicing [25] to collect method parameters which we subsequently call *slicing criteria*. We focus on slicing criteria that

[4] Our MalloDroid extension is available at `https://github.com/sfahl/mallodroid`.

represent remote origins which are used as input for relevant API calls, i. e. we are backtracking parameters that are URLs or hostnames for TLS connections.

Our approach is similar to the one applied by Poeplau et al. [19], who apply a backward slicing algorithm to identify security issues with dynamic code loading in Android apps.

In contrast to backwards slicing single and fixed method parameters, our targets are network origins. A network origin `String` can be a composition of multiple substrings. We take this fact into account by applying backwards slicing for multiple parameters. After backtracking, we join these (multiple) substrings to one origin string whenever possible. To break cycles, we stop the slicer after 80 iterations, which guarantees that the algorithm terminates and also makes sure we do not lose data.

5.3.1 Extracting Origin Strings

Origin strings can be compositions of multiple substrings (e.g. `https://` and `www.example.com` and `:443`). Thus, reconstructing an origin string might require combining multiple sliced substrings. Therefore, after program slicing, we apply a combination of backward and forward analysis. Backwards analysis is used for backtracking constant register values while forward analysis determines calls of a `String` instance. Both, back- and forward analyses are applied multiple times successively as long as we find new substrings that are part of a final origin string.

Algorithm 2 outlines pseudocode for handling `StringBuilder` objects. The algorithm makes use of the following functions:

methodsOnInstance returns a list of all method invocations on an instance (e.g. calls to `toString`, `append` or the constructor for `StringBuilder` objects).

backtrack applies the actual backtracking techniques to gather all substrings for the origin string composition.

add adds one `originSubstring` to the array of `originSubstrings` that make the origin string we are looking for.

join merges all `originSubstrings` to get the final origin string that is represented by the given `StringBuilder` object.

1. Identify instructions indicating the instantiation of a `StringBuilder` object (i.e. a `new-instance` instruction referring to the `StringBuilder` constructor) and store them in `sbos`.

Algorithm 2: StringBuilder Analysis

```
Data: stringBuilderObjects sbos
Result: new originSubstring O
begin
    for $sb \in sbos$ do
        methodInvocations ← methodsOnInstance(sb);
        originSubstrings ← ∅;
        origin ← null;
        for mi in methodInvocations do
            if isAppend(mi) then
                originSubstring ← backtrack(mi.regs[1]);
                if $originSubstring \neq null$ then
                    add(originSubstrings, originSubstring);
            else if isToString(mi) then
                break;
        join(O, originSubstrings);
```

2. Find all method invocations for each `StringBuilder` object `sb`;

3. For calls of the constructor or the `append()` method, backtrack the register value for the `String` parameter `originSubstring` and add it to all `originSubstrings`.

4. Stop on calls of the `toString` method and join all collected `originSubstrings` to a new `originSubstring`.

Similarly, we support `String` concatenation, formatting `Strings`, `UriBuilder` instances, Android `UI-Components`, `Intents`, `Bundles`, `Properties`, `Preferences` and `Resources`. For `Intents` and `Bundles` we identify the source in order to determine whether the corresponding component is exposed externally, e.g. via a `Service`.

5.4 Decide on Validation Strategy

In the final Step 4, we assess whether API and/or origin ownership is given (cf. Section 4). For pinning candidates, X.509 certificate information is collected and a decision for or against pinning is made (cf. Section 4.2).

5.5 Limitations

The described approach is limited in multiple ways. We decided to reverse engineer a large sample of free Android apps and analyze the resulting code. This limits the analysis compared to the analysis of original source code, e.g. we lose variable names and cannot preclude obfuscation. This is however the state of the art for large scale app analyses, since reaching out to developers and asking for source code does not scale well. We apply static code analysis and program slicing to determine the best certificate validation strategy to secure a TLS connection. Our approach does not consider native code in Android apps. We cannot track potential TLS connections in code that was dynamically loaded or when obfuscation was applied.

Since we apply static code analysis techniques, we might report some false positives: Some of the TLS connections we found and identified as being reachable code might not be used during real application usage. However, this is not a specific limitation of our work, but a general limitation of static code analysis.

We might also report some false negatives: Due to the strategy to classify origins to which apps developed by different developers connect as "not under control of the developer", we might miss the scenario that one company has several distinct developers write apps for them that all access the same domain. However, there exist no criteria to distinguish these cases from the common scenario that multiple apps by different developers accessing the same domain means that the domain is not under control of the developers. Therefore, these cases are included in the group of public origins for which we do not recommend pinning.

6 Evaluation

We applied the classification algorithm (cf. Section 4) to a set of 639,283 Android applications we downloaded from Google Play in October 2014. Our data extraction showed that of these apps, 573,258 implemented network connections.

In the following, we report details of our automated large-scale analysis. We report our results on a per-connection base (see Figure 2) as well as on a per-app base (see Figure 1), where an app is counted as eligible for pinning if at least one of its connections can be pinned. We distinguish between a conservative and an optimistic strategy rating (cf. Section 4). Altogether we found 20,020,535 calls to network related API calls (cf. Table 1). For these calls we tried to identify the origin strings. We could identify 1,062,810 calls as TLS connections due to the fact that the corresponding origin string's scheme was HTTPS. 2,420,104 connections were identified as plain HTTP, while 16,537,621 connections did not have a hard-coded origin string in the app's code. Hence, for 81% of all connections, it was not directly possible to determine whether TLS was used. However, a deeper analysis based on our classification criteria (cf. Section 4.2) gives detailed insights into the applicability of pinning. Table 2 gives an overview of the results.

Overall	**Con.**	**Apps**
Hard-coded HTTPS Origin	1,062,810	229,317
Hard-coded HTTP Origin	2,420,104	414,194
Non-hard-coded Origin	16,537,621	553,399
	20,020,535	573,258

Third Party Libraries	**Con.**	**Apps**
Hard-coded HTTPS Origin	917,567	203,159
Hard-coded HTTP Origin	1,659,933	310,331
Non-hard-coded Origin	14,564,581	512,055
	17,142,081	517,790

Custom Code	**Con.**	**Apps**
Hard-coded HTTPS Origin	145,243	48,755
Hard-coded HTTP Origin	760,171	184,184
Non-hard-coded Origin	1,973,040	246,636
	2,878,454	299,863

Table 2: Distribution of Network API Calls.

6.1 Library Code

The majority of network connections we identified were made in third party library code, i.e. users include third party libraries to make use of external functionality. Such connections can include both plain and TLS-protected connections. As described in Section 4.2, we recommend not to use pinning for those TLS connections, as API ownership is not given. Of the 20,021,137 TLS connections we could identify, we found that 17,142,081 (85.6%) connections are embedded in third party libraries.

Table 3 gives an overview of the top 10 third party libraries we found in our data-set.

Most of the identified libraries belong to ad networks (e.g. `com.google.ads.*`), crash reporting tools or app building kits (e.g. `org.apache.cordova.*`) that establish network

Library	Connections
com.google.ads.*	2,535,020
org.apache.cordova.*	1,145,108
com.qbiki.*	977,298
com.millennialmedia.*	730,408
com.facebook.*	551,373
com.Tobit.*	488,143
com.inmobi.*	352,855
com.flurry.*	340,929
com.startapp.*	276,988
com.adsdk.*	234,130

Table 3: Top 10 Third Party Libraries.

	Con.	Apps	Conservative	Optimistic
Shared Origin	99,996	40,691	-	-
Unique Origin	45,247	11,547	📌	📌
	145,243	45,549		

(a) HTTPS Origins in Custom Code.

Internal Intent	294,846	81,040	-	📌
Public Intent	14,599	8,268	-	-
Parcel	5,729	2,883	-	-
UI Component	31,266	18,766	-	-
Resource	124,356	32,113	-	📌
(Shared) Preference	87,051	31,975	-	📌
Variable	432,985	119,406	-	📌
JSON	96,438	32,200	-	📌
	1,973,040	326,651		

(b) Dynamic Origins in Custom Code.

Table 4: Origins in Custom Code – Connections marked with 📌 can be pinned.

connections without any interaction with an app's custom code.

We found the `AndroidPinning` library[5] to be the only library that supports pinning as a security feature out of the box. However, in our data-set we found only 14 apps that made use of this library (cf. Section 3.2).

6.2 Custom Code

Next, we identified network connections that were established in code that was actually written by apps' developers, i.e. network-related API objects were not instantiated within third party libraries.

We found 2,878,454 connections in custom code of which we identified 145,243 as TLS connections. 48,755 apps implemented hard-coded TLS origins as parts of their custom code. Based on the type of the deployed certificate and depending on whether the origin is shared, we evaluated which of these TLS connections are candidates for pinning.

For 1,973,040 of the connections that were implemented as part of apps' custom code, we could not identify hard-coded origins in apps. Those connections could be either HTTP or HTTPS and depend on further input available only at run-time, e.g. user input or Intents. For these connections, we consider a conservative as well as an optimistic scenario. Based on different assumptions, these scenarios allow us to estimate the applicability of pinning in Android apps.

6.2.1 Hard-coded Origins

Overall, we found 145,243 hard-coded HTTPS connections and 760,171 hard-coded HTTP connections in our data-set. For the HTTPS connections, we collected further information such as the number of connections that connect to the same origin and the origin's X.509 certificate whenever possible.

The 145,243 TLS connections we found included connections to 11,203 different TLS-enabled remote origins.

To investigate whether pinning is the recommendable validation strategy, it is important to know if an origin is shared between multiple apps authored by multiple developers or used by apps of a single developer only (cf. Section 4).

Shared Origins 1,301 of the extracted 11,203 hosts were present in multiple apps that were authored by multiple developers. We assume these hosts not to be under the control of an app's developer and hence recommend the default certificate validation strategy, since host-ownership is not given (cf. Section 4). This affects 99,996 of the TLS connections we identified in our data-set (cf. Table 4a).

Table 5 lists the top 10 shared origins in custom code we found in our data-set.

Unique Origins 6,012 origins were unique to one single app while 3,890 origins were included in no more than five apps owned by a single developer. Hence, we assume 9,902 origins to be under the con-

[5] `https://github.com/moxie0/AndroidPinning`

Hostname	Con.	Apps
graph.facebook.com	220,697	111,559
m.facebook.com	110,903	104,745
www.googleapis.com	30,101	20,120
bugsense.appspot.com	18,402	18,285
www.starbucks.com	32,063	16,029
www.facebook.com	39,969	13,923
docs.google.com	29,240	11,872
mobileclient.paypal.com	39,214	10,963
api.twitter.com	34,641	10,551
svcs.paypal.com	38,175	9,999

Table 5: Top 10 Remote Origins.

trol of a single developer each. We recommend pinning for these apps and their connections, since both host- and code-ownership are given (cf. Section 4). This affects 45,247 TLS connections in our data-set (cf. Table 4a).

To determine whether leaf pinning or CA pinning is the right choice, we analyzed the deployed certificates for the respective origins. Overall, we gathered 7,941 unique certificate chains. We used Androids pre-installed root CA certificates and Androids certificate validation strategy to verify the validity of the origins' certificates and found that 7,177 of all chains could be verified successfully, while verification failed for 764 chains. Of these non-validating certificate chains, 182 certificates were self-signed; 170 certificates were issued by an unknown CA; 335 certificates were already expired and for 160 certificates hostname verification failed. Table 6 gives an overview of the chains and the affected connections and apps in our data-set.

Verification Result	Con.	Apps
Chain Ok	40,433	10,176
Self-Signed	1,966	486
Custom CA	269	81
Expired	1,709	546
Hostname Mismatch	870	258
	45,247	11,547

Table 6: X.509 Certificates Statistics.

We recommend pinning for all of these connections (cf. Section 4). We recommend leaf pinning for the connections that use a self-signed certificate and CA pinning for all other connections (cf. Table 7).

Type	Connections	Apps
Leaf Pinning	44,978	11,247
CA Pinning	269	81

Table 7: Pinning Statistics (Conservative).

6.2.2 Dynamic Origins

While the unique origins in custom code are good candidates for pinning, the majority of origins for connections in custom code were not hard-coded into the apps at compile time (cf. Table 4). These connections' origins can depend on different external factors such as Intents, UI components etc.

For these connections, we distinguish two scenarios:

Conservative In the conservative scenario, we assume that connections that use dynamic origins cannot be pinned. This assumption prevents us from over-reporting the applicability of pinning, but probably underestimates its applicability as well. Assuming this scenario, 45,247 of the 1,062,810 TLS connections we found in our data-set would be good pinning candidates, which makes up 4.25%.

Optimistic In this scenario, we assume some of the 1,973,040 dynamic origins connections eligible for pinning. We still assume that connections that get their input from Public Intents, Parcels and UI Components are no good pinning candidates, since the app developer probably does not have control over the actual origin strings values. That leaves some of the remaining 1,921,446 connections that depend on dynamic origin strings eligible for pinning. We assume that – as for the custom code with hard-coded origins – 16% of all connections are HTTPS connections (cf. Table 4), which leaves us with 307,431 HTTPS connections. If we again assume that, like for the hard-coded custom code origins, 31.1% of the HTTPS connections can be pinned (cf. Table 4), we can recommend 95,611 connections for pinning. In combination with the 45,247 connections from the conservative scenario, we recommend to pin 140,858 connections in the optimistic scenario. However, while only 86.33% of all known HTTPS connections are implemented in third party library code, but 88.07% of our *assumed* HTTPS connections happen via third library code, the connections we optimistically recommend for pinning make up only 3.8% of all (assumed and definite) HTTPS connections (as opposed to 4.25% for definite HTTPS connections). We can optimistically pin 140,858 connections as opposed to only 45,247 connections in the conservative case. Naturally, we are unable to specify which pinning strategy we would suggest, but extrapolating from the conservative scenario i.e. applying the percentages of which pinning strategy is applicable in the small conservative data-set to the larger data-set, 140,020 cases would be eligible for leaf pinning, while we would recommend CA pinning for 838 connections.

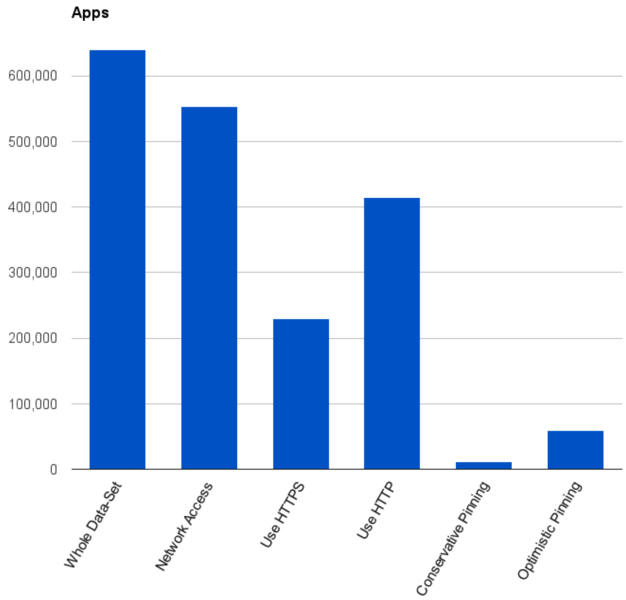


Figure 1: Statistics and Classification Results for Apps; *Network Access* includes apps with custom-coded, library- and dynamic HTTPS and HTTP connections.

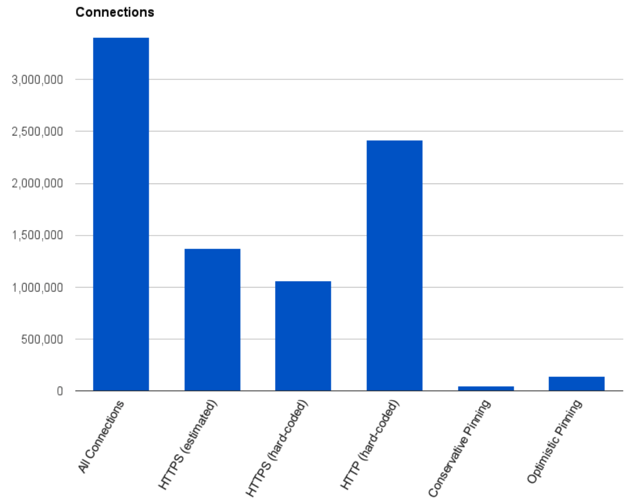


Figure 2: Statistics and Classification Results for Connections.

6.3 Update Frequencies

In addition to the strategy classification, a crucial requirement for pinning to work properly is the possibility to deploy quick updates for apps to distribute new pins. The applicability of pinning for Android apps in general depends on how quickly developers can push new certificate pins to their users' devices. Hence, we were interested in assessing the frequency for app updates.

Although newer Android versions have an auto update feature for apps, this feature is opt-in. Furthermore, the default is that app updates are only downloaded in case a device is connected to a WiFi. Hence, even auto-updates are not guaranteed to happen instantly.

Information about update frequencies is not easily accessible via Google Play. To gain insights into users' update behaviour, we cooperated with a popular anti-virus software vendor for Android with an install base of 5 million devices. Our cooperation partner runs a telemetry program and gathers user data for all users that participate in that program. From January 2014 to December 2014, we collected data for 784,721 unique apps and 871,911 unique users. The 871,911 users that participated in the telemetry program and gave their consent to anonymously analyze the data for our research yield the following meta information:

Pseudonym We assigned a 256-bit random pseudonym to each device to protect the users' privacy. The pseudonym did not reveal any private information.

DeviceInfo We collected manufacturer- and device model information as well as the installed Android version.

DeviceFlags We gathered three different flags for every device: (1) Whether developer options were enabled, (2) whether app installs from untrusted sources were allowed and (3) whether USB debugging was enabled.

PackageInfo For every (pre-)installed app we gathered the package name and version code.

PackageHashes For every (pre-)installed app we gathered SHA256 checksums of the packages and their corresponding signing keys.

Timestamps We gathered timestamps for when we saw an app version installed on a device.

Our interest focused on third party apps such as facebook or games, as these apps get updates pushed via Android's default update mechanism. We excluded Google apps and device vendor specific apps from our analysis, since these can be updated through special update channels provided by Google or the device vendor. We identified Google apps and device vendor apps based on the keys they have been signed with [6].

For third party apps, we found that on average 40% of all users update to a new version on the release day. Around half of all app users update within the first week after release and 70% update within 30 days after release. However, to update all devices, on average 200 days elapse. Figure 3 illustrates the average apps update periods.

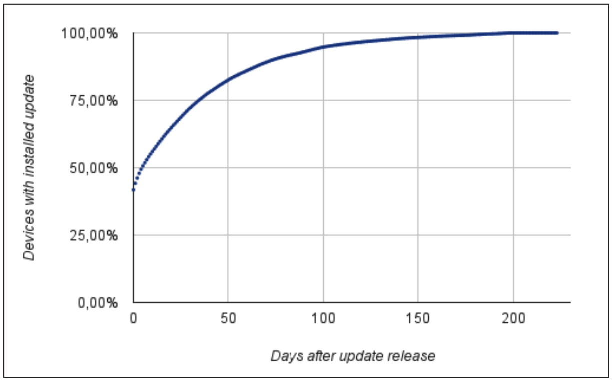


Figure 3: Average days between the release of a new app version and its installation.

These numbers illustrate that app updates are unsuitable for security-critical app components. In case of pinning, half of all users of an app would be unable to use the app during the first week after the update is released, since the pinning-secured TLS handshakes would fail [6].

6.4 Discussion

Our results have multiple major implications: The majority of TLS connections happen in third party libraries (86.33%) and another 68.85% of connections are established with shared hosts. However, although only 4.25% of all TLS connections seem to be good candidates for pinning, the current situation of only 45 apps that actually implement pinning leaves much room for improvement. An additional hurdle for deploying pinning are the lazy update frequencies of many users; losing half of the users for a total of one week after a certificate (pin) update is unacceptable for both app developers and their users. To facilitate the use of pinning for eligible developers, we cannot recommend to hard-code certificate pins into app binaries. Instead, pins should be included via configuration files that can easily be updated via a secure remote connection. Such an update mechanism could be enforced immediately and developers would not depend on the (lazy) update behaviour of their users. An alternative pinning deployment strategy would be to use multiple origins for a TLS connection, e. g. having `pin1.example.com` for one pin version and `pin2.example.com` for a second version. Both mechanisms allow for out-of-band pin updates but come with some extra effort on the developers' side.

7 Developer Support

After conducting a large scale analysis for real world Android apps and the applicability of pinning, we were also interested in the developers' point of view as the actors who actually implement pinning. First, we collected informative feedback from developers to learn more about their view on pinning. Second, based on their valuable and constructive feedback, we built a tool that supports developers in the process of deciding whether to implement pinning and eases the implementation.

7.1 Feedback

We analyzed all apps in our sample set and extracted possible candidates for pinning [7]. For those, we extracted the email addresses of the developers from Google Play, taking care not to create multiple emails for the same recipient, which left us with roughly 3,200 addresses. We emailed a random sample of 500 developers with an introductory email and the plea to provide us feedback on their experience with pinning, or, if they were not the developer, to please forward the email to their app's developer. Our outreach to developers was intended for a qualitative analysis of their feedback and comments to be used as the foundation to build our tool. A quantitative analysis was not our primary focus. We were able to gather 49[8] responses, of which we manually removed 4 who had answered in a nonsensical way or who clearly did not understand English. After we analyzed the 45 responses, we had gained insight into the major problem areas, a saturation of new input was reached and more responses would probably not have provided more valuable insights.

[6] We assume that apps use TLS-secured connections for critical components and do not work properly without Internet access.

[7] Here we made conservative choices to prevent unnecessarily bothering developers.

[8] Given the lack of a platform comparable to Amazon Mechanical Turk for software developer studies, which also means that they donate their time to our research for free, a response rate of 10% seems quite reasonable.

To build our tool with the best possible feature set and usability in mind, we were mainly interested in the following aspects:

Knowledge About Pinning 15 (a third) of our participants stated they knew what pinning was. We asked them to explain this knowledge, and their replies varied from correct mentions of "custom, self-signed certificates", "reduction of the reliance on intermediate/root certificates, if a intermediate/root gets compromised you don't." and "mobile apps that talk to the same well known server all the time" over "securing the communication between the app and the server without needing to pay to issuers out there" to "i don't know" and confused and/or wrong answers like "when you change the servers and/or certificates more often". We rated 80% of the answers as sensible.

Key Result: Only a quarter of the developers who gave us feedback have a basic understanding of pinning.

Desired Change: More detailed and critical explanation of what pinning is and how it works as part of the official Android documentation.

Obstacles Six participants had considered implementing pinning and decided against it. The reasons ranged from "laziness" over confusion to complaints about the complexity and the lack of an "out-of-the-box solution". To the two thirds of our participants who didn't know what pinning was, we showed a short explanatory text [9] and asked them to rate what they imagined could hypothetically keep them from using pinning or convince them to stay with the standard solution. We showed the same set of possible reasons to the developers who were informed about pinning and asked how much these reasons contributed to their not implementing pinning. They ranked "fear of losing users with old app versions / due to hard TLS fails" the highest, followed by "updates required when a certificate changes" and "complexity of the implementation". They said the standard solution was preferable, because "it is easier", they "trust in the existing CA-ecosystem" and "already own CA-signed certificates", but rather not because of "employing several different certificates".

Key Result: Of those who had heard of pinning, 40% had considered implementing it, but discounted it for being unusable or hard to implement.

Desired Change: Provide concrete sample code for the specific use case or app.

Wishlist We received wishes for "good tutorials and programming examples", "example code", "libraries across platforms", a "native Android API", a "test period and simple implementations" and the possibility to "do the same for the web front-end".

Key Result: Developers want better tool support and support in the decision process.

Desired Change: Easy-to-use tool support.

7.2 Tool Support

The developer feedback confirmed that more tool support is required and requested. When offering security solutions, we have to keep in mind that developers usually do not have a strong security focus and are not TLS experts, therefore, choosing and implementing secure solutions must be made as easy as possible. To this end, we built a tool that supports developers with implementing secure certificate validation in general; it additionally helps to decide whether pinning is the appropriate strategy. We made a web application publicly accessible at `https://pinning.android-ssl.org/`, which we base on our classification framework, the evaluation results and the developer study's results. We chose to implement our tool as a web application, since it is easily accessible and allows to keep the data backend up-to-date.

Developers and app users can upload APK files and have results presented to them in a clear web interface. First of all, the developers need to upload their app's APK file, whereupon the web application conducts all required information extraction steps (cf. Section 4) and presents the developer with an overview of relevant API calls and the corresponding remote origins that could be extracted. To increase accuracy, in the next step the developers are asked (1) whether they hold ownership for the relevant API calls and (2) whether they control the TLS configuration for the extracted remote origins (to keep the workflow as simple as possible and not overcharge the developer with unnecessary information, we filter out well known libraries and popular origins). Involving developers into the decision process is especially important in cases where automatic classification might not reveal accurate results, such as for configured origins (cf. Section 4).

Finally, the strategy classification is conducted and we present the developer with our recommendation for making their connections as secure as pos-

[9] taken and adapted from www.owasp.org

sible. In case the default strategy was selected, we do not recommend the developer to take further action. In case leaf or CA pinning is recommended, the developers are encouraged to increase their app's security by implementing pinning. However, we also inform the developers about the downsides of pinning in terms of updatability of certificate pins (cf. Section 6.3). In a last step, the developer is offered support for implementing pinning.

For any given relevant API call and the corresponding remote origin, concrete example code for pinning is generated over the following steps:

1. The remote origin's certificate is fetched and the corresponding pin is computed.
2. A `PinningTrustManager` that uses the pin for certificate validation is generated.
3. Surrounding code that includes the `PinningTrustManager` into the relevant API calls is generated, e.g. for an `HTTPSUrlConnection`, an `SSLContext` is generated that is initialized with the given `PinningTrustManager`.

The developer can then simply include this scaffold into the app and profit from a higher level of security. We asked the interested developers who left their contact information in our developer study to test our web application; 7 participated and gave positive feedback on its usability.

8 Limitations

In addition to the limitations described in Section 5.5, our work has three more limiations.

First, the update behaviour analysis we conducted for the users that participated in the AV's telemetry program might not necessarily represent the global update behaviour. However, we think that security affine users who install anti-virus software on their devices have a tendency to update their software more quickly than average users. Therefore, update frequencies for the global Android user population might be even worse.

Second, the feedback we got from app developers was based on self-reporting and might be influenced by a self-selection bias. Since we emailed developers who our classification framework had identified as good candidates for pinning, but offered them no incentive for taking part in our survey, we could only work with the developers who responded, leaving us with an opt-in bias. However, this is best practice for getting feedback from developers.

Third, our tool is currently implemented as a web service and a standalone command line tool. In the future, it would be reasonable to include the classification and recommendation process into the publication process in Google Play: An uploaded APK file could be run through the tool and unpinned but pinnable connections could be pointed out to the respective developer or pinned automatically via a library.

9 Conclusions

We conducted an extensive analysis on the applicability of pinning as an alternative and more secure certificate validation mechanism for non-browser software. Therefore, we analyzed 639,283 Android apps of which 229,317 (35.9%) use TLS to secure network connections and conservatively recommend pinning for 11,547 (1.8%) of all apps, or 5.0% of the apps that use TLS. This corresponds to 20,020,535 connections, of which 1,062,810 (5.3%) use TLS, of which 45,247 (4.25%) are conservatively recommendable for pinning. Optimistically, including estimates for unclassified connections as well as connections depending on dynamic code loading, we are able to suggest 140,858 connections or 58,817 (9.1%) apps to take pinning into consideration.

This contradicts the common assumption that pinning is a widely applicable solution for making TLS certificate validation in non-browser software more secure. Of the 229,317 apps we analyzed that make use of TLS to secure (some of) their network connections, 203,159 (88.6%) establish TLS connections via third-party libraries.

While we find that pinning is applicable only for relatively few apps and their TLS connections, a nominal-actual comparison illustrates that there is room for improving the current situation, as only 45 of the 11,481 apps that could benefit from pinning actually implement it. To help us understand affected developers and design a solution, we conducted a qualitative study with 45 developers, where we learned that pinning is relatively unknown and often neglected due to usability problems. These results incentivized us to build an easy-to-use web-application to support developers in the decision making process and guide them through the implementation of pinning for appropriate connections.

Our work concludes with the following take-aways:

Poor Support for Pinning The Android API lacks sufficient support for pinning: For low-level APIs, developers have to implement their own certificate validation and need detailed knowledge regarding pinning. For higher level APIs, support for pinning is missing entirely.

Limited Applicability The application of pinning in non-browser software such as apps is very limited: We recommend pinning for only 5.0% of the 229,317 TLS-enabled Android apps we analyzed.

Developer Education Developer feedback showed that only a third of the developers who could have implemented pinning had heard of it before. Pinning seems to be confusing and developers misinformed. In the future, better developer education is required as well as better developer support.

Security Updates Our analysis of update periods for Android apps suggests that Android requires mechanisms to quickly deploy security updates in the future (cf. Section 6.3).

Pinning Implementation The current Android documentation recommends to include pinning information at compile time, i. e. the recommendation is to add pins to the source code of an app. However, our analysis of the update behaviour of Android users suggests that developers should not implement certificate pins into an app's binary. Instead, we recommend to set pins via configuration files that are only accessible by the respective app.

References

[1] API, A. Android TLS API. `https://developer.android.com/training/articles/security-ssl.html`.

[2] Bates, A., Pletcher, J., Nichols, T., Hollembaek, B., Tian, D., Butler, K. R., and Alkhelaifi, A. Securing ssl certificate verification through dynamic linking. CCS '14, ACM, pp. 394–405.

[3] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and Polk, W. RFC 5280 Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. `https://datatracker.ietf.org/doc/rfc5280/?include_text=1`, May 2008.

[4] Desnos, A. Androguard. `http://code.google.com/p/androguard/`.

[5] Evans, C., Palmer, C., and Sleevi, R. Public Key Pinning Extension for HTTP. `http://tools.ietf.org/html/draft-ietf-websec-key-pinning-21`, Oct 2014. Internet-Draft.

[6] Fahl, S., Dechand, S., Perl, H., Fischer, F., Smrcek, J., and Smith, M. Hey, nsa: Stay away from my market! future proofing app markets against powerful attackers. CCS '14, ACM, pp. 1143–1155.

[7] Fahl, S., Harbach, M., Muders, T., Baumgärtner, L., Freisleben, B., and Smith, M. Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. CCS '12, ACM, pp. 50–61.

[8] Fahl, S., Harbach, M., Perl, H., Koetter, M., and Smith, M. Rethinking SSL Development in an Appified World. CCS '13, ACM, pp. 49–60.

[9] Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D., and Shmatikov, V. The Most Dangerous Code in the World: Validating SSL Certificates in Non-browser Software. CCS '12, ACM, pp. 38–49.

[10] Hoffman, P., and Schlyter, J. The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA. RFC 6698 (proposed standard), IETF, 2012. `http://tools.ietf.org/html/rfc6698`.

[11] Kranch, M., and Bonneau, J. Upgrading HTTPS in Mid-Air: An Empirical Study of Strict Transport Security and Key Pinning.

[12] Laurie, B., Langley, A., and Kasper, E. RFC 6962 Certificate Transparency. `http://tools.ietf.org/html/rfc6962`, June 2013.

[13] Lu, L., Li, Z., Wu, Z., Lee, W., and Jiang, G. Chex: Statically vetting android apps for component hijacking vulnerabilities. CCS '12, ACM, pp. 229–240.

[14] Marlinspike, M. Android Pinning. `https://github.com/moxie0/AndroidPinning`.

[15] Marlinspike, M. TACK: Trust Assertions for Certificate Keys. `http://tack.io/draft.html`.

[16] Marlinspike, M. SSL And The Future Of Authenticity. In BlackHat USA, 2011.

[17] OWASP. OWASP Certificate Pinning Guide. `https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning`.

[18] Perl, H., Fahl, S., and Smith, M. You Won't Be Needing These Any More: On Removing Unused Certificates From Trust Stores. In *Financial Cryptography and Data Security 2014* (2014).

[19] Poeplau, S., Fratantonio, Y., Bianchi, A., Kruegel, C., and Vigna, G. Execute This! Analyzing Unsafe and Malicious Dynamic Code.

[20] Rescorla, E. RFC 2818 HTTP Over TLS. `http://tools.ietf.org/html/rfc2818`, May 2000.

[21] Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., and Adams, C. RFC 6960 X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. `https://tools.ietf.org/html/rfc6960`, June 2013.

[22] Sounthiraraj, D., Sahs, J., Greenwood, G., Lin, Z., and Khan, L. SMV-Hunter: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps.

[23] Tendulkar, V., and Enck, W. An Application Package Configuration Approach to Mitigating Android SSL Vulnerabilities.

[24] Vallina-Rodriguez, N., Amann, J., Kreibich, C., Weaver, N., and Paxson, V. A tangled mass: The android root certificate stores. CoNEXT '14, ACM, pp. 141–148.

[25] Weiser, M. Program Slicing. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING VOL. SE-10, NO. 4* (1984), 352–357.

[26] Wendlandt, D., Andersen, D. G., and Perrig, A. Perspectives: Improving ssh-style host authentication with multi-path probing. ATC'08, USENIX Association, pp. 321–334.

De-anonymizing Programmers via Code Stylometry

Aylin Caliskan-Islam
Drexel University

Richard Harang
U.S. Army Research Laboratory

Andrew Liu
University of Maryland

Arvind Narayanan
Princeton University

Clare Voss
U.S. Army Research Laboratory

Fabian Yamaguchi
University of Goettingen

Rachel Greenstadt
Drexel University

Abstract

Source code authorship attribution is a significant privacy threat to anonymous code contributors. However, it may also enable attribution of successful attacks from code left behind on an infected system, or aid in resolving copyright, copyleft, and plagiarism issues in the programming fields. In this work, we investigate machine learning methods to de-anonymize source code authors of C/C++ using coding style. Our Code Stylometry Feature Set is a novel representation of coding style found in source code that reflects coding style from properties derived from abstract syntax trees.

Our random forest and abstract syntax tree-based approach attributes more authors (1,600 and 250) with significantly higher accuracy (94% and 98%) on a larger data set (Google Code Jam) than has been previously achieved. Furthermore, these novel features are robust, difficult to obfuscate, and can be used in other programming languages, such as Python. We also find that (i) the code resulting from difficult programming tasks is easier to attribute than easier tasks and (ii) skilled programmers (who can complete the more difficult tasks) are easier to attribute than less skilled programmers.

1 Introduction

Do programmers leave fingerprints in their source code? That is, does each programmer have a distinctive “coding style”? Perhaps a programmer has a preference for spaces over tabs, or `while` loops over `for` loops, or, more subtly, modular rather than monolithic code.

These questions have strong privacy and security implications. Contributors to open-source projects may hide their identity whether they are Bitcoin’s creator or just a programmer who does not want her employer to know about her side activities. They may live in a regime that prohibits certain types of software, such as censorship circumvention tools. For example, an Iranian programmer was sentenced to death in 2012 for developing photo sharing software that was used on pornographic websites [31].

The flip side of this scenario is that code attribution may be helpful in a forensic context, such as detection of ghostwriting, a form of plagiarism, and investigation of copyright disputes. It might also give us clues about the identity of malware authors. A careful adversary may only leave binaries, but others may leave behind code written in a scripting language or source code downloaded into the breached system for compilation.

While this problem has been studied previously, our work represents a qualitative advance over the state of the art by showing that Abstract Syntax Trees (ASTs) carry authorial ‘fingerprints.’ The highest accuracy achieved in the literature is 97%, but this is achieved on a set of only 30 programmers and furthermore relies on using programmer comments and larger amounts of training data [12, 14]. We match this accuracy on small programmer sets without this limitation. The largest scale experiments in the literature use 46 programmers and achieve 67.2% accuracy [10]. We are able to handle orders of magnitude more programmers (1,600) while using less training data with 92.83% accuracy. Furthermore, the features we are using are not trivial to obfuscate. We are able to maintain high accuracy while using commercial obfuscators. While abstract syntax trees can be obfuscated to an extent, doing so incurs significant overhead and maintenance costs.

Contributions. First, we use *syntactic features* for code stylometry. Extracting such features requires parsing of incomplete source code using a *fuzzy parser* to generate an *abstract syntax tree*. These features add a component to code stylometry that has so far remained almost completely unexplored. We provide evidence that these features are more fundamental and harder to obfuscate. Our complete feature set consists of a comprehensive set of around 120,000 layout-based, lexical, and syntactic features. With this complete feature set we are able to achieve a significant increase in accuracy compared to previous work. Second, we show that we can scale our method to 1,600 programmers without losing much accuracy. Third, this method is not specific to C or C++, and can be applied to any programming language.

We collected C++ source of thousands of contestants from the annual international competition "Google Code Jam". A bagging (portmanteau of "bootstrap aggregating") classifier - random forest was used to attribute programmers to source code. Our classifiers reach 98% accuracy in a 250-class closed world task, 93% accuracy in a 1,600-class closed world task, 100% accuracy on average in a two-class task. Finally, we analyze various attributes of programmers, types of programming tasks, and types of features that appear to influence the success of attribution. We identified the most important 928 features out of 120,000; 44% of them are syntactic, 1% are layout-based and the rest of the features are lexical. 8 training files with an average of 70 lines of code is sufficient for training when using the lexical, layout and syntactic features. We also observe that programmers with a greater skill set are more easily identifiable compared to less advanced programmers and that a programmer's coding style is more distinctive in implementations of difficult tasks as opposed to easier tasks.

The remainder of this paper is structured as follows. We begin by introducing applications of source code authorship attribution considered throughout this paper in Section 2, and present our AST-based approach in Section 3. We proceed to give a detailed overview of the experiments conducted to evaluate our method in Section 4 and discuss the insights they provide in Section 5. Section 6 presents related work, and Section 7 concludes.

2 Motivation

Throughout this work, we consider an analyst interested in determining the programmer of an anonymous fragment of source code purely based on its style. To do so, the analyst only has access to labeled samples from a set of candidate programmers, as well as from zero or more unrelated programmers.

The analyst addresses this problem by converting each labeled sample into a numerical feature vector, in order to train a machine learning classifier, that can subsequently be used to determine the code's programmer. In particular, this abstract problem formulation captures the following five settings and corresponding applications (see Table 1). The experimental formulations are presented in Section 4.2.

We emphasize that while these applications motivate our work, we have not directly studied them. Rather, we formulate them as variants of a machine-learning (classification) problem. Our data comes from the Google Code Jam competition, as we discuss in Section 4.1. Doubtless there will be additional challenges in using our techniques for digital forensics or any of the other real-world applications. We describe some known limitations in Section 5.

Programmer De-anonymization. In this scenario, the analyst is interested in determining the identity of an anonymous programmer. For example, if she has a set of programmers who she suspects might be Bitcoin's creator, Satoshi, and samples of source code from each of these programmers, she could use the initial versions of Bitcoin's source code to try to determine Satoshi's identity. Of course, this assumes that Satoshi did not make any attempts to obfuscate his or her coding style. Given a set of probable programmers, this is considered a closed-world machine learning task with multiple classes where anonymous source code is attributed to a programmer. This is a threat to privacy for open source contributors who wish to remain anonymous.

Ghostwriting Detection. Ghostwriting detection is related to but different from traditional plagiarism detection. We are given a suspicious piece of code and one or more candidate pieces of code that the suspicious code may have been plagiarized from. This is a well-studied problem, typically solved using code similarity metrics, as implemented by widely used tools such as MOSS [6], JPlag [25], and Sherlock [24].

For example, a professor may want to determine whether a student's programming assignment has been written by a student who has previously taken the class. Unfortunately, even though submissions of the previous year are available, the assignments may have changed considerably, rendering code-similarity based methods ineffective. Luckily, stylometry can be applied in this setting—we find the most stylistically similar piece of code from the previous year's corpus and bring both students in for gentle questioning. Given the limited set of students, this can be considered a closed-world machine learning problem.

Software Forensics. In software forensics, the analyst assembles a set of candidate programmers based on previously collected malware samples or online code repositories. Unfortunately, she cannot be sure that the anonymous programmer is one of the candidates, making this an *open world* classification problem as the test sample might not belong to any known category.

Copyright Investigation. Theft of code often leads to copyright disputes. Informal arrangements of hired programming labor are very common, and in the absence of a written contract, someone might claim a piece of code was her own after it was developed for hire and delivered. A dispute between two parties is thus a two-class classification problem; we assume that labeled code from both programmers is available to the forensic expert.

Authorship Verification. Finally, we may suspect that a piece of code was not written by the claimed programmer, but have no leads on who the actual programmer might be. This is the authorship verification problem. In this work, we take the textbook approach and model it as a two-class problem where positive examples come from previous works of the claimed programmer and negative examples come from randomly selected unrelated programmers. Alternatively, anomaly detection could be employed in this setting, e.g., using a one-class support vector machine [see 30].

As an example, a recent investigation conducted by Verizon [17] on a US company's anomalous virtual private network traffic, revealed an employee who was outsourcing her work to programmers in China. In such cases, training a classifier on employee's original code and that of random programmers, and subsequently testing pieces of recent code, could demonstrate if the employee was the actual programmer.

In each of these applications, the adversary may try to actively modify the program's coding style. In the software forensics application, the adversary tries to modify code written by her to hide her style. In the copyright and authorship verification applications, the adversary modifies code written by another programmer to match his own style. Finally, in the ghostwriting application, two of the parties may collaborate to modify the style of code written by one to match the other's style.

Application	Learner	Comments	Evaluation
De-anonymization	Multiclass	Closed world	Section 4.2.1
Ghostwriting detection	Multiclass	Closed world	Section 4.2.1
Software forensics	Multiclass	Open world	Section 4.2.2
Copyright investigation	Two-class	Closed world	Section 4.2.3
Authorship verification	Two/One-class	Open world	Section 4.2.4

Table 1: Overview of Applications for Code Stylometry

We emphasize that code stylometry that is robust to adversarial manipulation is largely left to future work. However, we hope that our demonstration of the power of features based on the abstract syntax tree will serve as the starting point for such research.

3 De-anonymizing Programmers

One of the goals of our research is to create a classifier that automatically determines the most likely author of a source file. Machine learning methods are an obvious choice to tackle this problem, however, their success crucially depends on the choice of a feature set that clearly represents programming style. To this end, we begin by parsing source code, thereby obtaining access to a wide range of possible features that reflect programming language use (Section 3.1). We then define a number of different features to represent both syntax and structure of program code (Section 3.2) and finally, we train a random forest classifier for classification of previously unseen source files (Section 3.3). In the following sections, we will discuss each of these steps in detail and outline design decisions along the way. The code for our approach is made available as open-source to allow other researchers to reproduce our results[1].

3.1 Fuzzy Abstract Syntax Trees

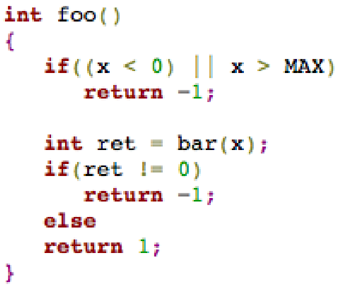
To date, methods for source code authorship attribution focus mostly on sequential feature representations of code such as byte-level and feature level n-grams [8, 13]. While these models are well suited to capture naming conventions and preference of keywords, they are entirely language agnostic and thus cannot model author characteristics that become apparent only in the composition of language constructs. For example, an author's tendency to create deeply nested code, unusually long functions or long chains of assignments cannot be modeled using n-grams alone.

Addressing these limitations requires source code to be parsed. Unfortunately, parsing C/C++ code using traditional compiler front-ends is only possible for *complete code*, i.e., source code where all identifiers can be resolved. This severely limits their applicability in the setting of authorship attribution as it prohibits analysis of lone functions or code fragments, as is possible with simple n-gram models.

As a compromise, we employ the fuzzy parser *Joern* that has been designed specifically with incomplete code in mind [32]. Where possible, the parser produces *abstract syntax trees* for code fragments while ignoring fragments that cannot be parsed without further information. The produced syntax trees form the basis for our feature extraction procedure. While they largely preserve the information required to create n-grams or bag-of-words representations, in addition, they allow a wealth of features to be extracted that encode programmer habits visible in the code's structure.

As an example, consider the function `foo` as shown in Figure 1, and a simplified version of its corresponding abstract syntax tree in Figure 2. The function contains a number of common language constructs found in many programming languages, such as if-statements (line 3 and 7), return-statements (line 4, 8 and 10), and function call expressions (line 6). For each of these constructs, the abstract syntax tree contains a corresponding node. While the leaves of the tree make classical syntactic features such as keywords, identifiers and operators accessible, inner nodes represent operations showing

[1] https://github.com/calaylin/CodeStylometry



```
int foo()
{
   if((x < 0) || x > MAX)
      return -1;

   int ret = bar(x);
   if(ret != 0)
      return -1;
   else
   return 1;
}
```

Figure 1: Sample Code Listing

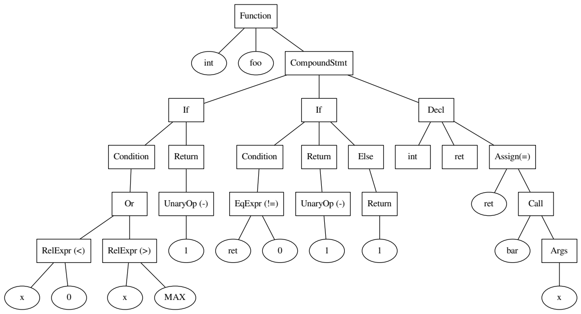


Figure 2: Corresponding Abstract Syntax Tree

how these basic elements are combined to form expressions and statements. In effect, the nesting of language constructs can also be analyzed to obtain a feature set representing the code's structure.

3.2 Feature Extraction

Analyzing coding style using machine learning approaches is not possible without a suitable representation of source code that clearly expresses program style. To address this problem, we present the *Code Stylometry Feature Set* (CSFS), a novel representation of source code developed specifically for code stylometry. Our feature set combines three types of features, namely *lexical features*, *layout features* and *syntactic features*. Lexical and layout features are obtained from source code while the syntactic features can only be obtained from ASTs. We now describe each of these feature types in detail.

3.2.1 Lexical and Layout Features

We begin by extracting numerical features from the source code that express preferences for certain identifiers and keywords, as well as some statistics on the use of functions or the nesting depth. Lexical and layout features can be calculated from the source code, without having access to a parser, with basic knowledge of the programming language in use. For example, we measure the number of functions per source line to determine the programmer's preference of longer over shorter functions. Furthermore, we tokenize the source file to obtain the number of occurrences of each token, so called *word unigrams*. Table 2 gives an overview of lexical features.

In addition, we consider *layout features* that represent code-indentation. For example, we determine whether the majority of indented lines begin with whitespace or tabulator characters, and we determine the ratio of whitespace to the file size. Table 3 gives a detailed description of these features.

Feature	**Definition**	**Count**
WordUnigramTF	Term frequency of word unigrams in source code	dynamic*
ln(num*keyword*/ length)	Log of the number of occurrences of *keyword* divided by file length in characters, where *keyword* is one of *do*, *else-if*, *if*, *else*, *switch*, *for* or *while*	7
ln(numTernary/ length)	Log of the number of ternary operators divided by file length in characters	1
ln(numTokens/ length)	Log of the number of word tokens divided by file length in characters	1
ln(numComments/ length)	Log of the number of comments divided by file length in characters	1
ln(numLiterals/ length)	Log of the number of string, character, and numeric literals divided by file length in characters	1
ln(numKeywords/ length)	Log of the number of unique keywords used divided by file length in characters	1
ln(numFunctions/ length)	Log of the number of functions divided by file length in characters	1
ln(numMacros/ length)	Log of the number of preprocessor directives divided by file length in characters	1
nestingDepth	Highest degree to which control statements and loops are nested within each other	1
branchingFactor	Branching factor of the tree formed by converting code blocks of files into nodes	1
avgParams	The average number of parameters among all functions	1
stdDevNumParams	The standard deviation of the number of parameters among all functions	1
avgLineLength	The average length of each line	1
stdDevLineLength	The standard deviation of the character lengths of each line	1
***About 55,000 for 250 authors with 9 files.**		

Table 2: Lexical Features

3.2.2 Syntactic Features

The syntactic feature set describes the properties of the language dependent abstract syntax tree, and keywords. Calculating these features requires access to an abstract syntax tree. All of these features are invariant to changes in source-code layout, as well as comments.

Table 4 gives an overview of our syntactic features. We obtain these features by preprocessing all C++ source files in the dataset to produce their abstract syntax trees.

Feature	Definition	Count
ln(numTabs/length)	Log of the number of tab characters divided by file length in characters	1
ln(numSpaces/length)	Log of the number of space characters divided by file length in characters	1
ln(numEmptyLines/length)	Log of the number of empty lines divided by file length in characters, excluding leading and trailing lines between lines of text	1
whiteSpaceRatio	The ratio between the number of whitespace characters (spaces, tabs, and newlines) and non-whitespace characters	1
newLineBefore OpenBrace	A boolean representing whether the majority of code-block braces are preceded by a newline character	1
tabsLeadLines	A boolean representing whether the majority of indented lines begin with spaces or tabs	1

Table 3: Layout Features

An abstract syntax tree is created for each function in the code. There are 58 node types in the abstract syntax tree (see Appendix A) produced by *Joern* [33].

Feature	Definition	Count
MaxDepthASTNode	Maximum depth of an AST node	1
ASTNodeBigramsTF	Term frequency AST node bigrams	dynamic*
ASTNodeTypesTF	Term frequency of 58 possible AST node type excluding leaves	58
ASTNodeTypesTFIDF	Term frequency inverse document frequency of 58 possible AST node type excluding leaves	58
ASTNodeTypeAvgDep	Average depth of 58 possible AST node types excluding leaves	58
cppKeywords	Term frequency of 84 C++ keywords	84
CodeInASTLeavesTF	Term frequency of code unigrams in AST leaves	dynamic**
CodeInASTLeaves TFIDF	Term frequency inverse document frequency of code unigrams in AST leaves	dynamic**
CodeInASTLeaves AvgDep	Average depth of code unigrams in AST leaves	dynamic**
***About 45,000 for 250 authors with 9 files.**		
****About 7,000 for 250 authors with 9 files.**		
****About 4,000 for 150 authors with 6 files.**		
****About 2,000 for 25 authors with 9 files.**		

Table 4: Syntactic Features

The AST node bigrams are the most discriminating features of all. AST node bigrams are two AST nodes that are connected to each other. In most cases, when used alone, they provide similar classification results to using the entire feature set.

The term frequency (TF) is the raw frequency of a node found in the abstract syntax trees for each file. The term frequency inverse document frequency (TFIDF) of nodes is calculated by multiplying the term frequency of a node by inverse document frequency. The goal in using the inverse document frequency is normalizing the term frequency by the number of authors actually using that particular type of node. The inverse document frequency is calculated by dividing the number of authors in the dataset by the number of authors that use that particular node. Consequently, we are able to capture how rare of a node it is and weight it more according to its rarity.

The maximum depth of an abstract syntax tree reflects the deepest level a programmer nests a node in the solution. The average depth of the AST nodes shows how nested or deep a programmer tends to use particular structural pieces. And lastly, term frequency of each C++ keyword is calculated. Each of these features is written to a feature vector to represent the solution file of a specific author and these vectors are later used in training and testing by machine learning classifiers.

3.3 Classification

Using the feature set presented in the previous section, we can now express fragments of source code as numerical vectors, making them accessible to machine learning algorithms. We proceed to perform feature selection and train a random forest classifier capable of identifying the most likely author of a code fragment.

3.3.1 Feature Selection

Due to our heavy use of unigram term frequency and TF/IDF measures, and the diversity of individual terms in the code, our resulting feature vectors are extremely large and sparse, consisting of tens of thousands of features for hundreds of classes. The dynamic *Code stylometry feature set*, for example, produced close to 120,000 features for 250 authors with 9 solution files each.

In many cases, such feature vectors can lead to overfitting (where a rare term, by chance, uniquely identifies a particular author). Extremely sparse feature vectors can also damage the accuracy of random forest classifiers, as the sparsity may result in large numbers of zero-valued features being selected during the random subsampling of the features to select a best split. This reduces the number of 'useful' splits that can be obtained at any given node, leading to poorer fits and larger trees. Large, sparse feature vectors can also lead to slowdowns in model fitting and evaluation, and are often more difficult to interpret. By selecting a smaller number of more informative features, the sparsity in the feature vector can be greatly reduced, thus allowing the classifier to both produce more accurate results and fit the data faster.

We therefore employed a feature selection step using WEKA's information gain [26] criterion, which evaluates the difference between the entropy of the distribution of classes and the entropy of the conditional distribution of classes given a particular feature:

$$IG(A, M_i) = H(A) - H(A|M_i) \qquad (1)$$

where A is the class corresponding to an author, H is Shannon entropy, and M_i is the i^{th} feature of the dataset. Intuitively, the information gain can be thought of as measuring the amount of information that the observation of the value of feature i gives about the class label associated with the example.

To reduce the total size and sparsity of the feature vector, we retained only those features that individually had non-zero information gain. (These features can be referred to as IG-CSFS throughout the rest of the paper.) Note that, as $H(A|M_i) \leq H(A)$, information gain is always non-negative. While the use of information gain on a variable-per-variable basis implicitly assumes independence between the features with respect to their impact on the class label, this conservative approach to feature selection means that we only use features that have demonstrable value in classification.

To validate this approach to feature selection, we applied this method to two distinct sets of source code files, and observed that sets of features with non-zero information gain were nearly identical between the two sets, and the ranking of features was substantially similar between the two. This suggests that the application of information gain to feature selection is producing a robust and consistent set of features (see Section 4 for further discussion). All the results are calculated by using CSFS and IG-CSFS. Using IG-CSFS on all experiments demonstrates how these features generalize to different datasets that are larger in magnitude. One other advantage of IG-CSFS is that it consists of a few hundred features that result in non-sparse feature vectors. Such a compact representation of coding style makes de-anonymizing thousands of programmers possible in minutes.

3.3.2 Random Forest Classification

We used the random forest ensemble classifier [7] as our classifier for authorship attribution. Random forests are ensemble learners built from collections of decision trees, each of which is grown by randomly sampling N training samples with replacement, where N is the number of instances in the dataset. To reduce correlation between trees, features are also subsampled; commonly $(logM)+1$ features are selected at random (without replacement) out of M, and the best split on these $(logM)+1$ features is used to split the tree nodes. The number of selected features represents one of the few tuning parameters in random forests: increasing the number of features increases the correlation between trees in the forest which can harm the accuracy of the overall ensemble, however increasing the number of features that can be chosen at each split increases the classification accuracy of each individual tree making them stronger classifiers with low error rates. The optimal range of number of features can be found using the out of bag (oob) error estimate, or the error estimate derived from those samples not selected for training on a given tree.

During classification, each test example is classified via each of the trained decision trees by following the binary decisions made at each node until a leaf is reached, and the results are then aggregated. The most populous class can be selected as the output of the forest for simple classification, or classifications can be ranked according to the number of trees that 'voted' for a label when performing relaxed attribution (see Section 4.3.4).

We employed random forests with 300 trees, which empirically provided the best trade-off between accuracy and processing time. Examination of numerous oob values across multiple fits suggested that $(logM)+1$ random features (where M denotes the total number of features) at each split of the decision trees was in fact optimal in all of the experiments (listed in Section 4), and was used throughout. Node splits were selected based on the information gain criteria, and all trees were grown to the largest extent possible, without pruning.

The data was analyzed via k-fold cross-validation, where the data was split into training and test sets stratified by author (ensuring that the number of code samples per author in the training and test sets was identical across authors). k varies according to datasets and is equal to the number of instances present from each author. The cross-validation procedure was repeated 10 times, each with a different random seed. We report the average results across all iterations in the results, ensuring that they are not biased by improbably easy or difficult to classify subsets.

4 Evaluation

In the evaluation section, we present the results to the possible scenarios formulated in the problem statement and evaluate our method. The corpus section gives an overview of the data we collected. Then, we present the main results to programmer de-anonymization and how it scales to 1,600 programmers, which is an immediate privacy concern for open source contributors that prefer to remain anonymous. We then present the training data requirements and efficacy of types of features. The obfuscation section discusses a possible countermeasure to programmer de-anonymization. We then present possible machine learning formulations along with the verification section that extends the approach to an open world problem. We conclude the evaluation with generalizing the method to other programming languages and providing software engineering insights.

4.1 Corpus

One concern in source code authorship attribution is that we are actually identifying differences in coding style, rather than merely differences in functionality. Consider the case where Alice and Bob collaborate on an open source project. Bob writes user interface code whereas Alice works on the network interface and backend analytics. If we used a dataset derived from their project, we might differentiate differences between frontend and backend code rather than differences in style.

In order to minimize these effects, we evaluate our method on the source code of solutions to programming tasks from the international programming competition *Google Code Jam (GCJ)*, made public in 2008 [2]. The competition consists of algorithmic problems that need to be solved in a programming language of choice. In particular, this means that all programmers solve the same problems, and hence implement similar functionality, a property of the dataset crucial for code stylometry analysis.

The dataset contains solutions by professional programmers, students, academics, and hobbyists from 166 countries. Participation statistics are similar over the years. Moreover, it contains problems of different difficulty, as the contest takes place in several rounds. This allows us to assess whether coding style is related to programmer experience and problem difficulty.

The most commonly used programming language was C++, followed by Java, and Python. We chose to investigate source code stylometry on C++ and C because of their popularity in the competition and having a parser for C/C++ readily available [32]. We also conducted some preliminary experimentation on Python.

A validation dataset was created from 2012's GCJ competition. Some problems had two stages, where the second stage involved answering the same problem in a limited amount of time and for a larger input. The solution to the large input is essentially a solution for the small input but not vice versa. Therefore, collecting both of these solutions could result in duplicate and identical source code. In order to avoid multiple entries, we only collected the small input versions' solutions to be used in our dataset.

The programmers had up to 19 solution files in these datasets. Solution files have an average of 70 lines of code per programmer.

To create our experimental datasets that are discussed in further detail in the results section;
(i) We first partitioned the corpus of files by year of competition. The "main" dataset includes files drawn from 2014 (250 programmers). The "validation" dataset files come from 2012, and the "multi-year" dataset files come from years 2008 through 2014 (1,600 programmers).
(ii) Within each year, we ordered the corpus files by the round in which they were written, and by the problem within a round, as all competitors proceed through the same sequence of rounds in that year. As a result, we performed stratified cross validation on each program file by the year it was written, by the round in which the program was written, by the problems solved in the round, and by the author's highest round completed in that year.

Some limitations of this dataset are that it does not allow us to assess the effect of style guidelines that may be imposed on a project or attributing code with multiple/mixed programmers. We leave these interesting questions for future work, but posit that our improved results with basic stylometry make them worthy of study.

4.2 Applications

In this section, we will go over machine learning task formulations representing five possible real-world applications presented in Section 2.

4.2.1 Multiclass Closed World Task

This section presents our main experiment—de-anonymizing 250 programmers in the difficult scenario where all programmers solved the same set of problems. The machine learning task formulation for de-anonymizing programmers also applies to ghostwriting detection. The biggest dataset formed from 2014's Google Code Jam Competition with 9 solution files to the same problem had 250 programmers. These were the easiest set of 9 problems, making the classification more challenging (see Section 4.3.6). We reached 91.78% accuracy in classifying 250 programmers with the *Code Stylometry Feature Set*. After applying information gain and using the features that had information gain, the accuracy was 95.08%.

We also took 250 programmers from different years and randomly selected 9 solution files for each one of them. We used the information gain features obtained from 2014's dataset to see how well they generalize. We reached 98.04% accuracy in classifying 250 programmers. This is 3% higher than the controlled large dataset's results. The accuracy might be increasing because of using a mixed set of Google Code Jam problems, which potentially contains the possible solutions' properties along with programmers' coding style and makes the code more distinct.

We wanted to evaluate our approach and validate our method and important features. We created a dataset from 2012's Google Code Jam Competition with 250 programmers who had the solutions to the same set of 9 problems. We extracted only the features that had positive information gain in 2014's dataset that was used as the main dataset to implement the approach. The classification accuracy was 96.83%, which is higher than the 95.07% accuracy obtained in 2014's dataset.

The high accuracy of validation results in Table 5 show that we identified the important features of code stylometry and found a stable feature set. This feature set does not necessarily represent the exact features for all possible datasets. For a given dataset that has ground truth information on authorship, following the same approach should generate the most important features that represent coding style in that particular dataset.

A = #programmers, F = max #problems completed		
N = #problems included in dataset (N ≤ F)		
A = 250 from 2014	**A = 250 from 2012**	**A = 250 all years**
F = 9 from 2014	**F = 9 from 2014**	**F ≥ 9 all years**
N = 9	**N = 9**	**N = 9**
Average accuracy after 10 iterations with IG-CSFS features		
95.07%	**96.83%**	**98.04%**

Table 5: Validation Experiments

4.2.2 Mutliclass Open World Task

The experiments in this section can be used in software forensics to find out the programmer of a piece of malware. In software forensics, the analyst does not know if source code belongs to one of the programmers in the candidate set of programmers. In such cases, we can classify the anonymous source code, and if the majority number of votes of trees in the random forest is below a certain threshold, we can reject the classification considering the possibility that it might not belong to any of the classes in the training data. By doing so, we can scale our approach to an open world scenario, where we might not have encountered the suspect before. As long as we determine a confidence threshold based on training data [30], we can calculate the probability that an instance belongs to one of the programmers in the set and accordingly accept or reject the classification.

We performed 270 classifications in a 30-class problem using all the features to determine the confidence threshold based on the training data. The accuracy was 96.67%. There were 9 misclassifications and all of them were classified with less than 15% confidence by the classifier. The class probability or classification confidence that source code fragment C is of class i is calculated by taking the percentage of trees in the random forest that voted for that particular class, as follows2:

$$P(C_i) = \frac{\sum_j V_j(i)}{|T|_f} \tag{2}$$

Where $V_j(i) = 1$ if the j^{th} tree voted for class i and 0 otherwise, and $|T|_f$ denotes the total number of trees in forest f. Note that by construction, $\sum_i P(C_i) = 1$ and $P(C_i) \geq 0 \; \forall \; i$, allowing us to treat $P(C_i)$ as a probability measure.

There was one correct classification made with 13.7% confidence. This suggests that we can use a threshold between 13.7% and 15% confidence level for verification, and manually analyze the classifications that did not pass the confidence threshold or exclude them from results.

We picked an aggressive threshold of 15% and to validate it, we trained a random forest classifier on the same set of 30 programmers 270 code samples. We tested on 150 different files from the programmers in the training set. There were 6 classifications below the 15% threshold and two of them were misclassified. We took another set of 420 test files from 30 programmers that were not in the training set. All the files from the 30 programmers were attributed to one of the 30 programmers in the training set since this is a closed world classification task, however, the highest confidence level in these classifications was 14.7%. The 15% threshold catches all the instances that do not belong to the programmers in the suspect set, gets rid of 2 misclassifications and 4 correct classifications. Consequently, when we see a classification with less than a threshold value, we can reject the classification and attribute the test instance to an unknown suspect.

4.2.3 Two-class Closed World Task

Source code author identification could automatically deal with source code copyright disputes without requiring manual analysis by an objective code investigator. A copyright dispute on code ownership can be resolved by comparing the styles of both parties claiming to have generated the code. The style of the disputed code can be compared to both parties' other source code to aid in the investigation. To imitate such a scenario, we took 60 different pairs of programmers, each with 9 solution files. We used a random forest and 9-fold cross validation to classify two programmers' source code. The average classification accuracy using CSFS set is 100.00% and 100.00% with the information gain features.

4.2.4 Two-class/One-class Open World Task

Another two-class machine learning task can be formulated for authorship verification. We suspect Mallory of plagiarizing, so we mix in some code of hers with a large sample of other people, test, and see if the disputed code gets classified as hers or someone else's. If it gets classified as hers, then it was with high probability really written by her. If it is classified as someone else's, it really was someone else's code. This could be an open world problem and the person that originally wrote the code could be a previously unknown programmer.

This is a two-class problem with classes Mallory and others. We train on Mallory's solutions to problems a, b, c, d, e, f, g, h. We also train on programmer A's solution to problem a, programmer B's solution to problem b, programmer C's solution to problem c, programmer D's solution to problem d, programmer E's solution to problem e, programmer F's solution to problem f, programmer G's solution to problem g, programmer H's solution to problem h and put them in one class called ABCDEFGH. We train a random forest classifier with 300 trees on classes Mallory and ABCDEFGH. We have 6 test instances from Mallory and 6 test instances from another programmer ZZZZZZ, who is not in the training set.

These experiments have been repeated in the exact same setting with 80 different sets of programmers ABCDEFGH, ZZZZZZ and Mallorys. The average classification accuracy for Mallory using the CSFS set is 100.00%. ZZZZZZ's test instances are classified as programmer ABCDEFGH 82.04% of the time, and classified as Mallory for the rest of the time while using the CSFS. Depending on the amount of false positives we are willing to accept, we can change the operating point on the ROC curve.

These results are also promising for use in cases where a piece of code is suspected to be plagiarized. Following the same approach, if the classification result of the piece of code is someone other than Mallory, that piece of code was with very high probability not written by Mallory.

4.3 Additional Insights

4.3.1 Scaling

We collected a larger dataset of 1,600 programmers from various years. Each of the programmers had 9 source code samples. We created 7 subsets of this large dataset in differing sizes, with 250, 500, 750, 1,000, 1,250, 1,500, and 1,600 programmers. These subsets are useful to understand how well our approach scales. We extracted the specific features that had information gain in the main 250 programmer dataset from this large dataset. In theory, we need to use more trees in the random forest as the number of classes increase to decrease variance, but we used fewer trees compared to smaller experiments. We used 300 trees in the random forest to run the experiments in a reasonable amount of time with a reasonable amount of memory. The accuracy did not decrease too much when increasing the number of programmers. This result shows that information gain features are robust against changes in class and are important properties of programmers' coding styles. The following Figure 3 demonstrates how well our method scales. We are able to de-anonymize 1,600 programmers using 32GB memory within one hour. Alternately, we can use 40 trees and get nearly the same accuracy (within 0.5%) in a few minutes.

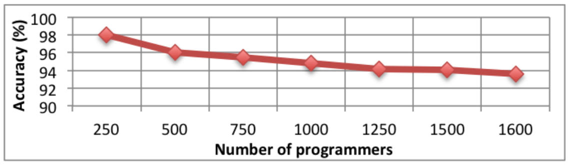


Figure 3: Large Scale De-anonymization

4.3.2 Training Data and Features

We selected different sets of 62 programmers that had F solution files, from 2 up to 14. Each dataset has the solutions to the same set of F problems by different sets of programmers. Each dataset consisted of programmers that were able to solve exactly F problems. Such an experimental setup makes it possible to investigate the effect of programmer skill set on coding style. The size of the datasets were limited to 62, because there were only 62 contestants with 14 files. There were a few contestants with up to 19 files but we had to exclude them since there were not enough programmers to compare them.

The same set of F problems were used to ensure that the coding style of the programmer is being classified and not the properties of possible solutions of the problem itself. We were able to capture personal programming style since all the programmers are coding the same functionality in their own ways.

Stratified F-fold cross validation was used by training on everyone's $(F-1)$ solutions and testing on the F^{th} problem that did not appear in the training set. As a result, the problems in the test files were encountered for the first time by the classifier.

We used a random forest with 300 trees and *(logM)+1* features with F-fold stratified cross validation, first with the *Code Stylometry Feature Set* (CSFS) and then with the CSFS's features that had information gain.

Figure 4 shows the accuracy from 13 different sets of 62 programmers with 2 to 14 solution files, and consequently 1 to 13 training files. The *CSFS* reaches an optimal training set size at 9 solution files, where the classifier trains on 8 $(F-1)$ solutions.

In the datasets we constructed, as the number of files increase and problems from more advanced rounds are included, the average line of code (LOC) per file also increases. The average lines of code per source code in the dataset is 70. Increased number of lines of code might have a positive effect on the accuracy but at the same time it reveals programmer's choice of program

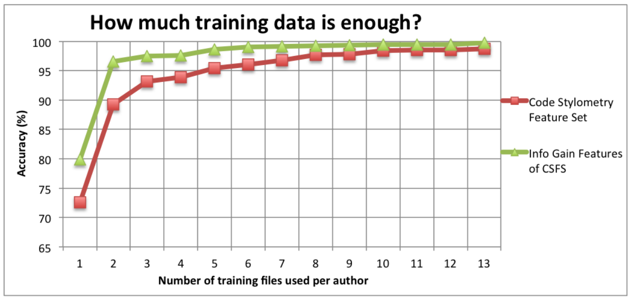


Figure 4: Training Data

length in implementing the same functionality. On the other hand, the average line of code of the 7 easier (76 LOC) or difficult problems (83 LOC) taken from contestants that were able to complete 14 problems, is higher than the average line of code (68) of contestants that were able to solve only 7 problems. This shows that programmers with better skills tend to write longer code to solve Google Code Jam problems. The mainstream idea is that better programmers write shorter and cleaner code which contradicts with line of code statistics in our datasets. Google Code Jam contestants are supposed to optimize their code to process large inputs with faster performance. This implementation strategy might be leading to advanced programmers implementing longer solutions for the sake of optimization.

We took the dataset with 62 programmers each with 9 solutions. We get 97.67% accuracy with all the features and 99.28% accuracy with the information gain features. We excluded all the syntactic features and the accuracy dropped to 88.89% with all the non-syntactic features and 88.35% with the information gain features of the non-syntactic feature set. We ran another experiment using only the syntactic features and obtained 96.06% with all the syntactic features and 96.96% with the information gain features of the syntactic feature set. Most of the classification power is preserved with the syntactic features, and using non-syntactic features leads to a significant decline in accuracy.

4.3.3 Obfuscation

We took a dataset with 9 solution files and 20 programmers and obfuscated the code using an off-the-shelf C++ obfuscator called stunnix [3]. The accuracy with the information gain code stylometry feature set on the obfuscated dataset is 98.89%. The accuracy on the same dataset when the code is not obfuscated is 100.00%. The obfuscator refactored function and variable names, as well as comments, and stripped all the spaces, preserving the functionality of code without changing the structure of the program. Obfuscating the data produced little detectable change in the performance of the classifier for this sample. The results are summarized in Table 6.

We took the maximum number of programmers, 20, that had solutions to 9 problems in C and obfuscated the code (see example in Appendix B) using a much more sophisticated open source obfuscator called Tigress [1]. In particular, Tigress implements *function virtualization*, an obfuscation technique that turns functions into interpreters and converts the original program into corresponding bytecode. After applying function virtualization, we were less able to effectively de-anonymize programmers, so it has potential as a countermeasure to programmer de-anonymization. However, this obfuscation comes at a cost. First of all, the obfuscated code is neither readable nor maintainable, and is thus unsuitable for an open source project. Second, the obfuscation adds significant overhead (9 times slower) to the runtime of the program, which is another disadvantage.

The accuracy with the information gain feature set on the obfuscated dataset is reduced to 67.22%. When we limit the feature set to AST node bigrams, we get 18.89% accuracy, which demonstrates the need for all feature types in certain scenarios. The accuracy on the same dataset when the code is not obfuscated is 95.91%.

Obfuscator	Programmers	Lang	Results w/o Obfuscation	Results w/ Obfuscation
Stunnix	20	C++	98.89%	100.00%
Stunnix	20	C++	98.89*%	98.89*%
Tigress	20	C	93.65%	58.33%
Tigress	20	C	95.91*%	67.22*%
*Information gain features				

Table 6: Effect of Obfuscation on De-anonymization

4.3.4 Relaxed Classification

The goal here is to determine whether it is possible to reduce the number of suspects using code stylometry. Reducing the set of suspects in challenging cases, such as having too many suspects, would reduce the effort required to manually find the actual programmer of the code.

In this section, we performed classification on the main 250 programmer dataset from 2014 using the information gain features. The classification was relaxed to a set of top *R* suspects instead of exact classification of the programmer. The relaxed factor *R* varied from 1 to 10. Instead of taking the highest majority vote of the decisions trees in the random forest, the highest *R* majority vote decisions were taken and the classification result was considered correct if the programmer was in the set of top *R* highest voted classes. The accuracy does not improve much after the relaxed factor is larger than 5.

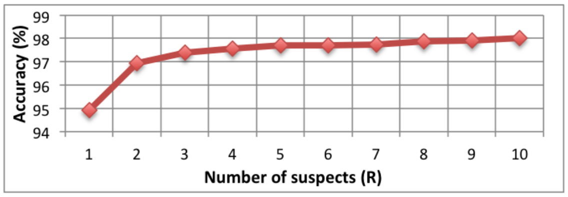


Figure 5: Relaxed Classification with 250 Programmers

4.3.5 Generalizing the Method

Features derived from ASTs can represent coding styles in various languages. These features are applicable in cases when lexical and layout features may be less discriminating due to formatting standards and reliance on whitespace and other 'lexical' features as syntax, such as Python's PEP8 formatting. To show that our method generalizes, we collected source code of 229 Python programmers from GCJ's 2014 competition. 229 programmers had exactly 9 solutions. **Using only the Python equivalents of syntactic features** listed in Table 4 and 9-fold cross-validation, the average accuracy is 53.91% for top-1 classification, 75.69% for top-5 relaxed attribution. The largest set of programmers to all work on the same set of 9 problems was 23 programmers. The average accuracy in identifying these 23 programmers is 87.93% for top-1 and 99.52% for top-5 relaxed attribution. The same classification tasks using the information gain features are also listed in Table 7. The overall accuracy in datasets composed of Python code are lower than C++ datasets. In Python datasets, we only used syntactic features from ASTs that were generated by a parser that was not fuzzy. The lack of quantity and specificity of features accounts for the decreased accuracy. The Python dataset's information gain features are significantly fewer in quantity, compared to C++ dataset's information gain features. Information gain only keeps features that have discriminative value all on their own. If two features only provide discriminative value when used together, then information gain will discard them. So if a lot of the features for the Python set are only jointly discriminative (and not individually discriminative), then the information gain criteria may be removing features that in combination could effectively discriminate between authors. This might account for the decrease when using information gain features. While in the context of other results in this paper the results in Table 7 appear lackluster, it is worth noting that even this preliminary test using only syntactic features has comparable performance to other prior work at a similar scale (see Section 6 and Table 9), demonstrating the utility of syntactic features and the relative ease of generating them for novel programming languages. Nevertheless, a CSFS equivalent feature set can be generated for other programming languages by implementing the layout and lexical features as well as using a fuzzy parser.

Lang.	Programmers	Classification	IG	Top-5	Top-5 IG
Python	23	87.93%	79.71%	99.52%	96.62
Python	229	53.91%	39.16%	75.69%	55.46

Table 7: Generalizing to Other Programming Languages

4.3.6 Software Engineering Insights

We wanted to investigate if programming style is consistent throughout years. We found the contestants that had the same username and country information both in 2012 and 2014. We assumed that these are the same people but there is a chance that they might be different people. In 2014, someone else might have picked up the same username from the same country and started using it. We are going to ignore such a ground truth problem for now and assume that they are the same people.

We took a set of 25 programmers from 2012 that were also contestants in 2014's competition. We took 8 files from their submissions in 2012 and trained a random forest classifier with 300 trees using CSFS. We had one instance from each one of the contestants from 2014. The correct classification of these test instances from 2014 is 96.00%. The accuracy dropped to 92.00% when using only information gain features, which might be due to the aggressive elimination of pairs of features that are jointly discriminative. These 25 programmers' 9 files from 2014 had a correct classification accuracy of 98.04%. These results indicate that coding style is preserved up to some degree throughout years.

To investigate problem difficulty's effect on coding style, we created two datasets from 62 programmers that had exactly 14 solution files. Table 8 summarizes the following results. A dataset with 7 of the easier problems out of 14 resulted in 95.62% accuracy. A dataset with 7 of the more difficult problems out of 14 resulted in 99.31% accuracy. This might imply that more difficult coding tasks have a more prevalent reflection of coding style. On the other hand, the dataset that had 62 programmers with exactly 7 of the easier problems resulted in 91.24% accuracy, which is a lot lower than the accuracy obtained from the dataset whose programmers were able to advance to solve 14 problems. This might indicate that, programmers who are advanced enough to answer 14 problems likely have more unique coding styles compared to contestants that were only able to solve the first 7 problems.

To investigate the possibility that contestants who are able to advance further in the rounds have more unique coding styles, we performed a second round of experiments on comparable datasets. We took the dataset with

12 solution files and 62 programmers. A dataset with 6 of the easier problems out of 12 resulted in 91.39% accuracy. A dataset with 6 of the more difficult problems out of 12 resulted in 94.35% accuracy. These results are higher than the dataset whose programmers were only able to solve the easier 6 problems. The dataset that had 62 programmers with exactly 6 of the easier problems resulted in 90.05% accuracy.

A = #programmers, F = max #problems completed					
N = #problems included in dataset (N ≤ F)					
A = 62					
F = 14		**F = 7**	**F = 12**		**F = 6**
N = 7	**N = 7**	**N = 7**	**N = 6**	**N = 6**	**N = 6**
Average accuracy after 10 iterations while using CSFS					
99.31%	*95.62%*[2]	**91.24%**[1]	94.35%	*91.39%*[2]	**90.05%**[1]
Average accuracy after 10 iterations while using IG CSFS					
99.38%	*98.62%*[2]	**96.77%**[1]	96.69%	*95.43%*[2]	**94.89%**[1]
[1] **Drop in accuracy due to programmer skill set.**					
[2] **Coding style is more distinct in more difficult tasks.**					

Table 8: Effect of Problem Difficulty on Coding Style

5 Discussion

In this section, we discuss the conclusions we draw from the experiments outlined in the previous section, limitations, as well as questions raised by our results. In particular, we discuss the difficulty of the different settings considered, the effects of obfuscation, and limitations of our current approach.

Problem Difficulty. The experiment with random problems from random authors among seven years most closely resembles a real world scenario. In such an experimental setting, there is a chance that instead of only identifying authors we are also identifying the properties of a specific problem's solution, which results in a boost in accuracy.

In contrast, our main experimental setting where all authors have only answered the nine easiest problems is possibly the hardest scenario, since we are training on the same set of eight problems that all the authors have algorithmically solved and try to identify the authors from the test instances that are all solutions of the 9th problem. On the upside, these test instances help us precisely capture the differences between individual coding style that represent the same functionality. We also see that such a scenario is harder since the randomized dataset has higher accuracy.

Classifying authors that have implemented the solution to a set of difficult problems is easier than identifying authors with a set of easier problems. This shows that coding style is reflected more through difficult programming tasks. This might indicate that programmers come up with unique solutions and preserve their coding style more when problems get harder. On the other hand, programmers with a better skill set have a prevalent coding style which can be identified more easily compared to contestants who were not able to advance as far in the competition. This might indicate that as programmers become more advanced, they build a stronger coding style compared to novices. There is another possibility that maybe better programmers start out with a more unique coding style.

Effects of Obfuscation. A malware author or plagiarizing programmer might deliberately try to hide his source code by obfuscation. Our experiments indicate that our method is resistant to simple off-the-shelf obfuscators such as stunnix, that make code look cryptic while preserving functionality. The reason for this success is that the changes stunnix makes to the code have no effect on syntactic features, e.g., removal of comments, changing of names, and stripping of whitespace.

In contrast, sophisticated obfuscation techniques such as function virtualization hinder de-anonymization to some degree, however, at the cost of making code unreadable and introducing a significant performance penalty. Unfortunately, unreadability of code is not acceptable for open-source projects, while it is no problem for attackers interested in covering their tracks. Developing methods to automatically remove stylometric information from source code without sacrificing readability is therefore a promising direction for future research.

Limitations. We have not considered the case where a source file might be written by a different author than the stated contestant, which is a ground truth problem that we cannot control. Moreover, it is often the case that code fragments are the work of multiple authors. We plan to extend this work to study such datasets. To shed light on the feasibility of classifying such code, we are currently working with a dataset of git commits to open source projects. Our parser works on code fragments rather than complete code, consequently we believe this analysis will be possible.

Another fundamental problem for machine learning classifiers are mimicry attacks. For example, our classifier may be evaded by an adversary by adding extra dummy code to a file that closely resembles that of another programmer, albeit without affecting the program's behavior. This evasion is possible, but trivial to resolve when an analysts verifies the decision.

Finally, we cannot be sure whether the original author is actually a Google Code Jam contestant. In this case, we can detect those by a classify and then verify approach as explained in Stolerman et al.'s work [30]. Each classification could go through a verification step to eliminate instances where the classifier's confidence is below a threshold. After the verification step, instances that do not belong to the set of known authors can be separated from the dataset to be excluded or for further manual analysis.

6 Related Work

Our work is inspired by the research done on authorship attribution of unstructured or semi-structured text [5, 22]. In this section, we discuss prior work on source code authorship attribution. In general, such work (Table 9) looks at smaller scale problems, does not use structural features, and achieves lower accuracies than our work.

The highest accuracies in the related work are achieved by Frantzeskou et al. [12, 14]. They used 1,500 7-grams to reach 97% accuracy with 30 programmers. They investigated the high-level features that contribute to source code authorship attribution in Java and Common Lisp. They determined the importance of each feature by iteratively excluding one of the features from the feature set. They showed that comments, layout features and naming patterns have a strong influence on the author classification accuracy. They used more training data (172 line of code on average) than us (70 lines of code). We replicated their experiments on a 30 programmer subset of our C++ data set, with eleven files containing 70 lines of code on average and no comments. We reach 76.67% accuracy with 6-grams, and 76.06% accuracy with 7-grams. When we used a 6 and 7-gram feature set on 250 programmers with 9 files, we got 63.42% accuracy. With our original feature set, we get 98% accuracy on 250 programmers.

The largest number of programmers studied in the related work was 46 programmers with 67.2% accuracy. Ding and Samadzadeh [10] use statistical methods for authorship attribution in Java. They show that among lexical, keyword and layout properties, layout metrics have a more important role than others which is not the case in our analysis.

There are also a number of smaller scale, lower accuracy approaches in the literature [9, 11, 18–21, 28], shown in Table 9, all of which we significantly outperform. These approaches use a combination of layout and lexical features.

The only other work to explore structural features is by Pellin [23], who used manually parsed abstract syntax trees with an SVM that has a tree based kernel to classify functions of two programmers. He obtains an average of 73% accuracy in a two class classification task. His approach explained in the white paper can be extended to our approach, so it is the closest to our work in the literature. This work demonstrates that it is non-trivial to use ASTs effectively. Our work is the first to use structural features to achieve higher accuracies at larger scales and the first to study how code obfuscation affects code stylometry.

There has also been some code stylometry work that focused on manual analysis and case studies. Spafford and Weeber [29] suggest that use of lexical features such as variable names, formatting and comments, as well as some syntactic features such as usage of keywords, scoping and presence of bugs could aid in source code attribution but they do not present results or a case study experiment with a formal approach. Gray et al. [15] identify three categories in code stylometry: the layout of the code, variable and function naming conventions, types of data structures being used and also the cyclomatic complexity of the code obtained from the control flow graph. They do not mention anything about the syntactic characteristics of code, which could potentially be a great marker of coding style that reveals the usage of programming language's grammar. Their case study is based on a manual analysis of three worms, rather than a statistical learning approach. Hayes and Offutt [16] examine coding style in source code by their consistent programmer hypothesis. They focused on lexical and layout features, such as the occurrence of semicolons, operators and constants. Their dataset consisted of 20 programmers and the analysis was not automated. They concluded that coding style exists through some of their features and professional programmers have a stronger programming style compared to students. In our results in Section 4.3.6, we also show that more advanced programmers have a more identifying coding style.

There is also a great deal of research on plagiarism detection which is carried out by identifying the similarities between different programs. For example, there is a widely used tool called Moss that originated from Stanford University for detecting software plagiarism. Moss [6] is able to analyze the similarities of code written by different programmers. Rosenblum et al. [27] present a novel program representation and techniques that automatically detect the stylistic features of binary code.

Related Work	**# of Programmers**	**Results**
Pellin [23]	2	73%
MacDonell et al.[21]	7	88.00%
Frantzeskou et al.[14]	8	100.0%
Burrows et al. [9]	10	76.78%
Elenbogen and Seliya [11]	12	74.70%
Kothari et al. [18]	12	76%
Lange and Mancoridis [20]	20	75%
Krsul and Spafford [19]	29	73%
Frantzeskou et al. [14]	30	96.9%
Ding and Samadzadeh [10]	46	67.2%
This work	**8**	**100.00%**
This work	**35**	**100.00%**
This work	**250**	**98.04%**
This work	**1,600**	**92.83%**

Table 9: Comparison to Previous Results

7 Conclusion and Future Work

Source code stylometry has direct applications for privacy, security, software forensics, plagiarism, copyright infringement disputes, and authorship verification. Source code stylometry is an immediate concern for programmers who want to contribute code anonymously because de-anonymization is quite possible. We introduce the first principled use of syntactic features along with lexical and layout features to investigate style in source code. We can reach 94% accuracy in classifying 1,600 authors and 98% accuracy in classifying 250 authors with eight training files per class. This is a significant increase in accuracy and scale in source code authorship attribution. In particular, it shows that source code authorship attribution with the *Code Stylometry Feature Set* scales even better than regular stylometric authorship attribution, as these methods can only identify individuals in sets of 50 authors with slightly over 90% accuracy [see 4]. Furthermore, this performance is achieved by training on only 550 lines of code or eight solution files, whereas classical stylometric analysis requires 5,000 words.

Additionally, our results raise a number of questions that motivate future research. First, as malicious code is often only available in binary format, it would be interesting to investigate whether syntactic features can be partially preserved in binaries. This may require our feature set to be improved in order to incorporate information obtained from control flow graphs.

Second, we would also like to see if classification accuracy can be further increased. For example, we would like to explore whether using features that have joint information gain alongside features that have information gain by themselves improve performance. Moreover, designing features that capture larger fragments of the abstract syntax tree could provide improvements. These changes (along with adding lexical and layout features) may provide significant improvements to the Python results and help generalize the approach further.

Finally, we would like to investigate whether code can be automatically normalized to remove stylistic information while preserving functionality and readability.

8 Acknowledgments

This material is based on work supported by the ARO (U.S. Army Research Office) Grant W911NF-14-1-0444, the DFG (German Research Foundation) under the project DEVIL (RI 2469/1-1), and AWS in Education Research Grant award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the ARO, DFG, and AWS.

References

[1] The tigress diversifying c virtualizer, http://tigress.cs.arizona.edu.

[2] Google code jam, https://code.google.com/codejam, 2014.

[3] Stunnix, http://www.stunnix.com/prod/cxxo/, November 2014.

[4] ABBASI, A., AND CHEN, H. Writeprints: A stylometric approach to identity-level identification and similarity detection in cyberspace. *ACM Trans. Inf. Syst. 26*, 2 (2008), 1–29.

[5] AFROZ, S., BRENNAN, M., AND GREENSTADT, R. Detecting hoaxes, frauds, and deception in writing style online. In *Security and Privacy (SP), 2012 IEEE Symposium on* (2012), IEEE, pp. 461–475.

[6] AIKEN, A., ET AL. Moss: A system for detecting software plagiarism. *University of California–Berkeley. See www. cs. berkeley. edu/aiken/moss. html 9* (2005).

[7] BREIMAN, L. Random forests. *Machine Learning 45*, 1 (2001), 5–32.

[8] BURROWS, S., AND TAHAGHOGHI, S. M. Source code authorship attribution using n-grams. In *Proc. of the Australasian Document Computing Symposium* (2007).

[9] BURROWS, S., UITDENBOGERD, A. L., AND TURPIN, A. Application of information retrieval techniques for source code authorship attribution. In *Database Systems for Advanced Applications* (2009), Springer, pp. 699–713.

[10] DING, H., AND SAMADZADEH, M. H. Extraction of java program fingerprints for software authorship identification. *Journal of Systems and Software 72*, 1 (2004), 49–57.

[11] ELENBOGEN, B. S., AND SELIYA, N. Detecting outsourced student programming assignments. *Journal of Computing Sciences in Colleges 23*, 3 (2008), 50–57.

[12] FRANTZESKOU, G., MACDONELL, S., STAMATATOS, E., AND GRITZALIS, S. Examining the significance of high-level programming features in source code author classification. *Journal of Systems and Software 81*, 3 (2008), 447–460.

[13] FRANTZESKOU, G., STAMATATOS, E., GRITZALIS, S., CHASKI, C. E., AND HOWALD, B. S. Identifying authorship by byte-level n-grams: The source code author profile (scap) method. *International Journal of Digital Evidence 6*, 1 (2007), 1–18.

[14] FRANTZESKOU, G., STAMATATOS, E., GRITZALIS, S., AND KATSIKAS, S. Effective identification of source code authors using byte-level information. In *Proceedings of the 28th International Conference on Software Engineering* (2006), ACM, pp. 893–896.

[15] GRAY, A., SALLIS, P., AND MACDONELL, S. Software forensics: Extending authorship analysis techniques to computer programs.

[16] HAYES, J. H., AND OFFUTT, J. Recognizing authors: an examination of the consistent programmer hypothesis. *Software Testing, Verification and Reliability 20*, 4 (2010), 329–356.

[17] INOCENCIO, R. U.s. programmer outsources own job to china, surfs cat videos, January 2013.

[18] Kothari, J., Shevertalov, M., Stehle, E., and Mancoridis, S. A probabilistic approach to source code authorship identification. In *Information Technology, 2007. ITNG'07. Fourth International Conference on* (2007), IEEE, pp. 243–248.

[19] Krsul, I., and Spafford, E. H. Authorship analysis: Identifying the author of a program. *Computers & Security 16*, 3 (1997), 233–257.

[20] Lange, R. C., and Mancoridis, S. Using code metric histograms and genetic algorithms to perform author identification for software forensics. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation* (2007), ACM, pp. 2082–2089.

[21] MacDonell, S. G., Gray, A. R., MacLennan, G., and Sallis, P. J. Software forensics for discriminating between program authors using case-based reasoning, feedforward neural networks and multiple discriminant analysis. In *Neural Information Processing, 1999. Proceedings. ICONIP'99. 6th International Conference on* (1999), vol. 1, IEEE, pp. 66–71.

[22] Narayanan, A., Paskov, H., Gong, N. Z., Bethencourt, J., Stefanov, E., Shin, E. C. R., and Song, D. On the feasibility of internet-scale author identification. In *Security and Privacy (SP), 2012 IEEE Symposium on* (2012), IEEE, pp. 300–314.

[23] Pellin, B. N. Using classification techniques to determine source code authorship. *White Paper: Department of Computer Science, University of Wisconsin* (2000).

[24] Pike, R. The sherlock plagiarism detector, 2011.

[25] Prechelt, L., Malpohl, G., and Philippsen, M. Finding plagiarisms among a set of programs with jplag. *J. UCS 8*, 11 (2002), 1016.

[26] Quinlan, J. Induction of decision trees. *Machine learning 1*, 1 (1986), 81–106.

[27] Rosenblum, N., Zhu, X., and Miller, B. Who wrote this code? identifying the authors of program binaries. *Computer Security–ESORICS 2011* (2011), 172–189.

[28] Shevertalov, M., Kothari, J., Stehle, E., and Mancoridis, S. On the use of discretized source code metrics for author identification. In *Search Based Software Engineering, 2009 1st International Symposium on* (2009), IEEE, pp. 69–78.

[29] Spafford, E. H., and Weeber, S. A. Software forensics: Can we track code to its authors? *Computers & Security 12*, 6 (1993), 585–595.

[30] Stolerman, A., Overdorf, R., Afroz, S., and Greenstadt, R. Classify, but verify: Breaking the closed-world assumption in stylometric authorship attribution. In *IFIP Working Group 11.9 on Digital Forensics* (2014), IFIP.

[31] Wikipedia. Saeed Malekpour, 2014. [Online; accessed 04-November-2014].

[32] Yamaguchi, F., Golde, N., Arp, D., and Rieck, K. Modeling and discovering vulnerabilities with code property graphs. In *Proc of IEEE Symposium on Security and Privacy (S&P)* (2014).

[33] Yamaguchi, F., Wressnegger, C., Gascon, H., and Rieck, K. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (2013), ACM, pp. 499–510.

A Appendix: Keywords and Node Types

AdditiveExpression	AndExpression	Argument
ArgumentList	ArrayIndexing	AssignmentExpr
BitAndExpression	BlockStarter	BreakStatement
Callee	CallExpression	CastExpression
CastTarget	CompoundStatement	Condition
ConditionalExpression	ContinueStatement	DoStatement
ElseStatement	EqualityExpression	ExclusiveOrExpression
Expression	ExpressionStatement	ForInit
ForStatement	FunctionDef	GotoStatement
Identifier	IdentifierDecl	IdentifierDeclStatement
IdentifierDeclType	IfStatement	IncDec
IncDecOp	InclusiveOrExpression	InitializerList
Label	MemberAccess	MultiplicativeExpression
OrExpression	Parameter	ParameterList
ParameterType	PrimaryExpression	PtrMemberAccess
RelationalExpression	ReturnStatement	ReturnType
ShiftExpression	Sizeof	SizeofExpr
SizeofOperand	Statement	SwitchStatement
UnaryExpression	UnaryOp	UnaryOperator
WhileStatement		

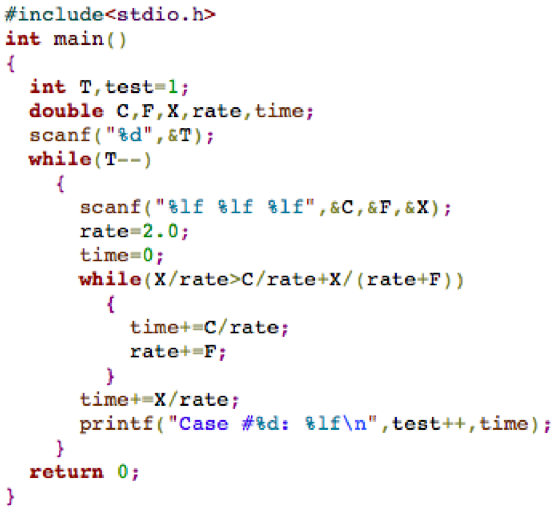
Table 10: Abstract syntax tree node types

Table 10 lists the AST node types generated by *Joern* that were incorporated to the feature set. Table 11 shows the C++ keywords used in the feature set.

alignas	alignof	and	and_eq	asm
auto	bitand	bitor	bool	break
case	catch	char	char16_t	char32_t
class	compl	const	constexpr	const_cast
continue	decltype	default	delete	do
double	dynamic_cast	else	enum	explicit
export	extern	false	float	for
friend	goto	if	inline	int
long	mutable	namespace	new	noexcept
not	not_eq	nullptr	operator	or
or_eq	private	protected	public	register
reinterpret_cast	return	short	signed	sizeof
static	static_assert	static_cast	struct	switch
template	this	thread_local	throw	true
try	typedef	typeid	typename	union
unsigned	using	virtual	void	volatile
wchar_t	while	xor	xor_eq	

Table 11: C++ keywords

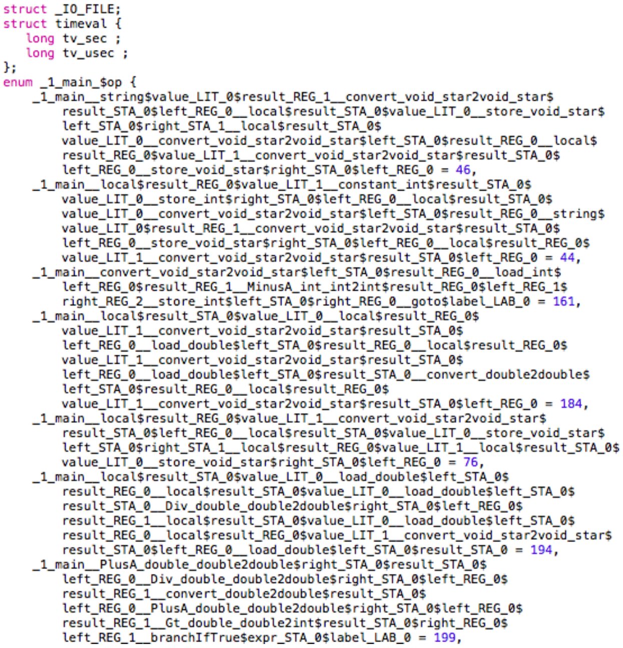
B Appendix: Original vs Obfuscated Code



```
#include<stdio.h>
int main()
{
  int T,test=1;
  double C,F,X,rate,time;
  scanf("%d",&T);
  while(T--)
    {
      scanf("%lf %lf %lf",&C,&F,&X);
      rate=2.0;
      time=0;
      while(X/rate>C/rate+X/(rate+F))
        {
          time+=C/rate;
          rate+=F;
        }
      time+=X/rate;
      printf("Case #%d: %lf\n",test++,time);
    }
  return 0;
}
```

Figure 6: A code sample *X*

Figure 6 shows a source code sample *X* from our dataset that is 21 lines long. After obfuscation with Tigress, sample *X* became 537 lines long. Figure 7 shows the first 13 lines of the obfuscated sample *X*.



```
struct _IO_FILE;
struct timeval {
   long tv_sec ;
   long tv_usec ;
};
enum _1_main_$op {
    _1_main__string$value_LIT_0$result_REG_1__convert_void_star2void_star$
        result_STA_0$left_REG_0__local$result_STA_0$value_LIT_0__store_void_star$
        left_STA_0$right_STA_1__local$result_STA_0$
        value_LIT_0__convert_void_star2void_star$left_STA_0$result_REG_0__local$
        result_REG_0$value_LIT_1__convert_void_star2void_star$result_STA_0$
        left_REG_0__store_void_star$right_STA_0$left_REG_0 = 46,
    _1_main__local$result_REG_0$value_LIT_1__constant_int$result_STA_0$
        value_LIT_0__store_int$right_STA_0$left_REG_0__local$result_STA_0$
        value_LIT_0__convert_void_star2void_star$left_STA_0$result_REG_0__string$
        value_LIT_0$result_REG_1__convert_void_star2void_star$result_STA_0$
        left_REG_0__store_void_star$right_STA_0$left_REG_0__local$result_REG_0$
        value_LIT_1__convert_void_star2void_star$result_STA_0$left_REG_0 = 44,
    _1_main__convert_void_star2void_star$left_STA_0$result_REG_0__load_int$
        left_REG_0$result_REG_1__MinusA_int_int2int$result_REG_0$left_REG_1$
        right_REG_2__store_int$left_STA_0$right_REG_0__goto$label_LAB_0 = 161,
    _1_main__local$result_STA_0$value_LIT_0__local$result_REG_0$
        value_LIT_1__convert_void_star2void_star$result_STA_0$
        left_REG_0__load_double$left_STA_0$result_REG_0__local$result_REG_0$
        value_LIT_1__convert_void_star2void_star$result_STA_0$
        left_REG_0__load_double$left_STA_0$result_STA_0__convert_double2double$
        left_STA_0$result_REG_0__local$result_REG_0$
        value_LIT_1__convert_void_star2void_star$result_STA_0$left_REG_0 = 184,
    _1_main__local$result_REG_0$value_LIT_1__convert_void_star2void_star$
        result_STA_0$left_REG_0__local$result_STA_0$value_LIT_0__store_void_star$
        left_STA_0$right_STA_1__local$result_REG_0$value_LIT_1__local$result_STA_0$
        value_LIT_0__store_void_star$right_STA_0$left_REG_0 = 76,
    _1_main__local$result_STA_0$value_LIT_0__load_double$left_STA_0$
        result_REG_0__local$result_STA_0$value_LIT_0__load_double$left_STA_0$
        result_STA_0__Div_double_double2double$right_STA_0$left_REG_0$
        result_REG_1__local$result_STA_0$value_LIT_0__load_double$left_STA_0$
        result_REG_0__local$result_REG_0$value_LIT_1__convert_void_star2void_star$
        result_STA_0$left_REG_0__load_double$left_STA_0$result_STA_0 = 194,
    _1_main__PlusA_double_double2double$right_STA_0$result_STA_0$
        left_REG_0__Div_double_double2double$right_STA_0$left_REG_0$
        result_REG_1__convert_double2double$result_STA_0$
        left_REG_0__PlusA_double_double2double$right_STA_0$left_REG_0$
        result_REG_1__Gt_double_double2int$result_STA_0$right_REG_0$
        left_REG_1__branchIfTrue$expr_STA_0$label_LAB_0 = 199,
```

Figure 7: Code sample *X* after obfuscation

RAPTOR: Routing Attacks on Privacy in Tor

Yixin Sun
Princeton University

Anne Edmundson
Princeton University

Laurent Vanbever
ETH Zurich

Oscar Li
Princeton University

Jennifer Rexford
Princeton University

Mung Chiang
Princeton University

Prateek Mittal
Princeton University

Abstract

The Tor network is a widely used system for anonymous communication. However, Tor is known to be vulnerable to attackers who can observe traffic at both ends of the communication path. In this paper, we show that prior attacks are just the tip of the iceberg. We present a suite of new attacks, called Raptor, that can be launched by Autonomous Systems (ASes) to compromise user anonymity. First, AS-level adversaries can exploit the asymmetric nature of Internet routing to increase the chance of observing at least one direction of user traffic at both ends of the communication. Second, AS-level adversaries can exploit natural churn in Internet routing to lie on the BGP paths for more users over time. Third, strategic adversaries can manipulate Internet routing via BGP hijacks (to discover the users using specific Tor guard nodes) and interceptions (to perform traffic analysis). We demonstrate the feasibility of Raptor attacks by analyzing historical BGP data and Traceroute data as well as performing real-world attacks on the live Tor network, while ensuring that we do not harm real users. In addition, we outline the design of two monitoring frameworks to counter these attacks: BGP monitoring to detect control-plane attacks, and Traceroute monitoring to detect data-plane anomalies. Overall, our work motivates the design of anonymity systems that are aware of the dynamics of Internet routing.

1 Introduction

Anonymity systems aim to protect user identities from untrusted destinations and third parties on the Internet. Among all of them, the Tor network [25] is the most widely used. As of February 2015, the Tor network comprises of 7,000 relays or proxies which together carry terabytes of traffic every day [8]. Tor serves millions of users and is often publicized by political dissidents, whistle-blowers, law-enforcement, intelligence agencies, journalists, businesses and ordinary citizens concerned about the privacy of their online communications [9].

Along with anonymity, Tor aims to provide low latency and, as such, does not obfuscate packet timings or sizes. Consequently, an adversary who is able to observe traffic on both segments of the Tor communication channel (*i.e.*, between the server and the Tor network, and between the Tor network and the client) can correlate packet sizes and packet timings to deanonymize Tor clients [45, 46].

There are essentially two ways for an adversary to gain visibility into Tor traffic, either by compromising (or owning enough) Tor relays or by manipulating the underlying network communications so as to put herself on the forwarding path for Tor traffic. Regarding network threats, large Autonomous Systems (ASes) such as Internet Service Providers (ISPs) can easily eavesdrop on a portion of all links, and observe any unencrypted information, packet headers, packet timing, and packet size. Recent declarations by Edward Snowden have confirmed that ASes poses a real threat. Among others, the NSA has a program called Marina which stores meta information about user communications for up to a year [15], while the GCHQ has a program called Tempora that stores meta-information for 30 days and buffers data for three days [36]. Also, and maybe more importantly, it has been shown that Tor was targeted by such adversaries in collusion with ASes [10, 12, 11].

In this paper, we present Raptor, a suite of novel traffic analysis attacks that deanonymize Tor users more effectively than previously thought possible. To do so, and unlike previous studies on AS-level adversaries [28, 26, 40], Raptor leverages the *dynamic aspects* of the Internet routing protocol, *i.e.* the Border Gateway Protocol (BGP).

Raptor attacks are composed of three individual attacks whose effects are compounded (§2). First, Raptor exploits the asymmetric nature of Internet routing: the BGP path from a sender to a receiver can be different

	Traffic Analysis	BGP Churn	BGP Hijack	BGP Interception
Symmetric	Known [45, 46]	*Novel* (§4)	*Novel* (§5)	*Novel* (§5)
Asymmetric	*Novel* (§3)			

Table 1: This paper describes Raptor, a suite of previously unknown attacks on the Tor Network

than the BGP path from the receiver to the sender. Internet routing asymmetry increases the chance of an AS-level adversary observing at least one direction of both communication endpoints, enabling a novel asymmetric traffic analysis attack. Second, Raptor exploits natural churn in Internet routing: BGP paths change over time due to link or router failures, setup of new Internet links or peering relationships, or changes in AS routing policies. Changes in BGP paths allow ASes to observe additional Tor traffic, enabling them to deanonymize an increasing number of Tor clients over time. Third, Raptor exploits the inherent insecurity of Internet routing: strategic adversaries can manipulate Internet routing via BGP hijack and BGP interception attacks against the Tor network. These attacks enable the adversary to observe user communications, and to deanonymize clients via traffic analysis.

Raptor attacks were briefly discussed in a preliminary and short workshop paper [48]. In this paper, we go further by measuring the importance of the attacks using real-world Internet control- and data-plane data. We also demonstrate the attacks feasibility by performing them on the live Tor network—*with success*. No real Tor users were harmed in our experiments (§7). Finally, we also describe efficient countermeasures to restore a good level of anonymity. To summarize, we make the following key contributions:

Asymmetric Traffic Analysis and BGP Churn: Using live experiments on the Tor network, we showed that Raptor's asymmetric traffic analysis attacks can deanonymize a user with a 95% accuracy, without any false positives (§3). Using historical BGP and Traceroute data, we showed that by considering routing asymmetry and routing churn, the threat of AS-level attacks increases by 50% and 100%, respectively (§4).

BGP Hijacks and Interceptions: We analyzed known BGP hijacks and interception attacks on the Internet and show multiple instances where Tor relays were among the target prefixes (§5). As an illustration, the recent Bitcoin Hijack attack [1] in 2014, as well as Indosat Hijack attacks [3, 2] in 2014 and 2011 involved multiple Tor relays. To demonstrate the feasibility of such attacks for the purpose of deanonymizing Tor clients, we successfully performed an interception attack against a live Tor relay. Overall, we found that more than 90% of Tor relays are vulnerable to our attacks.

Countermeasures: We present a comprehensive taxonomy of countermeasures against Raptor attacks (§6). In particular, we outline the design of a monitoring framework for the Tor network that aims to detect suspicious AS-level path changes towards Tor prefixes using both BGP and Traceroute monitoring.

2 Raptor Attacks

To communicate with a destination, Tor clients establish layered circuits through three subsequent Tor relays. The three relays are referred to as: *entry* (or *guard*) for the first one, *middle* for the second one, and *exit* relay for the last one. To load balance its traffic, Tor clients select relays with a probability that is proportional to their network capacity. Encryption is used to ensure that each relay learns the identity of only the previous hop and the next hop in the communications, and no single relay can link the client to the destination server.

It is well known that if an attacker can observe the traffic from the destination server to the exit relay as well as from the entry relay to the client (or traffic from the client to the entry relay and from the exit relay to the destination server), then it can leverage correlation between packet timing and sizes to infer the network identities of clients and servers (end-to-end timing analysis). This timing analysis works even if the communication is encrypted.

In the rest of the section, we present the three Raptor attacks and how they contrast to conventional symmetric traffic analysis. We start by discussing how seeing just one direction of the traffic for each segment (between the sender and the guard, and between the last relay and the destination) is sufficient for the adversary (§2.1). We then explain how ASes can exploit natural BGP dynamics (§2.2), or even launch active attacks (§2.3), to compromise the anonymity of Tor users.

2.1 Asymmetric Traffic Analysis

We propose asymmetric traffic analysis, a novel form of end-to-end timing analysis that allows AS-level adversaries to compromise the anonymity of Tor users. Let us suppose that a Tor client is uploading a large file to a Web server. Conventional traffic analysis considers only one scenario where adversaries observe traffic from the

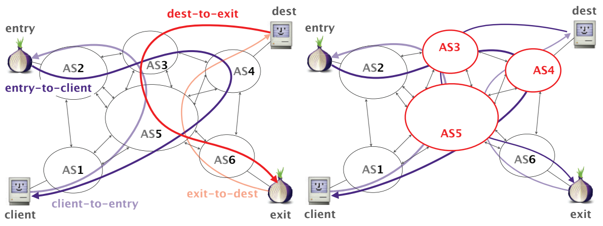


Figure 1: Asymmetric routing increases the power of AS-level adversaries. When considering forward traffic, i.e., client-to-entry and exit-to-destination flows, only AS5 can compromise anonymity. When considering both forward and backward traffic though, AS3, AS4 and AS5 can compromise anonymity. Our measurements confirm that asymmetric traffic analysis is feasible.

client to the entry relay, and from the exit relay to the Web server (same direction as the flow of traffic)[1].

However, Internet paths are often asymmetric: the path from the exit relay to the Web server may be different than the path from the Web server to the exit relay. Thus it is possible that an adversary may not be able to observe the data traffic on the path from the exit relay to the server, but it observes the TCP acknowledgment traffic on the path from the server to the exit relay.

We introduce an asymmetric traffic analysis attack that allows an adversary to deanonymize users as long as the adversary is able to observe *any direction of the traffic*, at both ends of the communication. Note that we can view the conventional end-to-end timing analysis as a special case of our attack, in which the adversary is able to observe traffic at both ends of the anonymous path, and in the same direction as the flow of traffic. Routing asymmetry increases the number of ASes who can observe at least one direction of traffic at both communication endpoints. We illustrate this scenario in Figure 1.

More concretely, our attack is applicable to four scenarios where an adversary observes (a) data traffic from the client to entry relay, and data traffic from exit relay to the server, or (b) data traffic from the client to entry relay, and TCP acknowledgment traffic from the server to exit relay, or (c) TCP acknowledgment traffic from guard relay to the client, and data traffic from exit relay to the server, or (d) TCP acknowledgment traffic from guard relay to the client, and TCP acknowledgment traffic from the server to the exit relay.

A key hurdle in asymmetric traffic correlation is that TCP acknowledgments are cumulative, and there is not a one-to-one correspondence between data packets and the TCP acknowledgment packets. We overcome this hurdle by observing that Tor (and other anonymity systems) use SSL/TLS for encryption, which leaves the TCP header unencrypted. Our attack inspects TCP headers in the observed traffic to retrieve the TCP sequence number field and TCP acknowledgment number field, and analyzes the correlation between these fields of both ends over time. Our experimental results in Section 3 show the feasibility of asymmetric traffic analysis, with a detection accuracy of 95%. Furthermore, asymmetric traffic analysis can be combined with other Raptor attacks, such as exploiting natural churn and BGP interception attack, which we discuss next.

2.2 Natural Churn

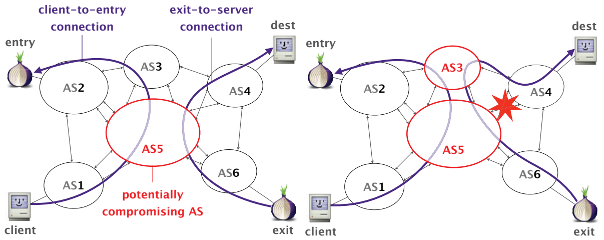


Figure 2: BGP churn increases the number of ASes that can deanonymize Tor traffic. Initially, only AS5 can deanonymize the client, seeing both direction of the traffic (left). After the failure of link (AS4, AS5), both AS5 and AS3 can deanonymize Tor traffic (right).

When users communicate with recipients over multiple time instances, then there is a potential for compromise of anonymity at every communication instance [49, 42]. Thus anonymity can degrade over time. Tor considers this threat from the perspective of adversarial relays (but not adversarial ASes).Tor clients use a fixed entry relay (guard relay) for a period of time (Dingledine et al. recommend 9 months [24]) to mitigate this threat with respect to adversarial relays. We note that the threat of AS-level adversaries still persists, because even though the entry relay is fixed, the set of ASes on the path between the client and the guard relay may change over time. Next, we discuss such attacks that rely on natural churn in BGP paths.

The underlying Internet paths between a client and guard relay vary over time due to changes in the physical topology (*e.g.*, failures, recoveries, and the rollout of new routers and links) and AS-level routing policies (*e.g.*, traffic engineering and new business relationships). These changes give a malicious AS surveillance power that increases over time. For example, AS 3 in Figure 2 does not lie on the original path from the exit to the des-

[1] If the traffic is flowing from the server to the client, then end-to-end timing analysis considers a scenario where the adversary observes traffic from the Web server to the exit relay and from the entry relay to the client.

tination, but a BGP routing change can put AS 3 on the path, allowing it to perform traffic analysis.

In Section 4, we show that the surveillance capability of an AS-level adversary can increase up to 50% when considering BGP churn over a period of one month.

2.3 BGP Hijack

So far, we discussed Raptor attacks that were *passive*. Strategic AS-level adversaries are also capable of launching *active* attacks, that deviate from honest routing behavior. Internet routing is vulnerable to attacks which enable an AS to manipulate inter-domain routing by advertising incorrect BGP control messages. While these attacks are well known in the networking community, we are the first to apply these attacks to anonymity systems such as Tor.

AS-level adversaries can hijack an IP prefix [51] by advertising the prefix as its own. The attack causes a fraction of Internet traffic destined to the prefix to be captured by the adversary. Tor relay nodes can observe a large amount of client traffic. For example, a Tor guard relay observes information about client IP addresses. Thus, the IP prefixes corresponding to Tor guard relays presents an attractive target for BGP hijack.

As a concrete attack example, we consider a scenario where an AS-level adversary aims to deanonymize the user associated with a connection to a sensitive Web server (say a whistleblowing website). The adversary can first use existing attacks on the Tor network to uncover the identity of the client's guard relay [39, 37, 31, 42]. Next, the adversary can launch a BGP hijack attack against the Tor relay. This allows the adversary to see traffic destined to the guard relay. BGP hijack thus enables an adversary to learn the set of all client IP addresses (anonymity set) associated with a guard relay (and the target connection to the sensitive Web server).

We note that in a prefix-hijack attack, the captured traffic is blackholed, and the client's connection to the guard is eventually dropped. Thus, it may not be possible to perform fine-grained traffic analysis to infer the true client identity from this anonymity set. However, the identification of a reduced anonymity set (as opposed to the entire set of Tor users) is already a significant amount of information leakage, and can be combined with other contextual information to break user anonymity [18]. In Section 5, we uncover several real-world BGP hijack attacks in which Tor relays were among the target prefixes.

2.4 BGP Interception

Our BGP hijack attack discussed above allows adversaries to capture traffic destined towards a target Tor prefix, but the captured traffic is blackholed, resulting in the connecting being dropped. Next, we discuss a more sophisticated routing attack called BGP interception attack [16], that allows adversaries to perform exact deanonymization of Tor users.

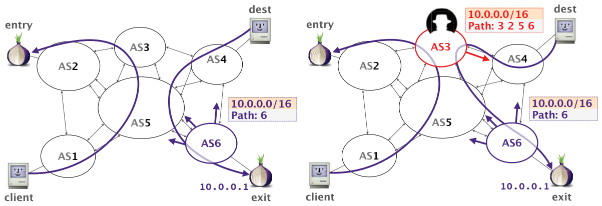


Figure 3: BGP interception attack enables ASes to selectively put themselves on some path. Here, AS3 only sees traffic between the client and the entry relay (left). By intercepting the prefix containing the exit relay (right), AS3 also sees traffic towards the exit relay, enabling it to deanonymize the Tor communication.

A prefix interception attack allows the malicious AS to become an intermediate AS in the path towards the guard relay, *i.e.*, after interception, the traffic is routed back to the actual guard relay. Such an interception attack allows the connection to be kept alive, enabling the malicious AS to exactly deanonymize the client via asymmetric traffic analysis.

Similar to the previous discussion, let us consider an adversary trying to deanonymize the user connecting to a sensitive website (the adversary already sees the traffic towards the website). The adversary can first uncover the identity of the guard relay using existing attacks [39] (as before), and then launch a prefix interception attack against the guard relay. Since the adversary routes the traffic back to the guard relay, the client's connection is kept alive, allowing the adversary to launch asymmetric traffic correlation attacks. Note that in contrast to BGP hijack attacks, BGP interception attacks can perform exact deanonymization of Tor clients.

These attacks enable malicious ASes to deanonymize user identity corresponding to a monitored target connection. Similarly, ASes that already see the client's traffic to its guard can position themselves to observe the traffic between the server and the exit relay by launching interception attacks against exit relays. Figure 3 illustrates this attack scenario.

Finally, we note that a remote adversary can launch interception attacks against both guard relays and exit relays simultaneously, to perform general surveillance of the Tor network. In Section 5, we demonstrate a real-world BGP interception attack against a live Tor relay by collaborating with autonomous system operators.

3 Asymmetric Traffic Analysis

In this section, we experimentally show that asymmetric traffic analysis attacks are feasible. We use the live Tor network for our experiments. To protect the safety of real Tor users, we generate our own traffic through the Tor network. Our goal is to investigate the accuracy of asymmetric traffic analysis in deanonymizing our generated traffic.

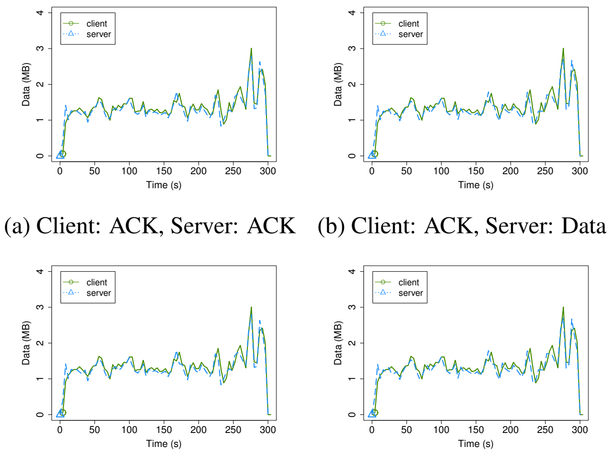
Experimental Setup: In order to generate our own traffic through the live Tor network, we use PlanetLab nodes as clients and Web servers. PlanetLab is an open platform for networking research, that provides access to hundreds of geographically distributed machines. We randomly pick 100 machines on PlanetLab, located across United States, Europe, and Asia. We installed Tor clients on 50 of those machines, and used the Privoxy tool (`www.privoxy.org`) to configure wget requests to tunnel over Tor. The remaining 50 machines were setup to be Web servers, each containing a 100*MB* image file.

We use the default Tor configuration on the 50 client machines. We launch wget requests on the 50 clients at the same time, each requesting a 100MB image file from one of the 50 web servers, respectively. We use tcpdump to capture data for 300 seconds at the clients and the servers during this process.

Asymmetric correlation analysis: In each packet trace, we first extract the TCP sequence number and TCP acknowledgment number fields in the TCP header. Using the TCP sequence and acknowledgment numbers, we next compute the number of transmitted data bytes per unit time. For each pair of observed traces, we compute the correlation between the vector of transmitted data bytes over time. For our analysis, we use the Spearman's rank correlation coefficient (other correlation metrics could also be applicable). For each client, our asymmetric traffic analysis attack selects the server trace with the highest correlation as the best match.

Results: Figure 4 illustrates our asymmetric analysis computed between a client server pair that is communicating. We can see high correlation in all four observation scenarios discussed in Section 2. Figure 5 illustrates our asymmetric analysis computed between a client server pair that is not communicating with each other. We can see that incorrectly matched pairs have poor correlation in all four observation scenarios. Figure 6 illustrates the detection accuracy rate grows as the duration of attack increases, especially in the first 30 seconds.

We computed the detection accuracy of our asymmetric traffic analysis attacks in all four scenarios after 300 seconds (by selecting the highest correlated pair), and obtained an average accuracy of 95% (Table 2). The error matches are all false negatives, for which the client has insignificant correlation coefficients with all servers, so it fails to be matched to any servers. We did not observe any false positives in our results.



(a) Client: ACK, Server: ACK (b) Client: ACK, Server: Data

(c) Client: Data, Server: ACK (d) Client: Data, Server: Data

Figure 4: Asymmetric traffic analysis shows high correlation between a matched client/server pair

	Client ACK/ Server ACK	Client ACK/ Server Data	Client Data/ Server ACK	Client Data/ Server Data
Overall	96%	94%	96%	94%
False negative	4%	6%	4%	6%
False positive	0%	0%	0%	0%

Table 2: Asymmetric traffic analysis accuracy rate

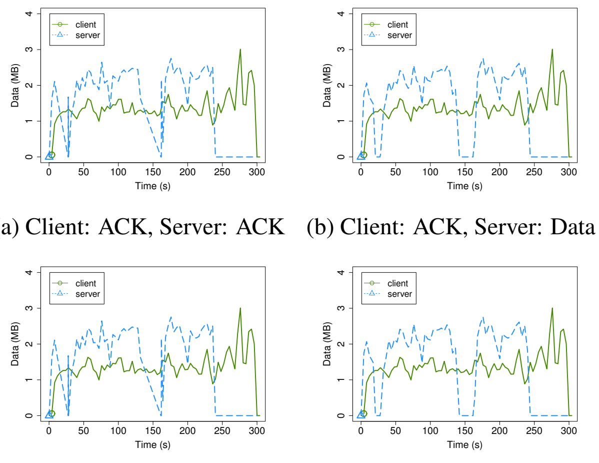
In addition to the actual observed error rate above, we also performed a statistical tests to compute the 95% confidence interval on our error rate, given our sample size of 50 client machines and 50 server machines. Table 3 illustrates the confidence intervals on our error rates.

	Client ACK/ Server ACK	Client ACK/ Server Data	Client Data/ Server ACK	Client Data/ Server Data
False negative	0.48% – 13.71%	1.25% – 16.54%	0.48% – 13.71%	1.25% – 16.54%
False positive	0% – 0.15%	0% – 0.15%	0% – 0.15%	0% – 0.15%

Table 3: Asymmetric traffic analysis error rate confidence interval

4 Natural Churn

In this section, we study and evaluate how routing dynamics, or churn, increase the power of AS-level adversaries in anonymity systems such as Tor. We start with an exhaustive control-plane analysis using collected BGP



(a) Client: ACK, Server: ACK (b) Client: ACK, Server: Data

(c) Client: Data, Server: ACK (d) Client: Data, Server: Data

Figure 5: Asymmetric traffic analysis shows low correlation between an unmatched client/server pair

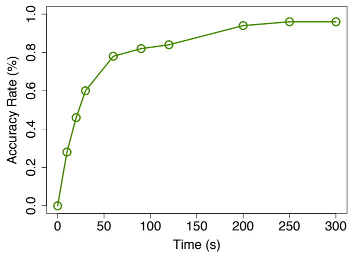


Figure 6: The accuracy of the attack quickly increases with time, reaching 80% within a minute, 95% after five minutes.

data (§4.1). Our results show that churn can increase the amount of compromised Tor circuits by up to 50% over a period of one month. We then confirmed our results by performing targeted data-plane measurements on the Tor network (§4.2). Again, churn significantly increased the percentage of vulnerable Tor circuits, nearly tripling it.

4.1 Control-plane Evaluation

We quantified the impact of churn by measuring how it increased the probability of a single AS (say AS X) to end up simultaneously on the path between a client and a guard relay and on the path between a destination and an exit relay. When this happens, we considered AS X as (potentially) compromising for the pair (client, destination) using the corresponding Tor circuit. Observe that our analysis leverages asymmetric traffic analysis (§3) as it only requires X to be on-path for two publicly-known prefixes, covering the guard and the exit relay.

Datasets We collected 612+ *million* BGP updates pertaining to 550,000 IP prefixes collected by six RIPE-maintained BGP Looking Glass (*rrc*00, *rrc*01, *rrc*03, *rrc*04, *rrc*11, *rrc*14) [6] in January 2015 over 250+ BGP sessions. We processed the dataset to remove any artifacts caused by session resets [20]. In parallel, we also collected Tor-related data (IP address, flags and bandwidth) of about 6755 Tor relays active during the same period of time [4]. Among all Tor relays, 1459 (resp. 1182) of them were listed as guards (resp. exits) and 338 relays were listed as both guard and exit.

We considered each BGP session as a proxy for Tor clients and destinations. Note that analysis implicitly accounts for *any* Internet host reachable directly or indirectly through these BGP sessions. Our dataset contains sessions belonging to major Internet transit providers such as Level-3, ATT, NTT, etc. that provide transit to millions of hosts.

Static baseline. We computed a static baseline by considering the amount of compromising ASes at the beginning of our dataset, *without considering any updates*. On each BGP session s_i, we computed and maintained the routing table used to forward Tor traffic by considering all the BGP announcements and withdrawals received over s_i. More precisely, we kept track of the most-specific routing table entry that was used to forward traffic to any Tor guard or exit relays. We refer to those as *Tor prefixes*. In this context, a routing table entry for a relay r is a five-tuple (t_i, t_f, p, e, L) composed of: *i)* the initial time t_i at which the entry started to be used by the router for forwarding traffic to r; *ii)* the final time t_f at which the entry stopped to be used by the router; *iii)* the corresponding IP prefix p; *iv)* a boolean e denoting whether the r is an entry or an exit relay; and *v)* the list of all the ASes L that will see the traffic en-route to reach r (*i.e.*, the `AS-PATH`).

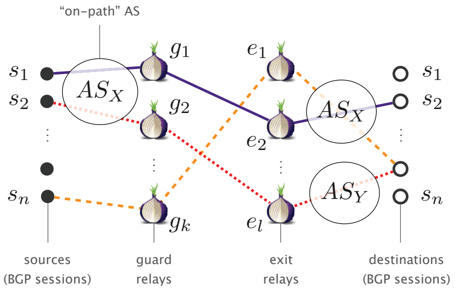
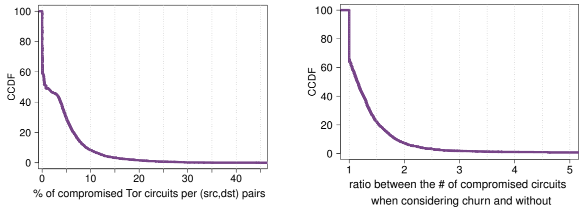


Figure 7: Control-plane evaluation setup

Using the routing-table data, we accounted, for each AS X, the number of pairs ((s_i, g_i), (s_j, e_j)) for which it appeared simultaneously in the `AS-PATH`. Here, s_i (resp. s_j) refers to a client (resp. destination) session, while g_i (resp. e_j) refers to a Tor guard (resp. exit) relay. To ensure meaningful results, we only considered cases in which s_i and s_j are in different ASes to ensure enough diversity in the paths seen. As illustration, in Fig. 7, AS_X is a compromising AS for the pair ((s_1, g_1), (s_2, e_2)), meaning it can deanonymize any clients connected beyond s_1 and exchanging data with a destination connected beyond s_2 which uses g_1 (resp. e_2) as a guard (resp. exit) relay.



(a) Static baseline (b) Churn-induced increase

Figure 8: Without considering churn, more than 5% of all the possible Tor circuits are compromised by at least one AS in 20% of the cases (left). The amount of compromised circuits increase for the majority of the (src, dest) pairs (60%) when considering churn, by up to 50% in 20% of the cases (right).

Fig. 8a depicts the percentage of compromised Tor circuits for each source and destination as a Complementary Cumulative Distribution Function (CCDF). A point (x,y) on the curve means that x% of all Tor circuits, *i.e.* `(guard,exit)` pairs, are compromised for at least y% of all the `(src,dst)` pairs. We see that, for 50% of all the `(src,dst)` pairs, *at least* 0.75% of the Tor circuits are compromised by at least one AS. This number grows to 6% and 13% of the Tor circuits considering the 75th- and 95th-percentile, respectively.

Measuring the effect of churn. We computed the number of extra Tor circuits that got compromised by at least one AS over one month. To be fair, we only considered a Tor circuit as compromised if it crossed the same AS for at least 30 seconds as it is unlikely that a time-correlation attack can be performed in shorter timescale. Fig. 8b plots the ratio between the amount of compromised Tor circuits for each `(src,dst)` pair at the end of the month with respect to the static baseline amount. We see that churn significantly increases the probability of compromise. Indeed, the amount of compromised circuits increase for 60% of the `(src,dst)`. The increase reaches 50% (ratio of 1.5) in 20% of the cases.

In addition to increasing the number of compromised Tor circuits, churn also increases the number of compromisable `(src,dst)` pairs. Indeed, while 5593 `(src,dst)` pairs could be compromised without updates, that number increased to 5754 pairs when considering updates (an augmentation of nearly 3%).

Few powerful ASes see some traffic for a large majority of the Tor circuits. Due to their central position in the Internet, a few ASes naturally tend to see a lot of Tor traffic crossing them. To account for this effect, we compute how many Tor circuits crossed each AS from at least one `(src,dst)` pair. The top 10 ASes in terms of compromised circuits are listed in Table 4. Large networks such as NTT or Level3 are able to see Tor traffic for up to 90% of Tor circuits.

Name	ASN	Tor circuits (%) seen	Country
NTT	2914	91	US
IIJ	2497	91	Japan
BroadbandONE	19151	91	US
Inet7	13030	91	CH
Level3	3356	88	US
Tinet	3257	86	DE
Cogent	174	63	US
Level3/GBLX	3549	58	US
TATA AMERICA	6453	53	US
TeliaSonera	1299	50	SWE

Table 4: A few well-established ASes simultaneously see some traffic for up to 90% of all (entry, exit) relays pairs.

4.2 Data-plane Evaluation

Next, we aim to quantify the impact of churn using data-plane information collected via `traceroute`.

Datasets We ran `traceroute` between 70 RIPE Atlas probes [5] to measure the actual forwarding path taken by packets entering and exiting the Tor network. We selected one probe in 70 different ASes, split in the following four sets:

- S_1 : 10 ASes that contain the most Tor clients [35];
- S_2 : 25 ASes that cumulatively contained ~50% of all guard relay bandwidth;
- S_3 : 25 ASes that cumulatively contained ~50% of all exit relay bandwidth;
- S_4 : 10 ASes that contain the most Tor destinations [35].

We then ran daily traceroutes over a 3 weeks period between all probes in S_1 towards all probes in S_2 (and vice-versa), measuring the forwarding paths P_1 between Tor clients and guard relays, and the paths P_2 between guard relays and Tor clients. Similarly, we measured the forwarding paths P_3 between exit relays and Tor destinations, and the paths P_4 between Tor destinations and exit relays. Overall, we measured $10 \times 25 \times 25 \times 10 = 62500$ possible Tor circuits.

Churn nearly tripled the amount of vulnerable Tor circuits. If we use conventional methodology and only look for common ASes between P_1 and P_3, we found 12.8% of Tor circuits to be vulnerable on the first day of the experiment (red line in Fig. 9). In comparison,

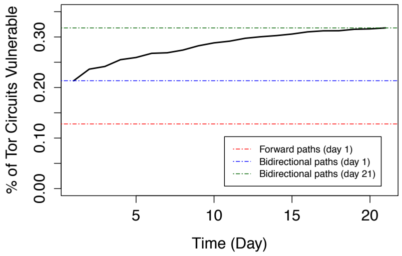


Figure 9: Percentage of Tor circuits vulnerable to an AS level adversary

if we also consider asymmetric paths (*i.e.*, also look for common ASes between P_1 and P_4, P_2 and P_3, and P_2 and P_4), the percentage of vulnerable Tor circuits nearly doubled to 21.3% on the first day (blue line in Fig. 9), and nearly tripled to 31.8% at the end of the three week period (green line in Fig. 9).

5 BGP Attacks: Hijack and Interception

In this section, we study and evaluate the feasibility of BGP hijack and interception attacks on the Tor network. First, we show that Tor relays tend to be concentrated within few ASes and IP prefixes—making those highly attractive targets for hijack and interception attacks (§5.1). Second, we show that, in several real-world BGP hijack attacks, Tor relays were among the target prefixes (§5.2). Third, we perform a real-world BGP interception attack against a live Tor guard relay, *with success*, to demonstrate the ability to accurately deanonymize Tor clients (§5.3).

5.1 Tor relays concentration

The amount of Tor traffic attracted by a hijack or an interception attack depends on the number of relays that lie within the corresponding prefix. As such, prefixes and ASes that host many relays of high bandwidth are an interesting targets for attackers. To evaluate how vulnerable the Tor network was to hijack and interceptions attacks, we computed the number of relays present in each AS and in each BGP prefix. Surprisingly, close to 30% of all relays are hosted in only 6 ASes and 70 prefixes. Together, these relays represent almost 40% of the bandwidth in the entire Tor network (see Table 5). As such, these few prefixes constitute *extremely* attractive targets.

	% relays	% bw	# pfx	Name	ASN
	10.5	23	11.80	OVH	16276
	6.30	13	6.68	Hetzner	24940
	4.78	7	10.52	Online.net	12876
	3.04	4	2.58	Wedos	197019
	2.04	14	4.27	Leaseweb	16265
	1.69	9	3.86	PlusServer	8972
Total	28.35	70	39.71		

Table 5: 6 ASes and 70 prefixes host ~30% of all Tor guard and exit relays as well as ~40% of the entire Tor network bandwidth. As such, these constitute extremely attractive targets for hijacks and interceptions attacks.

5.2 Known Prefix Hijacking Attacks

While there have been numerous well-documented BGP prefix hijacks and interceptions, it was unknown whether Tor traffic was intercepted or not. To this extent, we studied occurrences of well-known prefix hijacks and looked for leaked prefixes covering at least a Tor relay. To do so, we gathered BGP updates from Routeviews [7] around the time of each attack and filtered out the ones unrelated to Tor prefixes. Overall, we found that three well-known hijacks affected Tor relays: two separate incidents involving one of Indonesia's largest telecommunication networks, Indosat, as well as one malicious hijack attack whose goal was to steal Bitcoins.

Event	# hijacked relays	# hijacked guards	# hijacked exits
Indosat 2011	5 (0.24%)	1 (0.15%)	4 (0.44%)
Indosat 2014	44 (0.80%)	38 (1.80%)	17 (1.65%)

Table 6: Summary statistics for known Indosat prefix hijacking events.

Indosat 2011. On January 14th, 2011, Indosat (AS4761) originated 2,800 new prefixes, which covered 824 different ASes [2]. 7 of these prefixes affected the Tor network by covering 5 of the Tor relays. As discussed in Section 2, Indosat *could* potentially have learned information about the client IP addresses associated with each of the guard relays (reduced anonymity set).

Indosat 2014. On April 3, 2014, Indosat originated 417,038 new prefixes; it usually originates 300 prefixes [3]. This compromised 44 Tor relays, 38 of which were guard relays and 17 of which were exit relays (11 hijacked relays were both guards and exits). Table 6 shows the summary statistics of both Indosat hijacking incidents.

Canadian Bitcoin 2014. From February 2014 to May 2014, an attacker compromised 51 networks at

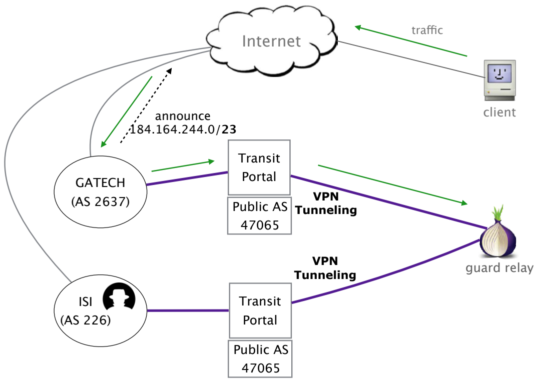


Figure 10: Transit Portal setup

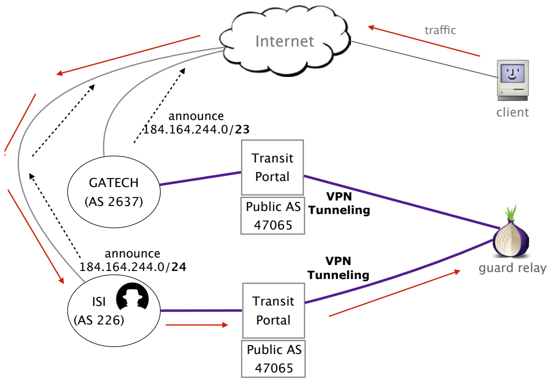


Figure 11: ISI performing interception attack

19 different ISPs, and resulted in the theft of approximately $83,000 in Bitcoin [1]. We found that 198.245.63.0/24 and 162.243.142.0/24 were hijacked, and contained a Tor relay, 198.245.63.228. AS16276 (OVH) owns 198.245.63.0/24, but this prefix was hijacked by AS21548 (MTO Telecom). The Tor relay that consequently was hijacked, 198.245.63.228, was a guard relay located in Montreal, Quebec.

While we do not make any claims about the intent of the above hijacking ASes, our analysis shows the existential threat of real-world routing attacks on the Tor network. Furthermore, the fact that the Tor and the research community missed noticing the presence of Tor relays among the hijacked prefixes is surprising.

5.3 BGP Prefix Interception Attack Experiment

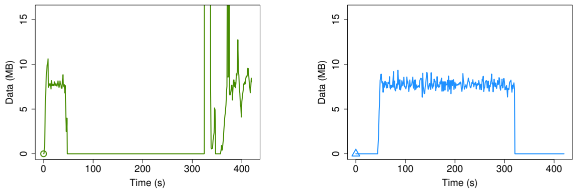
Methodology and setup. We now demonstrate the feasibility of the interception attack by performing one, with success, on the live Tor network. For that, we set up a machine to run as a Tor guard relay and made it reachable to the Internet by announcing a /23 prefix in BGP using Transit Portal (TP) [43]. TP enables virtual ASes to establish full BGP connectivity with the rest of the Internet by proxying their announcements via dozens of worldwide deployments. Next, we configure the 50 Tor clients in PlanetLab to use our Tor guard relay as the entry relay to reach 50 web servers, also hosted in PlanetLab.

In order to perform the BGP prefix interception attack, we used two TP deployments (`GATECH` and `ISI`), located in different ASes. GATECH TP served the purpose of the "good" AS through which Tor traffic is normally routed, while ISI TP served as the "malicious" AS which performed the interception attack. We connected the two TPs to our Tor relay machine via VPN tunnels. First, in order for our Tor guard relay (running on `184.164.244.1`) to be reachable, we advertised 184.164.244.0/23 (run by us in its entirety for the duration of our research) via the GATECH TP, so that traffic destined for IP addresses within that range will be routed, first to the GATECH TP, and then sent to our machine via the corresponding tunnel. We illustrate our setup in Fig. 10.

Next, We advertise BGP prefix `184.164.244.0/24` via the ISI TP, which constitutes a more-specific prefix attack against the original announcement announced by the GATECH TP. Thus, after the new BGP prefix announcement gets propagated through the internet, Tor client traffic that is destined for our guard will be sent to ISI instead. Since we configured the ISI TP to forward traffic to our guard machine, the Tor relay can still receive the traffic and keep the Tor connection alive after the attack. We illustrate the interception attack model in Fig. 11.

Our setup constitutes a BGP interception attack. Initially, traffic is routed via GATECH and arrives at our Tor relay machine via GATECH tunnel. After the attack happens, traffic drains from GATECH tunnel and gets routed via ISI, and thus comes to our Tor relay machine via ISI tunnel instead. Since the traffic still arrives at the relay machine, it is an interception attack and the connection does not get interrupted. We use `tcpdump` on our relay machine, listening to ISI tunnel, to capture client TCP acknowledgment traffic coming from that tunnel, which is exactly the data that an adversary would get from launching such an interception attack.

In the experiment, we first launch simultaneous HTTP requests using `wget` at the 50 Tor clients for the 100MB file at the 50 web servers. Then, 20 seconds after launching the `wget` requests, we start announcing the more-specific prefix via ISI. We use `tcpdump` listening to ISI tunnel to capture TCP acknowledgment traffic sent from the Tor clients during the interception attack. We also use `tcpdump` to capture traffic at the web servers during the whole process. Finally, 300 seconds after launching the



(a) Traffic Flow via GATECH (b) Traffic Flow via ISI

Figure 12: Traffic Flow During the Experiment

attack, we send a withdrawal message via the ISI TP, so the traffic will be routed via GATECH again as normal.

Our interception attack successfully deanonymized Tor sources with a 90% accuracy rate. In Fig. 12, we plot the Tor traffic flow captured on our relay machine from both GATECH tunnel and ISI tunnel. We can see that all traffic is routed via GATECH at the beginning. At $t = 20s$, ISI starts advertising a more specific /24 prefix, which takes approximately 35 seconds for it to be propagated through the internet and drain the traffic from GATECH. At $t = 55s$, traffic starts showing up via ISI, and GATECH does not receive traffic any more. Then, at $t = 300s$, ISI withdraws the IP prefix announcement, which takes approximately 22 seconds for the traffic to appear back on GATECH again. During this interception process, the connection stays alive.

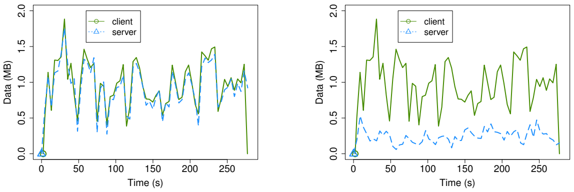
The captured data from ISI tunnel is client TCP acknowledgment traffic. Thus, we will employ our Asymmetric Traffic Analysis approach, described in Section 3, with sample size of 50 client machines and 50 server machines to do the correlation analysis to deanonymize users' identity. We achieve 90% accuracy rate (see Table 7).

	Accuracy Rate	False Negative	False Positive
Client ACK/Server ACK	90%	8%	2%

Table 7: Asymmetric Traffic Analysis accuracy rate

Fig. 13 shows an example of a client with its correlated server and an uncorrelated server, respectively. Note that the time shown on the graph has been adjusted according to the time that traffic starts showing via ISI.

The detection accuracy rate in the interception attack case decreases from the average 95% in static asymmetric traffic analysis to 90%. One main reason is that we configure all 50 Tor clients to connect to the same Tor guard relay, which leads to significantly higher probability that many of them will share the same Tor exit relay (especially those clients which are in the same AS) as well, and as a result, their bandwidths are highly likely to be similar. And also, all the clients start requesting files from the web servers at the same time, so the bandwidth they could achieve will be limited by the guard and the exit relay, which leads to similar bandwidths due to the guard/exit bottleneck. However, this scenario is an extreme and very unlikely case in real Tor connections. With fewer clients connecting to the same Tor guard relay at the same time, the accuracy of the asymmetric traffic analysis should be higher.



(a) Client & Correlated Server (b) Client & Uncorrelated Server

Figure 13: Client ACK versus Server ACK analysis

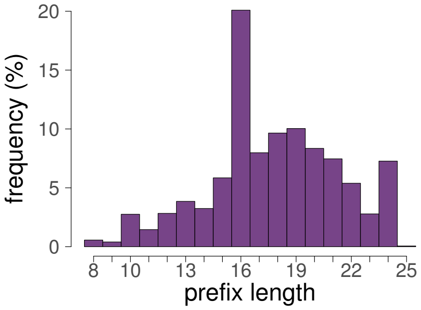


Figure 14: >90% of BGP prefixes hosting relays are shorter than /24, making them vulnerable to our attack.

The vast majority of Tor relays are vulnerable to our attacks. Technically, only prefixes shorter than /24 can be hijacked globally with a more-specific prefix attack as longer prefixes tend to be filtered by default by many ISPs. To make sure of the feasibility of our attack, we computed the prefix length distribution of Tor prefixes (see Fig.14). We can see that *more than* 90% of BGP prefixes hosting relays have prefix length shorter than /24, making them directly vulnerable to a more-specific prefix attack such as ours.

6 Countermeasures Sketch

In this section, we first describe a taxonomy of countermeasures against Raptor attacks. Second, we describe a general approach for AS-aware anonymous communication in which Tor clients are aware of the dynamics of Internet routing. Finally, we describe exploratory approaches for detecting and preventing BGP hijack and interception attacks against Tor.

6.1 Countermeasure Taxonomy

There are two main categories of countermeasures: (a) approaches that reduce the chance of an AS-level adversary observing both ends of the anonymous communication, and (b) approaches that aim to mitigate correlation attacks even when an adversary observes both ends of the anonymous communication. Figure 15 illustrates the design space of potential countermeasures against Raptor attacks. In this work, we advocate the former line of defense – namely, to monitor both routing control-plane and data-plane, and to strategically select Tor relays that minimize the chance of compromise (§6.2). We also advocate defenses that aim to detect and prevent routing attacks (§6.3). We do not focus on the class of approaches that aim to mitigate correlation analysis by obfuscating packet sizes and timings, as they are generally considered too costly to deploy (Appendix A).

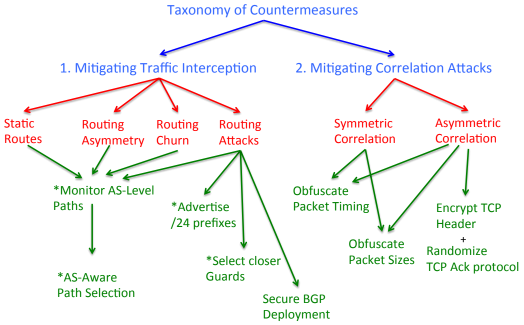


Figure 15: Taxonomy of Countermeasures

6.2 AS-Aware Path Selection

To minimize opportunities for AS-level traffic analysis, the Tor network can monitor the path dynamics between the clients and the guard relays, and between the exit relays and the destinations. Information about path dynamics can be obtained using data-plane (*e.g.*, `traceroute`) or control-plane (*e.g.*, BGP feed) tools. For instance, each relay could publish the list of any ASes it used to reach each destination prefix in the last month. This information can be distributed to all Tor clients as part of the Tor network consensus data. Tor clients can use this data in relay selection, perhaps in combination with their own traceroute measurements of the forward path to each guard relay. For example, Tor clients should select relays such that the same AS does not appear in both the first and the last segments, after *taking path dynamics into account*.

6.3 Mitigating Routing Attacks in Tor

Next, we consider two approaches for mitigating Raptor's routing attacks: detection and prevention.

6.3.1 Monitoring Framework for Detection Routing Attacks

We propose that the Tor network monitor the routing control-plane and data-plane for robust detection of routing attacks. Detecting routing attacks serves two purposes: (1) First, this serves to raise awareness about the problem and hold attackers accountable. (2) Second, Tor directory authorities can notify clients. Such notifications allow the end-user to respond by either suspending its use of Tor (since most hijacks and interceptions are short lived), or choose another Tor relay . Next, we discuss two proof-of-concept monitoring frameworks, based on BGP data and traceroute data respectively.

BGP Monitoring Framework. Our BGP monitoring framework gathers BGP data from the Routeviews project. The framework filters BGP updates to consider data about prefixes that involve a Tor relay. Building upon prior work in routing attack detection [16], we implement the following heuristics. (1) *Frequency heuristic:* routing attacks can be characterized by an AS announcing a path once (or extremely rarely) to a prefix that it does not own. The frequency heuristic detects attacks that exhibit this behavior. It measures the frequency of each AS that originates a given prefix; if the frequency is lower than a specified threshold, then it could be a potential hijack attack. (2) *Time Heuristic.* Most known attacks, including those discussed in § 5, last a relatively short amount of time. The time heuristic measures the amount of time each path to a prefix is announced for; if the amount of time is extremely small (below a specified threshold), then there is the possibility of it being a routing attack.

Detection Capability: We tested our BGP monitoring framework based on BGP data during known prefix hijack attacks, that were discussed in § 5. As a preliminary validation, the frequency and time heuristics were able to detect all of the known attacks; the threshold used for the frequency heuristic was .00001 (the fraction {# of announcements for prefix p originated by AS A}/{total # of announcements for prefix p}), and the threshold used for the time heuristic was .01 (the fraction {length of time that prefix p is originated by AS A}/{total length of time prefix p is announced by any AS}).

Traceroute Monitoring Framework. The BGP monitoring framework provides measurements of actual AS-level paths from BGP collector nodes. However, the input data to the monitoring framework is limited to peers who chose to participate in frameworks such as Routeviews, and BGP data is only a noisy indicator of the routing control-plane. For robust detection of attacks, it is also necessary to monitor the data-plane, which we do via a Traceroute monitoring framework.

Traceroute is a network diagnostic tool that infers the routers traversed by internet packets. To analyze both attacks and changes in AS-level paths to the Tor network, we have built a traceroute monitoring framework that runs traceroutes from 450 PlanetLab machines to all Tor entry and exit relays and stores the resulting traceroute data. The set of all Tor entry and exit relays is updated daily to accommodate new relays that have received the guard and exit flags. BGP hijack and interception attacks typically affect a variety of users from different vantage points. Thus, traceroute measurements from 450 geographically diverse PlanetLab have the ability to detect data-plane anomalies arising out of routing attacks. The PlanetLab machines are distributed across 140 ASes. Meanwhile, the Tor entry relays are distributed across 982 ASes and the exit relays are distributed across 882 ASes. We use Team-Cymru (`http://www.team-cymru.org/`) to compute the mapping between an IP address and its autonomous system. We will make the data collected by our Traceroute monitoring framework available to the research community.

Detection Capability: As a preliminary validation, our Traceroute monitoring framework was able to detect the the BGP interception attack discussed in § 5. From the traceroute data, we observed AS-level path changes from *every* PlanetLab node to our Tor guard relay, indicating an anomaly.

6.3.2 Preventing Routing Attacks in Tor

In addition to monitoring the routing control-plane and data-plane with respect to the Tor network, the following approaches can help *prevent* the threat of Raptor's routing attacks.

Advertising /24 Tor prefixes: Our experimental measurements indicate that over 90% of Tor relays have a prefix length shorter than /24. This allows an AS-level adversary to launch a BGP hijack or interception attack against these Tor relays by advertising a more specific prefix for them (globally). We advocate that the Tor relay operators should be running Tor relays with a prefix length of /24. Autonomous systems typically filter route advertisements of prefix longer than /24, so AS-level adversaries will not be able to launch a more specific hijack or interception attack.

Favoring closer guard relays: Even if a Tor relay advertises a /24 prefix, an AS-level adversary can launch an equally specific prefix hijack or interception attack (by advertising another /24). In this case, the impact of the attack is localized around the attacker's autonomous system, since the route is not globally propagated. We advocate that Tor clients select their guard relays by favoring Tor relays with a shorter AS-level path between them. Tor clients could either obtain AS-level path information via the Tor network consensus download mechanism, or they can perform traceroutes themselves. This further mitigates the risk to Tor clients due to an equally specific prefix attack. We note that by selecting guard relays that are closer to the client in the AS topology, the risk of asymmetric traffic analysis and BGP churn is also mitigated. [2]

Securing inter-domain routing: The research community has proposed multiple protocols for securing inter-domain routing [41, 32, 19, 29, 17]. Real-world deployment of these protocols would mitigate the BGP hijack and interception attacks on Tor. However, this approach requires buy-in from multiple stakeholders in the complex ecosystem of the Internet, and progress on this front has been slow. We hope that the concerns we raise about the compromise of user anonymity in Tor can help accelerate the momentum for improving BGP security.

7 Discussion and Ethical Considerations

Colluding adversaries. In this paper, we quantified the threat of Raptor attacks from the perspective of individual autonomous systems. In practice, autonomous systems can collude with each other to increase their capability of monitoring Tor traffic. For example, autonomous systems within the same legal jurisdiction may be forced to monitor Tor traffic and share it with a single entity that may launch Raptor attacks.

Applicability to other anonymity systems. It is important to note that our attacks merely consider Tor as an example of a low-latency anonymity system. Raptor attacks are broadly applicable to other deployed anonymity systems such as I2P, Freenet and Tribler [47, 21, 50].

Ethical considerations. We introduce and evaluate several novel attacks against the Tor network. The Tor network has a userbase of several million users [9], and these users are especially concerned about the privacy of their communications. Thus, it is of utmost importance that our real-world experiments on the Tor network do not compromise the privacy and safety of Tor users. In this paper, we take multiple precautions to safeguard the privacy of Tor users:

- *Attack our own traffic.* All of our attacks only experiment with traffic that we created ourselves, i.e., we deanonymize our own traffic. In fact, we do not store or analyze traffic of any real Tor user.

[2] We note that if clients select closer guards, then knowledge of the guards reveals probabilistic information about the clients. We will investigate this trade-off in future work.

- *Attack our own relay*. Similarly, to demonstrate the threat of prefix interception attacks on the live Tor network, we launch interception attacks against relays that we already control, i.e., we hijack/intercept our own prefix.

- *Firewall our Tor relay*. We also used network-level firewalls to ensure that real Tor users will never use relays that we control: traffic from real users is dropped by the firewall. Only authorized traffic that we create ourselves can bypass the firewall and use our Tor relay.

8 Related Work

AS-level adversaries: It is well known that an adversary who can observe users' communications at both ends of the segment can deanonymize Tor clients [45, 54]. Feamster and Dingledine were the first to consider the attack from the perspective of an AS-level adversary [28]. Later, Edman and Syverson explored the impact of Tor path selection strategies on the security of the network [26]. Recently, Johnson et al. analyzed the security of the Tor network against AS-level adversaries in terms of user understandable metrics for anonymity [34], and Akhoondi *et al.* [13] considered path selection algorithms that minimize opportunities for AS-level end-to-end traffic analysis. Finally, Murdoch *et al.* [40] considered the analogous analysis with respect to Internet exchange level adversaries, which are also in a position to observe a significant fraction of Internet traffic.

We build upon these works and introduce Raptor attacks, that leverage routing asymmetry, routing churn, and routing attacks to compromise user anonymity more effectively than previously thought possible.

The attack observations in Raptor were briefly discussed in a preliminary and short workshop paper [48]. In this paper, we go further by measuring the importance of the attacks using real-world Internet control- and data-plane data. We also demonstrate the attacks feasibility by performing them on the live Tor network—*with success*. Finally, we also describe efficient countermeasures to restore a good level of anonymity.

Traffic analysis of Tor: An important thread of research aims to perform traffic analysis of Tor communications via side-channel information about Tor relays. Murdoch et al. [39], Evans et al. [27], and Jansen et al. [33] have demonstrated attacks that use node congestion and protocol-level details as a side channel to uncover Tor relays involved in anonymous paths. Furthermore, Mittal et al. [37] and Hopper et al. [30, 31] proposed the use of network throughput and network latency as a side channel to fingerprint Tor relays involved in anonymous paths. We note that most of these attacks provide probabilistic information about Tor relays, and may not deanonymize the Tor clients. In contrast, Raptor attacks can completely deanonymize Tor clients.

BGP insecurity: The networking research community has extensively studied attacks on inter-domain routing protocols including BGP hijack [51, 52, 53, 44] and interception attacks [16]. Similarly, there has been much work on proposing secure routing protocols that resist the above attacks [41, 32, 17, 19, 29]. However, we are the first to study the implications of these attacks on privacy technologies such as the Tor network. Arnbak et al. [14] discuss surveillance capabilities of autonomous systems from a legal perspective, but do not discuss anonymity systems.

9 Conclusion

Raptor attacks exploit the dynamics of Internet routing (such as routing asymmetry, routing churn, and routing attacks) to enable an AS-level adversary to effectively compromise user anonymity.

Our experimental results show that Raptor attacks present a serious threat to the security of anonymity systems. Our key results include (1) demonstration of asymmetric traffic correlation on the live Tor network, which achieves 95% accuracy with no observed false positives, (2) quantifying the impact of routing asymmetry and routing churn on AS-level attacks – an increase of 50% to 100% respectively compared to conventional attacks, (3) uncovering historical BGP hijacks involving Tor relays, and (4) successful demonstration of a traffic analysis attack via BGP interception on the live Tor network. We also outlined a taxonomy of countermeasures against our attacks.

Our work highlights the dangers of abstracting network routing from the analysis of anonymity systems such as Tor, and motivates the design of next generation anonymity systems that resist Raptor.

10 Acknowledgments

Thanks to Ethan Katz-Bassett for support on setting up Transit Portal provided by the PEERING project. Thanks to ATLAS project for donating credits for our experimental setup. Thanks to Matthew Wright, Nick Feamster, Nikita Borisov and Roger Dingledine for helpful discussions. This work was supported by the NSF under the grant CNS-1423139.

References

[1] BGP hijacking for cryptocurrency profit. http://www.secureworks.com/

cyber-threat-intelligence/threats/bgp-hijacking-for-cryptocurrency-profit/.

[2] Bgpmon: Hijack by AS4761 Indosat - a quick report. http://www.bgpmon.net/hijack-by-as4761-indosat-a-quick-report/.

[3] BGPMon: Hijack Event Today by Indosat. http://www.bgpmon.net/hijack-event-today-by-indosat/.

[4] CollecTor: Your friendly data-collecting service in the Tor network. https://collector.torproject.org/.

[5] RIPE Atlas. https://atlas.ripe.net/.

[6] RIPE RIS Raw Data. https://www.ripe.net/data-tools/stats/ris/ris-raw-data.

[7] Routeviews. http://www.routeviews.org/.

[8] Tor metrics portal. https://metrics.torproject.org. Accessed, February 2015.

[9] Who uses Tor? https://www.torproject.org/about/torusers.html.en. Accessed, February 2015.

[10] How the NSA attacks Tor/Firefox users with QUANTUM and FOXACID. https://www.schneier.com/blog/archives/2013/10/how_the_nsa_att.html, Oct. 2013.

[11] Peeling back the layers of Tor with EgotisticalGiraffe. http://www.theguardian.com/world/interactive/2013/oct/04/egotistical-giraffe-nsa-tor-document, Oct. 2013.

[12] Tor stinks. http://www.theguardian.com/world/interactive/2013/oct/04/tor-stinks-nsa-presentation-document, Oct. 2013.

[13] Akhoondi, M., Yu, C., and Madhyastha, H. V. Lastor: A low-latency AS-aware Tor client. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2012), SP '12, IEEE Computer Society, pp. 476–490.

[14] Arnbak, A., and Goldberg, S. Loopholes for circumventing the constitution: Warrantless bulk surveillance on americans by collecting network traffic abroad. In *HotPETs* (2014). Available at http://ssrn.com/abstract=2460462.

[15] Ball, J. NSA stores metadata of millions of web users for up to a year, secret files show. http://www.theguardian.com/world/2013/sep/30/nsa-americans-metadata-year-documents, Sep. 2013.

[16] Ballani, H., Francis, P., and Zhang, X. A study of prefix hijacking and interception in the Internet. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 2007), SIGCOMM '07, ACM, pp. 265–276.

[17] Boldyreva, A., and Lychev, R. Provable security of S-BGP and other path vector protocols: Model, analysis and extensions. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 541–552.

[18] Brandom, R. FBI agents tracked harvard bomb threats despite Tor. http://www.theverge.com/2013/12/18/5224130/fbi-agents-tracked-harvard-bomb-threats-across-tor. Accessed, July 2014.

[19] Chan, H., Dash, D., Perrig, A., and Zhang, H. Modeling adoptability of secure BGP protocol. In *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 2006), SIGCOMM '06, ACM, pp. 279–290.

[20] chun Cheng, P., Zhao, X., Zhang, B., and Zhang, L. Longitudinal Study of BGP Monitor Session Failures. *ACM SIGCOMM Computer Communication Review (CCR)* (April 2010).

[21] Clarke, I., Sandberg, O., Wiley, B., and Hong, T. W. Freenet: A distributed anonymous information storage and retrieval system. In *International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobservability* (New York, NY, USA, 2001), Springer-Verlag New York, Inc., pp. 46–66.

[22] Danezis, G. Mix-networks with restricted routes. In *Privacy Enhancing Technologies*, R. Dingledine, Ed., vol. 2760 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2003, pp. 1–17.

[23] Danezis, G., Dingledine, R., and Mathewson, N. Mixminion: Design of a type iii anonymous remailer protocol. In *Proceedings of the 2003*

IEEE Symposium on Security and Privacy (Washington, DC, USA, 2003), SP '03, IEEE Computer Society, pp. 2–.

[24] DINGLEDINE, R., HOPPER, N., KADIANAKIS, G., AND MATHEWSON, N. One fast guard for life (or 9 months). In *HotPETs* (2014).

[25] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. Tor: The second-generation onion router. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13* (Berkeley, CA, USA, 2004), SSYM'04, USENIX Association.

[26] EDMAN, M., AND SYVERSON, P. AS-awareness in Tor path selection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2009), CCS '09, ACM, pp. 380–389.

[27] EVANS, N. S., DINGLEDINE, R., AND GROTHOFF, C. A practical congestion attack on Tor using long paths. In *Proceedings of the 18th Conference on USENIX Security Symposium* (Berkeley, CA, USA, 2009), SSYM'09, USENIX Association, pp. 33–50.

[28] FEAMSTER, N., AND DINGLEDINE, R. Location diversity in anonymity networks. In *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society* (New York, NY, USA, 2004), WPES '04, ACM, pp. 66–76.

[29] GILL, P., SCHAPIRA, M., AND GOLDBERG, S. Let the market drive deployment: A strategy for transitioning to BGP security. In *Proceedings of the ACM SIGCOMM 2011 Conference* (New York, NY, USA, 2011), SIGCOMM '11, ACM, pp. 14–25.

[30] HOPPER, N., VASSERMAN, E. Y., AND CHAN-TIN, E. How much anonymity does network latency leak? In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2007), CCS '07, ACM, pp. 82–91.

[31] HOPPER, N., VASSERMAN, E. Y., AND CHAN-TIN, E. How much anonymity does network latency leak? *ACM Trans. Inf. Syst. Secur. 13*, 2 (Mar. 2010), 13:1–13:28.

[32] HU, Y.-C., PERRIG, A., AND SIRBU, M. Spv: Secure path vector routing for securing BGP. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 2004), SIGCOMM '04, ACM, pp. 179–192.

[33] JANSEN, R., TSCHORSCH, F., JOHNSON, A., AND SCHEUERMANN, B. The sniper attack: Anonymously deanonymizing and disabling the Tor network. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS '14)* (2014), Internet Society.

[34] JOHNSON, A., WACEK, C., JANSEN, R., SHERR, M., AND SYVERSON, P. Users get routed: Traffic correlation on Tor by realistic adversaries. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2013), CCS '13, ACM, pp. 337–348.

[35] JUEN, J. Protecting anonymity in the presence of autonomous system and Internet exchange level adversaries, 2012. MS Thesis, University of Illinois at Urbana-Champaign.

[36] MACASKILL, E., BORGER, J., HOPKINS, N., DAVIES, N., AND BALL, J. GCHQ taps fibre-optic cables for secret access to world's communications. `http://www.theguardian.com/uk/2013/jun/21/gchq-cables-secret-world-communications-nsa`, June 2013.

[37] MITTAL, P., KHURSHID, A., JUEN, J., CAESAR, M., AND BORISOV, N. Stealthy traffic analysis of low-latency anonymous communication using throughput fingerprinting. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2011), CCS '11, ACM, pp. 215–226.

[38] MÖLLER, U., COTTRELL, L., PALFRADER, P., AND SASSAMAN, L. Mixmaster Protocol — Version 2. IETF Internet Draft, July 2003.

[39] MURDOCH, S. J., AND DANEZIS, G. Low-cost traffic analysis of Tor. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2005), SP '05, IEEE Computer Society, pp. 183–195.

[40] MURDOCH, S. J., AND ZIELIŃSKI, P. Sampled traffic analysis by Internet-exchange-level adversaries. In *Proceedings of the 7th International Conference on Privacy Enhancing Technologies* (Berlin, Heidelberg, 2007), PET'07, Springer-Verlag, pp. 167–183.

[41] OORSCHOT, P. V., WAN, T., AND KRANAKIS, E. On interdomain routing security and pretty secure BGP (psBGP). *ACM Trans. Inf. Syst. Secur. 10*, 3 (July 2007).

[42] OVERLIER, L., AND SYVERSON, P. Locating hidden servers. In *Security and Privacy, 2006 IEEE Symposium on* (May 2006), pp. 100–114.

[43] SCHLINKER, B., ZARIFIS, K., CUNHA, I., FEAMSTER, N., AND KATZ-BASSETT, E. PEERING: An AS for us. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2014), HotNets-XIII, ACM, pp. 18:1–18:7.

[44] SHI, X., XIANG, Y., WANG, Z., YIN, X., AND WU, J. Detecting prefix hijackings in the Internet with Argus. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference* (New York, NY, USA, 2012), IMC '12, ACM, pp. 15–28.

[45] SHMATIKOV, V., AND WANG, M.-H. Timing analysis in low-latency mix networks: Attacks and defenses. In *Proceedings of the 11th European Conference on Research in Computer Security* (Berlin, Heidelberg, 2006), ESORICS'06, Springer-Verlag, pp. 18–33.

[46] SYVERSON, P., TSUDIK, G., REED, M., AND LANDWEHR, C. Towards an analysis of onion routing security. In *International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobservability* (New York, NY, USA, 2001), Springer-Verlag New York, Inc., pp. 96–114.

[47] TIMPANARO, J. P., CHRISMENT, I., AND FESTOR, O. A bird's eye view on the I2P anonymous file-sharing environment. In *Proceedings of the 6th International Conference on Network and System Security* (Berlin, Heidelberg, 2012), NSS'12, Springer-Verlag, pp. 135–148.

[48] VANBEVER, L., LI, O., REXFORD, J., AND MITTAL, P. Anonymity on quicksand: Using BGP to compromise Tor. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2014), HotNets-XIII, ACM, pp. 14:1–14:7.

[49] WRIGHT, M., ADLER, M., LEVINE, B. N., AND SHIELDS, C. Defending anonymous communications against passive logging attacks. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2003), SP '03, IEEE Computer Society.

[50] ZEILEMAKER, N., AND POUWELSE, J. Open source column: Tribler: P2P search, share and stream. *SIGMultimedia Rec. 4*, 1 (Mar. 2012), 20–24.

[51] ZHANG, Z., ZHANG, Y., HU, Y. C., AND MAO, Z. M. Practical defenses against BGP prefix hijacking. In *Proceedings of the 2007 ACM CoNEXT Conference* (New York, NY, USA, 2007), CoNEXT '07, ACM.

[52] ZHANG, Z., ZHANG, Y., HU, Y. C., MAO, Z. M., AND BUSH, R. iSPY: Detecting IP prefix hijacking on my own. *IEEE/ACM Trans. Netw. 18*, 6 (Dec. 2010), 1815–1828.

[53] ZHENG, C., JI, L., PEI, D., WANG, J., AND FRANCIS, P. A light-weight distributed scheme for detecting IP prefix hijacks in real-time. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 2007), SIGCOMM '07, ACM, pp. 277–288.

[54] ZHU, Y., FU, X., GRAHAM, B., BETTATI, R., AND ZHAO, W. On flow correlation attacks and countermeasures in mix networks. In *Proceedings of the 4th International Conference on Privacy Enhancing Technologies* (Berlin, Heidelberg, 2005), PET'04, Springer-Verlag, pp. 207–225.

A Appendix: Rejected Countermeasures

Obfuscating packet timings and sizes: While the use of high latency mix networks [38, 23] and constant rate cover traffic [22] can mitigate timing analysis even against an adversary that observes all communications, these defenses are considered too costly to be deployed in the Tor network.

Mitigating asymmetric attacks: Recall that our asymmetric correlation attack leverages information in the TCP header, namely the sequence number field that indicates the number of acknowledged bytes. One potential countermeasure would be to encrypt the TCP header, by leveraging IP-layer encryption techniques such as IPSec. However, this approach introduces several challenges. First, it would require a substantial engineering effort to migrate Tor towards IPSEC. Second, since IPSEC is not widely used, this would make Tor traffic easy to distinguish from other encrypted traffic, thwarting its use for applications such as censorship resistance. Finally, encrypting the TCP header may not complete solve the attack. For example, an adversary could attempt to correlate TCP data packets with simply the number of TCP ACK packets, disregarding the sequence number field.

Circuit Fingerprinting Attacks:
Passive Deanonymization of Tor Hidden Services

Albert Kwon[†], Mashael AlSabah[‡§†*], David Lazar[†], Marc Dacier[‡], and Srinivas Devadas[†]

[†]*Massachusetts Institute of Technology*, {kwonal, lazard, devadas}@mit.edu
[‡]*Qatar Computing Research Institute*, mdacier@qf.org.qa
[§]*Qatar University*, malsabah@qu.edu.qa

This paper sheds light on crucial weaknesses in the design of hidden services that allow us to break the anonymity of hidden service clients and operators passively. In particular, we show that the *circuits*, paths established through the Tor network, used to communicate with hidden services exhibit a very different behavior compared to a general circuit. We propose two attacks, under two slightly different threat models, that could identify a hidden service client or operator using these weaknesses. We found that we can identify the users' involvement with hidden services with more than 98% true positive rate and less than 0.1% false positive rate with the first attack, and 99% true positive rate and 0.07% false positive rate with the second. We then revisit the threat model of previous website fingerprinting attacks, and show that previous results are directly applicable, with greater efficiency, in the realm of hidden services. Indeed, we show that we can correctly determine which of the 50 monitored pages the client is visiting with 88% true positive rate and false positive rate as low as 2.9%, and correctly deanonymize 50 monitored hidden service servers with true positive rate of 88% and false positive rate of 7.8% in an open world setting.

1 Introduction

In today's online world where gathering users' personal data has become a business trend, Tor [14] has emerged as an important privacy-enhancing technology allowing Internet users to maintain their anonymity online. Today, Tor is considered to be the most popular anonymous communication network, serving millions of clients using approximately 6000 volunteer-operated relays, which are run from all around the world [3].

In addition to sender anonymity, Tor's hidden services allow for receiver anonymity. This provides people with a free haven to host and serve content without the fear of being targeted, arrested or forced to shut down [11]. As a result, many sensitive services are only accessible through Tor. Prominent examples include human rights and whistleblowing organizations such as Wikileaks and Globalleaks, tools for anonymous messaging such as TorChat and Bitmessage, and black markets like Silkroad and Black Market Reloaded. Even many non-hidden services, like Facebook and DuckDuckGo, recently have started providing hidden versions of their websites to provide stronger anonymity guarantees.

That said, over the past few years, hidden services have witnessed various active attacks in the wild [12, 28], resulting in several takedowns [28]. To examine the security of the design of hidden services, a handful of attacks have been proposed against them. While they have shown their effectiveness, they all assume an active attacker model. The attacker sends crafted signals [6] to speed up discovery of *entry guards*, which are first-hop routers on circuits, or use congestion attacks to bias entry guard selection towards colluding entry guards [22]. Furthermore, all previous attacks require a malicious client to continuously attempt to connect to the hidden service.

In this paper, we present the first practical *passive* attack against hidden services and their users called *circuit fingerprinting* attack. Using our attack, an attacker can identify the presence of (client or server) hidden service activity in the network with high accuracy. This detection reduces the anonymity set of a user from millions of Tor users to just the users of hidden services. Once the activity is detected, we show that the attacker can perform website fingerprinting (WF) attacks to deanonymize the hidden service clients and servers. While the threat of WF attacks has been recently criticized by Juarez *et al.* [24], we revisit their findings and demonstrate that the world of hidden services is the ideal setting to wage WF attacks. Finally, since the attack is passive, it is undetectable until the nodes have been deanonymized, and can target thousands of hosts retroactively just by having access to clients' old network traffic.

[*] Joint first author.

Approach. We start by studying the behavior of Tor circuits on the live Tor network (for our own Tor clients and hidden services) when a client connects to a Tor hidden service. Our key insight is that during the circuit construction and communication phase between a client and a hidden service, Tor exhibits fingerprintable traffic patterns that allow an adversary to efficiently and accurately identify, and correlate circuits involved in the communication with hidden services. Therefore, instead of monitoring every circuit, which may be costly, the first step in the attacker's strategy is to identify suspicious circuits with high confidence to reduce the problem space to just hidden services. Next, the attacker applies the WF attack [10, 36, 35] to identify the clients' hidden service activity or deanonymize the hidden service server.

Contributions. This paper offers the following contributions:

1. We present key observations regarding the communication and interaction pattern in the hidden services design in Tor.

2. We identify distinguishing features that allow a passive adversary to easily detect the presence of hidden service clients or servers in the local network. We evaluate our detection approach and show that we can classify hidden service circuits (from the client- and the hidden service-side) with more than 98% accuracy.

3. For a stronger attacker who sees a majority of the clients' Tor circuits, we propose a novel circuit correlation attack that is able to quickly and efficiently detect the presence of hidden service activity using a sequence of only the first 20 cells with accuracy of 99%.

4. Based on our observations and results, we argue that the WF attacker model is *significantly more realistic and less costly* in the domain of hidden services as opposed to the general web. We evaluate WF attacks on the identified circuits (from client and hidden service side), and we are able to classify hidden services in both open and closed world settings.

5. We propose defenses that aim to reduce the detection rate of the presence of hidden service communication in the network.

Roadmap. We first provide the reader with a background on Tor, its hidden service design, and WF attacks in Section 2. We next present, in Section 3, our observations regarding different characteristics of hidden services. In Section 4, we discuss our model and assumptions, and in Sections 5 and 6, we present our attacks and evaluation. In Section 7, we demonstrate the effectiveness of WF attacks on hidden services. We then discuss possible future countermeasures in Section 8. Finally, we overview related works in Section 9, and conclude in Section 10.

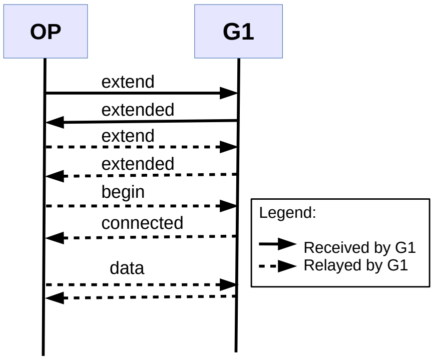


Figure 1: Cells exchanged between the client and the entry guard to build a general circuit for non-hidden streams after the circuit to G1 has been created.

2 Background

We will now provide the necessary background on Tor and its hidden services. Next, we provide an overview of WF attacks.

2.1 Tor and Hidden Services

Alice uses the Tor network simply by installing the Tor browser bundle, which includes a modified Firefox browser and the *Onion Proxy* (OP). The OP acts as an interface between Alice's applications and the Tor network. The OP learns about Tor's relays, *Onion Routers* (ORs), by downloading the network *consensus* document from *directory servers*. Before Alice can send her traffic through the network, the OP builds *circuits* interactively and incrementally using 3 ORs: an entry guard, middle, and exit node. Tor uses 512-byte fixed-sized *cells* as its communication data unit for exchanging control information between ORs and for relaying users' data.

The details of the circuit construction process in Tor proceeds as follows. The OP sends a `create_fast` cell to establish the circuit with the entry guard, which responds with a `created_fast`. Next, the OP sends an `extend` command cell to the entry guard, which causes it to send a `create` cell to the middle OR to establish the circuit on behalf of the user. Finally, the OP sends another `extend` to the middle OR to cause it to create the circuit at exit. Once done, the OP will receive an `extended` message from the middle OR, relayed by the entry guard. By the end of this operation, the OP will have shared keys used for layered encryption, with every hop on the circuit.[1] The exit node peels the last layer of the encryption and establishes the TCP connection to Alice's destination. Figure 1 shows the cells exchanged between OP and the entry guard for regular Tor connections, after the exchange of the `create_fast` and `created_fast` messages.

Tor uses TCP secured with TLS to maintain the OP-to-OR and the OR-to-OR connections, and multiplexes circuits within a single TCP connection. An OR-to-OR connection multiplexes circuits from various users, whereas an OP-to-OR connection multiplexes circuits from the same user. An observer watching the OP-to-OR TCP connection should not be able to tell apart which TCP segment belongs to which circuit (unless only one circuit is active). However, an entry guard is able to differentiate the traffic of different circuits (though the contents of the cells are encrypted).

Tor also allows receiver anonymity through hidden services. Bob can run a server behind his OP to serve content without revealing his identity or location. The overview of creation and usage of hidden services is depicted in Figure 2. In order to be reachable by clients, Bob's OP will generate a hidden service *descriptor*, and execute the following steps. First, Bob's OP chooses a random OR to serve as his *Introduction Point* (IP), and creates a circuit to it as described above. Bob then sends an `establish_intro` message that contains Bob's public key (the client can select more than one IP). If the OR accepts, it sends back an `intro_established` to Bob's OP. Bob now creates a signed descriptor (containing a timestamp, information about the IP, and its public key), and computes a descriptor-id based on the public key hash and validity duration. The descriptor is then published to the hash ring formed by the hidden service directories, which are the ORs that have been flagged by the network as "HSDir". Finally, Bob advertises his hidden service URL `z.onion` out of band, which is derived from the public key. This sequence of exchanged cells to create a hidden service is shown in Figure 3.

In Figure 4, we show how Alice can connect to Bob. Using the descriptor from the hidden service directories, The exchange of cells goes as follows. First, Alice's OP selects a random OR to serve as a *Rendezvous Point* (RP) for its connection to Bob's service, and sends an `establish_rendezvous` cell (through a Tor circuit). If the OR accepts, it responds with a `rendezvous_established` cell. In the meantime, Alice's OP builds another circuit to one of Bob's IPs, and sends an `introduce1` cell along with the address of RP and a cookie (one-time secret) encrypted under Bob's public key. The IP then relays that information to Bob and an `introduce2` cell, and sends an `introduce_ack` towards Alice. At this point, Bob's OP builds a circuit towards Alice's RP and sends it a `rendezvous1`, which causes the RP to send a `rendezvous2` towards Alice. By the end of this operation, Alice and Bob will have shared keys established through the cookie, and can exchange data through the 6 hops between them.

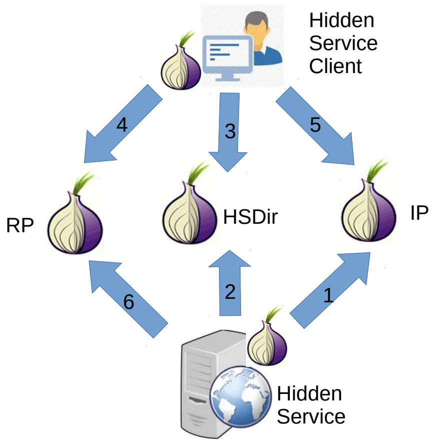


Figure 2: Circuit construction for Hidden Services.

2.2 Website Fingerprinting

One class of traffic analysis attacks that has gained research popularity over the past few years is the website fingerprinting (WF) attack [10, 36, 35, 9]. This attack demonstrates that a *local passive* adversary observing the (SSH, IPsec, or Tor) encrypted traffic is able, under certain conditions, to identify the website being visited by the user.

In the context of Tor, the strategy of the attacker is as follows. The attacker tries to emulate the network conditions of the monitored clients by deploying his own client who visits websites that he is interested in classifying through the live network. During this process, the attacker collects the network traces of the clients. Then, he trains a supervised classifier with many identifying features of a network traffic of a website, such as the sequences of packets, size of the packets, and inter-packet timings. Using the model built from the samples, the attacker then attempts to classify the network traces of users on the live network.

WF attacks come in two settings: open- or closed-world. In the closed-world setting, the attacker assumes that the websites visited are among a list of *k* known websites, and the goal of the attacker is to identify which one. The open-world setting is more realistic in that it assumes that the client will visit a larger set of websites

[1] We have omitted the details of the Diffie-Hellman handshakes (and the Tor Authentication Protocol (TAP) in general), as our goal is to demonstrate the data flow only during the circuit construction process.

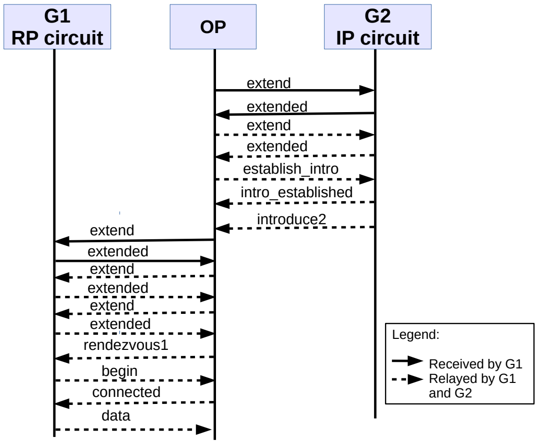


Figure 3: Cells exchanged in the circuit between the entry guards and the hidden service operator after the circuits to G1 and G2 have been created. Note that both G1 and G2 might be the same OR, and that entry guards can only view the first `extend` cell they receive.

n, and the goal of the attacker is to identify if the client is visiting a monitored website from a list of *k* websites, where $k \ll n$.

Hermann *et al.* [20] were the first to test this attack against Tor using a multinomial Naive Bayes classifier, which only achieved 3% success rate since it relied on packet sizes which are fixed in Tor. Panchenko *et al.* [33] improved the results by using a Support Vector Machine (SVM) classifier, using features that are mainly based on the volume, time, and direction of the traffic, and achieved more than 50% accuracy in a *closed-world* experiment of 775 URLs. Several subsequent papers have worked on WF in open-world settings, improved on the classification accuracy, and proposed defenses [10, 36, 35, 9].

3 Observations on Hidden Service Circuits

To better understand different circuit behaviors, we carried out a series of experiments, which were designed to show different properties of the circuits used in the communication between a client and a Hidden Service (HS), such as the Duration of Activity (DoA), incoming and outgoing cells, presence of multiplexing, and other potentially distinguishing features. DoA is the period of time during which a circuit sends or receives cells. The expected lifetime of a circuit is around 10 minutes, but circuits may be alive for more or less time depending on their activities.

For the remainder of this paper, we use the following terminology to denote circuits:

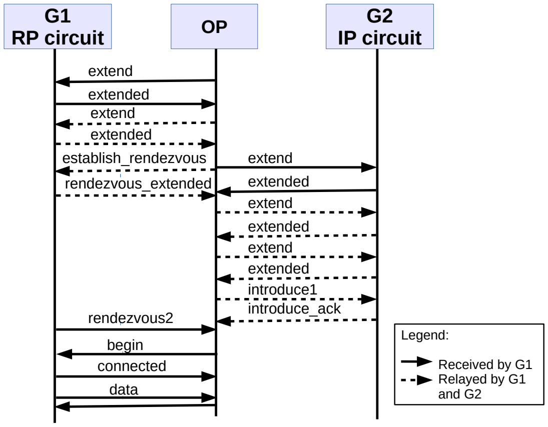


Figure 4: Cells exchanged in the circuit between the entry guards and the client attempting to access a hidden service after the circuits to G1 and G2 have been created.

- HS-IP: This is the circuit established between the Hidden Service (HS) and its Introduction Point (IP). The purpose of this circuit is to listen for incoming client connections. This circuit corresponds to arrow 1 in Figure 2.
- Client-RP: This is the circuit that a client builds to a randomly chosen Rendezvous Point (RP) to eventually receive a connection from the HS after he has expressed interest in establishing a communication through the creation of a Client-IP circuit. This circuit corresponds to arrow 4 in Figure 2.
- Client-IP: This is the circuit that a client interested in connecting to a HS builds to one of the IPs of the HS to inform the service of its interest in waiting for a connection on its RP circuit. This circuit corresponds to arrow 5 in Figure 2.
- HS-RP: This is the circuit that the HS builds to the RP OR chosen by the client to establish the communication with the interested client. Both this circuit and the Client-RP connect the HS and the client together over Tor. This circuit corresponds to arrow 6 in Figure 2.

For our hidden service experiments, we used more than 1000 hidden services that are compiled in `ahmia.fi` [2], an open source search engine for Tor hidden service websites. We base our observations on the logs we obtained after running all experiments for a three month period from January to March, 2015. This is important in order to realistically model steady-state Tor processes, since Tor's circuit building decisions are influenced by the circuit build time distributions. Furthermore, we configured our Tor clients so that they do not use fixed entry guards (by setting UseEntryGuards to 0). By doing so, we increase variety in our data collection, and do not limit ourselves to observations that are only obtained by using a handful of entry guards.

3.1 Multiplexing Experiment

To understand how stream multiplexing works for Client-RP and Client-IP circuits, we deployed a single Tor process on a local machine which is used by two applications: `firefox` and `wget`. Both automate hidden services browsing by picking a random `.onion` domain from our list of hidden services described above. While the `firefox` application paused between fetches to model user think times [19], the `wget` application accessed pages sequentially without pausing to model a more aggressive use. Note that the distribution of user think times we used has a median of 13 seconds, and a long tail that ranges between 152 to 3656 seconds for 10% of user think times. Since both applications are using the same Tor process, our intention is to understand how Tor multiplexes streams trying to access different `.onion` domains. We logged for every `.onion` incoming stream, the circuit on which it is attached. We next describe our observations.

Streams for different `.onion` domains are not multiplexed in the same circuit. When the Tor process receives a stream to connect to a `.onion` domain, it checks if it already has an existing RP circuit connected to it. If it does, it attaches the stream to the same circuit. If not, it will build a new RP circuit. We verified this by examining a 7-hour log from the experiment described above. We found that around 560 RP circuits were created, and each was used to connect to a different `.onion` domain.

Tor does not use IP or RP circuits for general streams. Tor assigns different purposes to circuits when they are established. For streams accessing non-hidden servers, they use general purpose circuits. These circuits can carry multiple logical connections; i.e., Tor multiplexes multiple non-hidden service streams into one circuit. On the other hand, streams accessing a `.onion` domain are assigned to circuits that have a rendezvous-related purpose, which differ from general circuits. We verified the behavior through our experiments, and also by reviewing Tor's specification and the source code.

3.2 Hidden Service Traffic Experiment

The goal of this experiment is to understand the usage of IP and RP circuits from the hidden server and from the client points of view. We deployed a hidden service on the live Tor network through which a client could visit a cached version of any hidden service from our list above, which we had previously crawled and downloaded. Our hidden service was simultaneously accessed by our five separate Tor instances, four of which use `wget`, while one uses `firefox`. Every client chooses a random page from our list of previously crawled hidden pages and requests it from our HS. Again, all clients pause between fetches for a duration that is drawn from a distribution of user think times. During the whole hour, we logged the usage of the IP and RP circuits observed from our hidden server, and we logged the RP and IP circuits from our 5 clients. We ran this experiment more than 20 times over two months before analyzing the results.

In addition, to get client-side traffic from live hidden services, we also deployed our five clients described above to access our list of real Tor HSs, rather than our deployed HS.

Similarly, to understand the usage of general circuits, and to compare their usage to IP, and RP circuits, we also ran clients as described above, with the exception that the clients accessed general (non-hidden) websites using Alexa's top 1000 URL [1]. From our experiments, we generated the cumulative distribution function (CDF) of the DoA, the number of outgoing and incoming cells, which are shown in Figure 5a, 5b, and 5c. We present our observations below.

IP circuits are unique. Figure 5a shows the CDF of the DoA for different circuit types. Interestingly, we observe that IP circuits from the hidden service side (i.e., HS-IP) are long lived compared to other circuit types. We observe that the DoA of IP circuits showed an age of around 3600 seconds (i.e., an hour), which happens to be the duration of each experiment. This seems quite logical as these have to be long living connections to ensure a continuous reachability of the HS through its IP. Another unique aspect of the hidden services' IP circuits, shown in Figure 5b, was that they had exactly 3 outgoing cells (coming from the HS): 2 `extend` cells and one `establish_intro` cell. The number of incoming cells (from the IP to the HS) differ however, depending on how many clients connect to them. Intuitively, one understands that any entry guard could, possibly, identify an OP acting on behalf of an HS by seeing that this OP establishes with him long-lived connections in which it only sends 3 cells at the very beginning. Furthermore, from the number of incoming client cells, an entry guard can also evaluate the popularity of that HS.

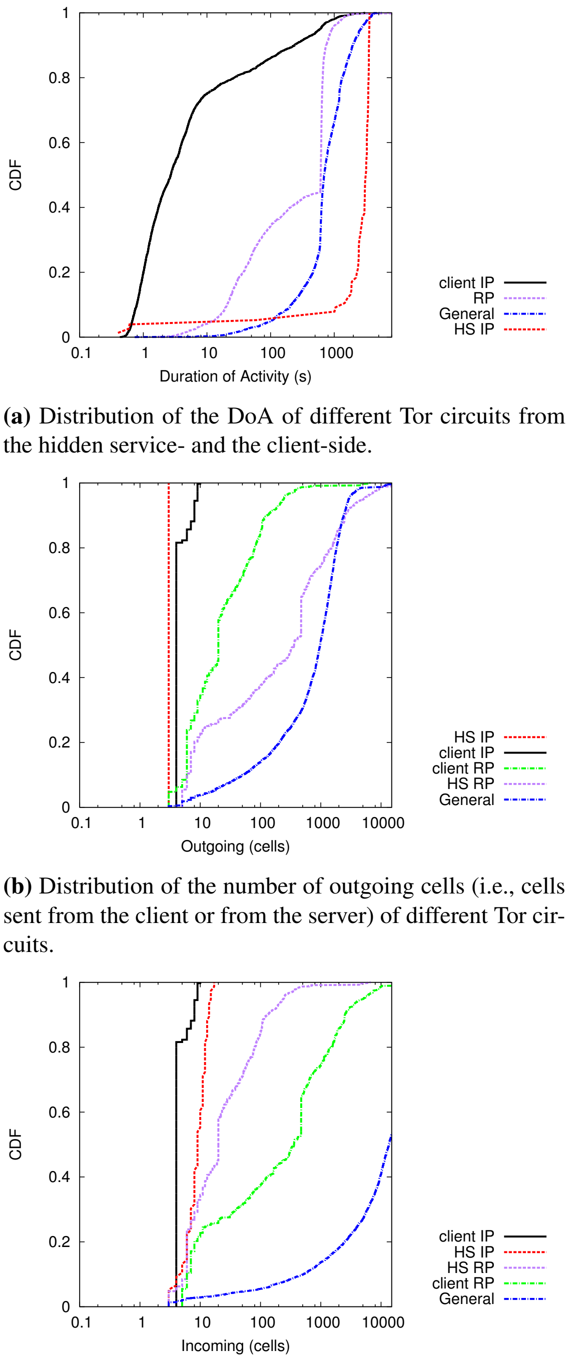
Client-IP circuits are also unique because they have the same number of incoming and outgoing cells. This is evidenced by the identical distributions of the number of incoming and outgoing cells shown in Figures 5b and 5c. For most cases, they had 4 outgoing and 4 incoming cells. The OP sends 3 `extend` and 1 `introduce1` cells, and receives 3 `extended` and 1 `introduce_ack` cells. Some conditions, such as RP failure, occasionally resulted in more exchanged cells, but IP circuits still had the same number of incoming and outgoing cells. Another unique feature was that, contrary to the HS-IP circuits, the Client-IP circuits are very short lived – their median DoA is around a second, as shown in Figure 5a, and around 80% of Client-IP circuits have a DoA that is less than or equal to 10 seconds. We expect this behavior as Client-IP circuits are not used at all once the connection to the service is established.

Active RP circuits, like general circuits, had a median DoA of 600 seconds, which is the expected lifetime of a Tor circuit. This was in particular observed with the clients which accessed our HS (the same RP circuit is reused to fetch different previously crawled pages). On the other hand, when the clients access live Tor hidden services, they have significantly lower DoA. Indeed, we observe (Figure 5a) that general circuits tend to have a larger DoA than RP circuits. The reason for this is that the same RP circuit is not used to access more than one hidden domain. Once the access is over, the circuit is not used again. On the other hand, general circuits can be used to access multiple general domains as long as they have not been used for more than 600 seconds.

HS-RP circuits have more outgoing cells than incoming cells. This is quite normal and expected since that circuit corresponds to the fetching of web pages on a server by a client. Typically, the client sends a few requests for each object to be retrieved in the page whereas the server sends the objects themselves which are normally much larger than the requests. There can be exceptions to this observation when, for instance, the client is uploading documents on the server or writing a blog, among other reasons.

Similarly, because RP circuits do not multiplex streams for different hidden domains, they are also expected to have a smaller number of outgoing and incoming cells throughout their DoA compared to active general circuits. As can be seen in Figures 5b, and 5c, one may distinguish between Client-RP and HS-RP circuits by observing the total number of incoming and outgoing cells. (Note that, as expected, the incoming distributions for the client and for the hidden service RP circuits from Figure 5c are the same as the outgoing distribution for hidden service and client RP, respectively, from Figure 5b.)

The incoming and outgoing distributions of RP circuits are based on fetching a hidden page, so the distributions we see in the figures might represent baseline distributions, and in the real network, they may have more incoming and outgoing cells based on users' activity. Although the exact distributions of the total number of incoming and outgoing cells for RP circuits is based on our models and may not reflect the models of users on the live network, we believe that the general trends are realistic. It is expected that clients mostly send small requests, while hidden services send larger pages.



(a) Distribution of the DoA of different Tor circuits from the hidden service- and the client-side.

(b) Distribution of the number of outgoing cells (i.e., cells sent from the client or from the server) of different Tor circuits.

(c) Distribution of the number of incoming cells (i.e., cells sent to the client or from the server) of different Tor circuits.

Figure 5: Cumulative distribution functions showing our observations from the experiments. Note that the X-axis scales exponentially.

Table 1: Edit distances of hidden pages across several weeks.

Edit distance	1 week	2 weeks	3 weeks	8 weeks
Q1	1	0.997	0.994	0.980
Median	1	1	1	1
Q3	1	1	1	1
Mean	0.96	0.97	0.96	0.927

Table 2: Edit distances of Alexa pages across several weeks.

Edit distance	1 week	2 weeks	3 weeks	8 weeks
Q1	0.864	0.846	0.81	0.71
Median	0.95	0.94	0.92	0.88
Q3	0.995	0.990	0.98	0.96
Mean	0.90	0.88	0.86	0.8

3.3 Variability in Hidden Pages

Over a period of four weeks, we downloaded the pages of more than 1000 hidden services once per week. We then computed the edit distance, which is the number of insertions, deletions, and substitutions of characters needed to transform the page retrieved at time T with the ones retrieved at time $T+k$ weeks (with $k \in [1..8]$). Table 1 shows the three quartiles and the mean for the distribution of edit distances computed, which demonstrates that the pages remained almost identical. For comparison, we also downloaded the pages of Alexa's top 1000 URLs, and computed the edit distances in Table 2. This is not surprising since the sources of variations in the pages are mostly due to dynamism, personalized advertisements, or different locations. None of these sources is applicable to hidden services since clients are anonymous when they initiate the connections. Note that hidden services may implement personalized pages for a user after he or she logs into his or her account; however in the context of this paper, we are mainly concerned with the retrieval of the very first page.

4 Threat Model

Alice's anonymity is maintained in Tor as long as no single entity can link her to her destination. If an attacker controls the entry and the exit of Alice's circuit, her anonymity can be compromised, as the attacker is able to perform traffic or timing analysis to link Alice's traffic to the destination [5, 23, 25, 32]. For hidden services, this implies that the attacker needs to control the two entry guards used for the communication between the client and the hidden service. This significantly limits the attacker, as the probability that both the client and the hidden service select a malicious entry guard is much lower than the probability that only one of them makes a bad choice.

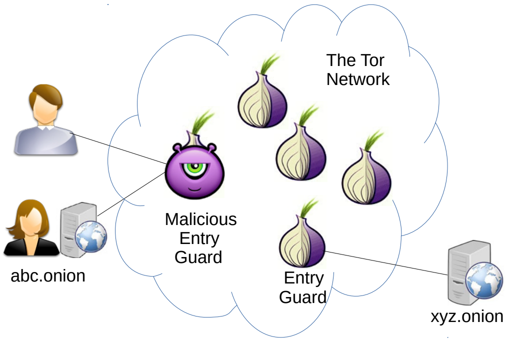


Figure 6: Our adversary can be a malicious entry guard that is able to watch all circuits

Our goal is to show that it is possible for a local passive adversary to deanonymize users with hidden service activities without the need to perform end-to-end traffic analysis. We assume that the attacker is able to monitor the traffic between the user and the Tor network. The attacker's goal is to identify that a user is either operating or connected to a hidden service. In addition, the attacker then aims to identify the hidden service associated with the user.

In order for our attack to work effectively, the attacker needs to be able to extract circuit-level details such as the lifetime, number of incoming and outgoing cells, sequences of packets, and timing information. We note that similar assumptions have been made in previous works [10, 35, 36]. We discuss the conditions under which our assumptions are true for the case of a network admin/ISP and an entry guard.

Network administrator or ISP. A network administrator (or ISP) may be interested in finding out who is accessing a specific hidden service, or if a hidden service is being run from the network. Under some conditions, such an attacker can extract circuit-level knowledge from the TCP traces by monitoring all the TCP connections between Alice and her entry guards. For example, if only a single active circuit is used in every TCP connection to the guards, the TCP segments will be easily mapped to the corresponding Tor cells. While it is hard to estimate how often this condition happens in the live network, as users have different usage models, we argue that the probability of observing this condition increases over time.

Malicious entry guard. Controlling entry guards allows the adversary to perform the attack more realistically and effectively. Entry guards are in a perfect position to perform our traffic analysis attacks since they have full visibility to Tor circuits. In today's Tor network, each OP chooses 3 entry guards and uses them for

45 days on average [16], after which it switches to other guards. For circuit establishment, those entry guards are chosen with equal probability. Every entry guard thus relays on average 33.3% of a user's traffic, and relays 50% of a user's traffic if one entry guard is down. Note that Tor is currently considering using a single fast entry guard for each user [13]. This will provide the attacker with even better circuit visibility which will exacerbate the effectiveness of our attack. This adversary is shown in Figure 6.

5 Circuit Fingerprinting Attack

In this section, we present our circuit fingerprinting attacks. Our attack allows an adversary to accurately and efficiently identify the presence of hidden service activity of a client or a server, and the circuit used to communicate with or by the hidden service (i.e., RP circuit). We first present an attack feasible for a more traditional attacker. Then, we describe a stronger attack for a more powerful adversary who can see more of the circuits from a user.

5.1 Classifying Special Circuits

Since the attacker is monitoring thousands of users, who produce hundreds of thousands of circuits, it is important to find an easy and straightforward approach to flag potentially "interesting" circuits for further examination. The attacker can exploit the simple and surprisingly distinctive features exhibited by IP and RP circuits (both client and hidden service side) to identify those circuits. In particular, we use the following features which are based on our observations in Section 3:

- **Incoming and outgoing cells**: This category of features will be useful in identifying IP circuits. For example, if a circuit sends precisely 3 cells, but has slightly more incoming cells (within a 1-hour duration), then this circuit is HS-IP with a high probability. Furthermore, if a circuit sends more than 3 cells, but has the exact same number of incoming and outgoing cells, then it is a client-IP with a high probability. This feature is also useful in distinguishing Client-RP from HS-RP circuits since we expect that HS-RP circuits to have more outgoing than incoming cells, and vice-versa for Client-RP circuits.

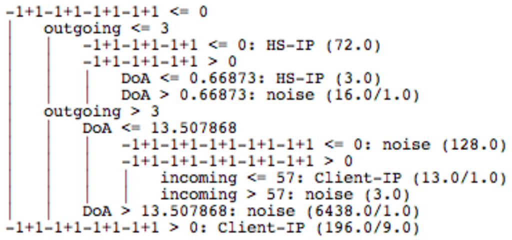
- **Duration of activity**: This feature is useful in distinguishing three groups of circuits: Client-IP circuits, HS-IP circuits, and all other circuits consisting of general, Client-, and HS-RP circuits. Recall that HS-IP circuits are long lived by design in order to be contacted by all interested clients, whereas client-IP circuits are inactive after performing the introduction process between the client and the hidden service, and have a median DoA of 1 second. Active general, Client-RP and HS-RP circuits can be alive and have a median of 600 seconds, which is the default lifetime of a circuit in Tor.

- **Circuit construction sequences**: We represent each of the first 10 cells (enough cells to capture the sequence of circuit establishment) either by the string -1 or +1. Each string encodes the direction of the corresponding cell. For example, the sequence "-1-1+1" corresponds to two outgoing cells followed by one incoming cell. This feature is useful in distinguishing Client-RP circuits from general and HS-RP circuits. The reason is that the circuit construction cell sequences in the case of Client-RP circuits differs from HS-RP and general circuits. This can be observed in Figures 1, 2, and 3. For example, we noticed that the sequence -1+1-1+1-1+1+1-1+1-1 is very common in Client-RP circuits, which corresponds to the sequence between the OP and G1 in Figure 3. However, HS-RP and general circuits have similar sequences so this feature alone cannot differentiate between those two circuit types.

Strategy. Different features are more indicative of certain circuit types. To best exploit those features, we perform our classification in two steps. First, the adversary looks for Client-IP and HS-IP circuits since those are the easiest ones to classify. This also allows the adversary to figure out if he is monitoring a HS or client of a HS. In the second step, the adversary examines the non-IP circuits to find RP circuits among them.

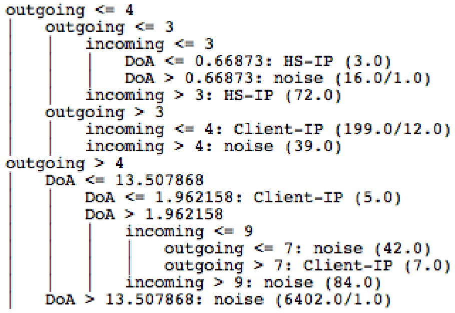
We use decision-tree classification algorithms, since identifying IP and RP circuits is dependent on an if-then-else conditional model as we discussed above. Tree-based algorithms build decision trees whose internal nodes correspond to the tests of features, and the branches correspond to the different outcomes of the tests. The leaves of the tree correspond to the classes, and the classification of a test instance corresponds to selecting the path in the tree whose branch values best reflect the new testing instance. Decisional trees have been used previously in the traffic classification literature [27, 4, 26] and are ideal for our problem.

Figures 7 and 8 depict decision trees which we use in the first step of this attack to identify the IP circuits. Note that general and RP circuits are treated as "noise". The tree in Figure 7 uses all features described above, and has a size of 15 nodes and 8 leaves, whereas the tree in Figure 8 omits the sequences, and only relies on incoming/outgoing packets and the DoA, which results in 10 leaves and a total size of 19 nodes. Both trees are very



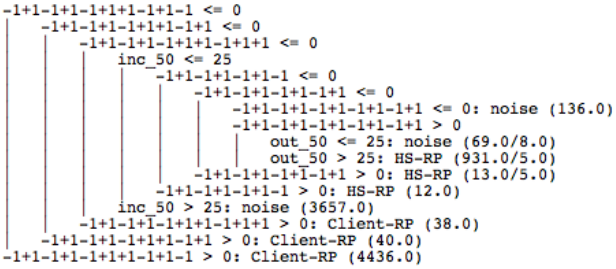
```
-1+1-1+1-1+1-1+1 <= 0
|   outgoing <= 3
|   |   -1+1-1+1-1+1 <= 0: HS-IP (72.0)
|   |   -1+1-1+1-1+1 > 0
|   |   |   DoA <= 0.66873: HS-IP (3.0)
|   |   |   DoA > 0.66873: noise (16.0/1.0)
|   outgoing > 3
|   |   DoA <= 13.507868
|   |   |   -1+1-1+1-1+1-1+1-1+1 <= 0: noise (128.0)
|   |   |   -1+1-1+1-1+1-1+1-1+1 > 0
|   |   |   |   incoming <= 57: Client-IP (13.0/1.0)
|   |   |   |   incoming > 57: noise (3.0)
|   |   DoA > 13.507868: noise (6438.0/1.0)
-1+1-1+1-1+1-1+1 > 0: Client-IP (196.0/9.0)
```

Figure 7: Decisional Tree (C4.5 algorithm) used in identifying IP circuits when cell sequences are used.



```
outgoing <= 4
|   outgoing <= 3
|   |   incoming <= 3
|   |   |   DoA <= 0.66873: HS-IP (3.0)
|   |   |   DoA > 0.66873: noise (16.0/1.0)
|   |   incoming > 3: HS-IP (72.0)
|   outgoing > 3
|   |   incoming <= 4: Client-IP (199.0/12.0)
|   |   incoming > 4: noise (39.0)
outgoing > 4
|   DoA <= 13.507868
|   |   DoA <= 1.962158: Client-IP (5.0)
|   |   DoA > 1.962158
|   |   |   incoming <= 9
|   |   |   |   outgoing <= 7: noise (42.0)
|   |   |   |   outgoing > 7: Client-IP (7.0)
|   |   |   incoming > 9: noise (84.0)
|   DoA > 13.507868: noise (6402.0/1.0)
```

Figure 8: Decisional Tree (C4.5 algorithm) used in identifying IP circuits when cell sequences are not used



```
-1+1-1+1-1+1+1-1+1-1 <= 0
|   -1+1-1+1-1+1+1-1+1 <= 0
|   |   -1+1-1+1-1+1+1-1+1+1 <= 0
|   |   |   inc_50 <= 25
|   |   |   |   -1+1-1+1-1+1-1 <= 0
|   |   |   |   |   -1+1-1+1-1+1-1+1 <= 0
|   |   |   |   |   |   -1+1-1+1-1+1-1+1-1+1 <= 0: noise (136.0)
|   |   |   |   |   |   -1+1-1+1-1+1-1+1-1+1 > 0
|   |   |   |   |   |   |   out_50 <= 25: noise (69.0/8.0)
|   |   |   |   |   |   |   out_50 > 25: HS-RP (931.0/5.0)
|   |   |   |   |   -1+1-1+1-1+1-1+1 > 0: HS-RP (13.0/5.0)
|   |   |   |   -1+1-1+1-1+1-1 > 0: HS-RP (12.0)
|   |   |   inc_50 > 25: noise (3657.0)
|   |   -1+1-1+1-1+1+1-1+1+1 > 0: Client-RP (38.0)
|   -1+1-1+1-1+1+1-1+1 > 0: Client-RP (40.0)
-1+1-1+1-1+1+1-1+1-1 > 0: Client-RP (4436.0)
```

Figure 9: Decisional Tree (C4.5 algorithm) used in identifying RP circuits out of web browsing circuits.

small, which allows for efficient classification. We discuss their performance in Section 6.1.

Once the adversary succeeds in identifying IP circuits, he is able to mark suspicious clients, and he can proceed to identifying their RP circuits. This can reduce his classification costs, and false positives. One challenge in distinguishing RP circuits from general circuits is that we cannot rely on DoA or the total number of incoming and outgoing cells as we did for IP circuits: in the case of general and RP circuits, those values are based on the user activity and can be biased by our models. To avoid such biases, we rely again on features that are protocol-dependent rather than user-dependent. Using our observation about sequences of Client-RP described previously, we can classify the circuit. Finally, to distinguish between HS-RP and general circuits, we use the first 50 cells of each circuit, and count the number of its incoming and outgoing cells. HS-RP circuits will generally have more outgoing than incoming, and the opposite should be true for general browsing circuits.

Figure 9 depicts a decision tree for classifying Client-RP, HS-RP and general circuits. It can be seen from the tree that Client-RP circuits are completely distinguished by their packet sequence fingerprint. Recall that those sequences represent the first 10 cells from the circuit, which is important as we want our sequences to be application independent. Also, HS-RP and general circuits are distinguished from each other by the fraction of incoming and outgoing cells of the first 50 cells. The tree contains a total of 17 nodes and only 9 leaves. We present the performance of this tree in Section 6.1.

5.2 Correlating Two Circuits

As mentioned in Section 4, Tor is considering using only a single entry guard per user. This changes the adversarial model: a malicious entry guard can now see *all* of the circuits used by a connected user. In this scenario, the attacker can see both IP and RP circuits. Even for a traditional entry guard, it has at least 11-25% chance of seeing both circuits. Such an attacker can leverage the fact that *the process of establishing the connection with the hidden service is fingerprintable*.

A client accessing a hidden service will exhibit a different circuit construction and data flow pattern from that of a client accessing a non-hidden service. For a client accessing a hidden service, the OP first builds the RP circuit, and simultaneously starts building a circuit to the IP. In contrast, a client visiting a regular website only establishes one circuit. (Figures 1 and 4 in Section 2 illustrate the exact flow of cells.) Using this fact, the attacker can classify behavior of *pairwise circuits*, and learn the RP of a user. In particular, we show that the first 20 cells of a circuit pair, which include all the cells used to establish connections with IP and RP, are enough to identify IP-RP pairs.

6 Evaluation

To evaluate our features with different machine learning algorithms, we used Weka [18], a free and open-source suite which provides out-of-the-box implementation of various machine learning algorithms. We experimented with the following algorithms: CART [8], which builds binary regression trees based on the Gini impurity, C4.5 [34], which uses information gain to rank possible outcomes, and *k*-nearest neighbors (*k*-NN for short), which considers the neighbors that lie close to each other

Table 3: Number of instances of different circuit types

Dataset	HS-IP	HS-RP	Client-IP	Client-RP	general
IP-Noise	76	954	200	4514	3862
RP-Noise	N/A	954	N/A	4514	3862

in the feature space. When the k-NN classifier is used, we set $k = 2$ with a weight that is inversely proportional to the distance. For each algorithm, we study the true positive rate ($TPR = \frac{TP}{TP+FN}$) and the false positive rate ($FPR = \frac{FP}{TN+FP}$). Specifically, TPR is the rate of correctly identified sensitive class, and FPR is the rate of incorrectly identified non-sensitive class. We collected network traces over the live Tor networks for our clients and server, and we did not touch or log the traffic of other users. We used the latest stable release of the Tor source code (tor-0.2.5.10) in all our experiments.

6.1 Accuracy of Circuit Classification

Datasets. From our long-term experiments described in Section 3, we extracted 5 types of circuits: Client-IP, Client-RP, HS-IP, HS-RP and general circuits. From every circuit, we created an instance consisting of its sequences (first 10 cell), DoA, total incoming and outgoing number of cells within the first 50 cells in the circuit, and a class label corresponding to the circuit type. Furthermore, since Tor is mainly used for web browsing [29], we designed our datasets so that most of the instances are "general" circuits to reflect realistically what an adversary would face when attempting to classify circuits of real users on the live network.

Recall that our general circuits are generated by browsing a random page from the top 1000 websites published by Alexa [1]. This list contains very small webpages (such as localized versions of google.com), and large websites (such as cnn.com). While it is not clear if this set of websites represents what real Tor users visit, we believe that this would not affect our approach since our features are protocol-dependent rather than website- or user-dependent. This is also true about our RP circuits. Therefore, we believe that the specific models and websites should not have an impact on our classification approach. Table 3 shows the number of instances of every class for both datasets.

Since we perform the classification in two steps, we created the following datasets:

- IP-Noise dataset: This dataset consists of 76 HS-IP circuits, 200 Client-IP circuits, and 6593 "noise" circuits. The 200 Client-IP circuits were selected uniformly at random from a large collection of 4514 Client-IP circuits.[2] The circuits labeled with the class "noise" consist of 954 HS-RP, 4514 Client-RP, and 3862 general browsing circuits.

- RP-Noise dataset: This dataset contains 200 Client-RP, 200 HS-RP circuit, and 3862 "noise" circuits (general browsing). The Client-RP and HS-RP circuits were selected uniformly at random from our collection of 954 and 4514 HS-RP and Client-RP circuits, respectively. Again, our goal is to imitate the conditions that the adversary would most likely face on the live network, where the majority of circuits to be classified are general browsing circuits.

Results. We used n-fold cross-validation for the three classification algorithms. This is a validation technique where the dataset is divided into n subsets and $n-1$ subsets are used for training and 1 subset is used for testing, and the process is repeated n times, where each subset is used for validation exactly once. Finally, the results from all n folds are averaged. We set n to 10 for our experiments.

We found that both C4.5 and CART perform equally well in classifying both datasets. We also found that k-NN performs well when cell sequences are used as features but otherwise performs poorly. For the IP-Noise dataset, when cell sequences are not used as a feature, as shown in Figure 10, the per-class TPR for CART ranges between 91.5% (Client-IP class) and 99% (for noise class), whereas the per-class accuracy for C4.5 ranges between 95.5% (Client-IP), and 99.8% (for noise class). k-NN performs worse with a TPR ranging from 55% (for HS-IP class) and 99% (for noise class). k-NN also has a high FPR for the noise class that exceeds 20%. Both C4.5 and Cart have 0% FPR for HS-IP, and have 0.2% and 0.1% FPR for Client-IP, respectively. However, we found that Cart has 7% FPR for the noise class because 17 out of 200 Client-IP instances got misclassified as noise. Therefore, based on the TPR and FPR rates, we conclude that C4.5 outperforms k-NN and Cart for the IP-Noise dataset when no sequences are used as features.

Figure 11 shows that when sequences are used as classification features, all three classifiers perform very well, but C4.5 still outperforms both Cart and k-NN with a nearly perfect per-class TPR. Interestingly, all classifiers provide 0% FPR for HS-IP and very low FPR for Client-IP and noise. We note that C4.5 also provides the best performance since it provides the highest TPR and lowest FPR among other classifiers.

[2] Recall that Client-IP are short-lived and one of these circuits is created every time a client attempts to connect to a HS, whereas HS-IP circuit samples are the most difficult to obtain since we observe each of them for an hour before we repeat experiments.

Table 4: Impact of different features on the TPR and FPR for the RP-Noise dataset. The table shows the accuracy results (TPR / FPR) if individual categories of features are used.

TPR/FPR	Sequences	Cells	Sequences and Cells
HS-RP	0% / 0%	95% / 0.1%	95.5% / 0.1%
Client-RP	98% / 0%	15% / 0.3%	99% / 0%
Noise	100% / 46%	99.5% / 44.8%	99.9% / 2.5%

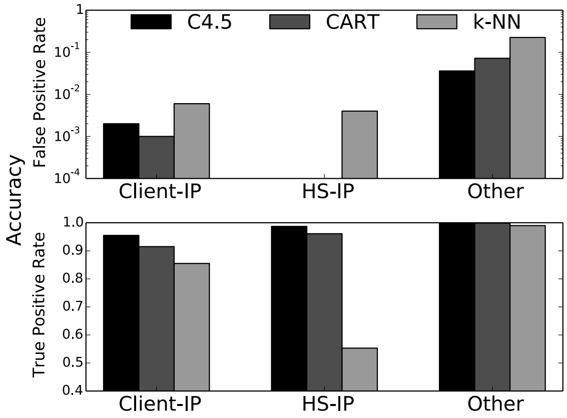


Figure 10: TPR and FPR of circuit classification with 3 classes when no cell sequences are used. FPR is shown in log scale.

For the RP-Noise dataset, we observe that both C4.5 and Cart provide identical performances in terms of TPR and FPR as shown in Figure 12. Both provide very high TPR for all classes, and 0% FPR for HS-RP and Client-RP classes. The FPR achieved by C4.5 and CART for the noise class is also low at 3%. *k*-NN provides slightly higher TPR for the HS-RP class than CART and C4.5, but the TPR is very similar to that achieved by CART and C4.5. We thus conclude that all classifiers perform equally well for the RP-noise dataset. Table 4 shows the impact on the overall accuracy based on different features.

6.2 Accuracy of Circuit Correlation

Datasets. We collected data of both the clients' and the servers' IP and RP circuit pairs for different hidden services. To collect client side circuit pairs, we used both `firefox` and `wget` (with flags set to mimic a browser as much as possible) to connect to hidden and non-hidden services from many different machines, each with one Tor instance. Each client visited 1000 different hidden services and 1000 most popular websites [1] several times. To collect server side circuit pairs, we spawned our own hidden service, and had multiple clients connect and view a page. The number of simultaneously connecting clients ranged from 1 to 10 randomly.

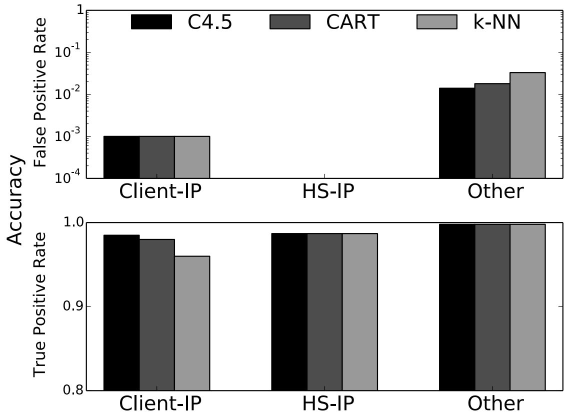


Figure 11: TPR and FPR of circuit classification with 3 classes when cell sequences are used. FPR is shown in log scale.

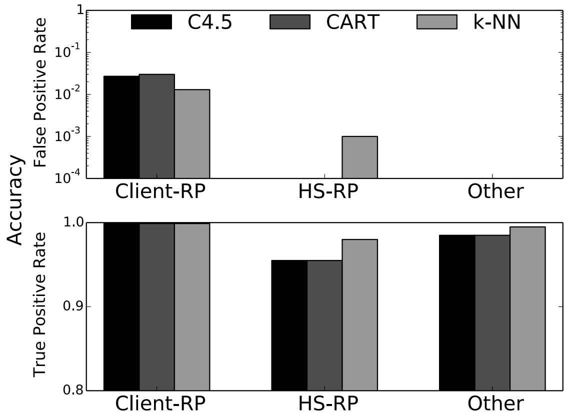


Figure 12: TPR and FPR of circuit classification with 3 classes. FPR is shown in log scale.

We then extracted traces of the first 20 cells of 6000 IP-RP circuit pairs (3000 client and 3000 server pairs) and 80000 of non-special circuit pairs. The non-special circuit pairs included any combination that is not IP-RP (i.e., IP-general, RP-general, general-general).

Result. The accuracy of IP-RP classification is shown in Table 5. We again used 10-fold cross validation to evaluate our attack. We can see that IP-RP circuit pairs are very identifiable: all three algorithms have 99.9% true positive rate, and less than 0.05% false positive rate. This accuracy is likely due to the uniqueness of the exact sequence of cells for IP-RP circuits. From the 6000 sequences of IP-RP pairs and 80000 non-special pairs, there were 923 and 31000 unique sequences respectively. We found that only 14 sequences were shared between the two classes. Furthermore, of those 14 sequences, only 3 of them had more than 50 instances.

This result implies that an adversary who can see a user's IP and RP (e.g., entry guard) can classify IP and RP circuits with almost 100% certainty by observing a

Table 5: IP/RP Pair Classification

Algorithm	True Positive Rate	False Positive Rate
CART	0.999	$2.07 \cdot 10^{-4}$
C4.5	0.999	$3.45 \cdot 10^{-4}$
k-NN	0.999	$6.90 \cdot 10^{-5}$

few cells. Moreover, the attack can be carried out in near real-time speed since we only need the first 20 cells. The attacker can thus effectively rule out most non-sensitive circuits, making data collection much easier.

7 Website Fingerprinting Revisited

In this section, we discuss the impact of our observations and attacks on WF, and show the result of applying modern WF techniques to hidden services. We show that the adversary can classify both the clients' and the operators' hidden service activities with high probability.

7.1 Adversaries Targeting Hidden Services

Juarez *et al.* [24] recently criticized various WF attacks because they made assumptions which were too advantageous for the adversary, and exacerbated the effectiveness of their attacks. In this section, we discuss some of the points that were raised by Juarez *et al.* [24] and show how our attacks address the concerns in the case of attacking hidden services.

Noisy streams. Previous WF attacks assumed that the adversary is able to eliminate noisy background traffic [10, 35, 36]. For example, if the victim's file download stream (noise) is multiplexed in the same circuit with the browsing stream (target), the attacker is able to eliminate the noisy download stream from the traces. With a lack of experimental evidence, such an assumption might indeed overestimate the power of the attack.

In the world of hidden services, we observed that Tor uses separate circuits for different `.onion` domains (Section 3). Furthermore, Tor does not multiplex general streams accessing general non-hidden services with streams accessing hidden services in the same circuit. From the attacker's perspective, this is a huge advantage since it simplifies traffic analysis; the attacker does not have to worry about noisy streams in the background of target streams. Furthermore, the previous assumption that the attacker can distinguish different pages loads is still valid [35]. User "think times" still likely dominate the browsing session, and create noticeable time gaps between cells.

Size of the world. All previous WF attacks have a problem space that is potentially significantly smaller than a realistic setting. Even in Wang *et al.*'s "large" open-world setting, the number of all websites are limited to 10,000 [35]. Moreover, different combinations of websites sharing one circuit could make it impossible to bound the number of untrainable streams. This implies that the false positive rate of WF techniques in practice is significantly higher, since the ratio of trained non-monitored pages to all non-monitored pages go down.

However, in the case of hidden services, the size of the world is significantly smaller than that of the world wide web. Also, while it is true that not all existing hidden services are publicly available, it has been shown that enumerating hidden services is possible [6][3]. In some cases, the attacker could be mainly interested in identifying a censored list of services that make their onion address public. Furthermore, we do not need to consider the blow up of the number of untrainable streams. Since RP always produces clean data, the number of untrained streams is bounded by the number of available hidden services.

Rapidly changing pages. The contents of the general web changes very rapidly as shown by Juarez *et al.* [24]. However, hidden pages show minimal changes over time (Section 3), contrary to non-hidden pages. The slowly changing nature of hidden services reduces the attacker's false positives and false negatives, and minimizes the cost of training. Furthermore, hidden services do not serve localized versions of their pages.

Replicability. Another assumption pointed out by Juarez *et al.* [24], which we share and retain from previous WF attacks, is the replicability of the results. That is, we are assuming that we are able to train our classifier under the same conditions as the victim. Indeed, we acknowledge that since it is difficult to get network traces of users from the live Tor network, we are faced with the challenge of having to design experiments that realistically model the behavior of users, hidden services, and the conditions of the network. That said, our attacks described above use features that are based on circuit interactions and are independent of the users' browsing habits or locations, which can reduce the false positive rate for the WF attacker.

Based on the above discussion, we claim that our attacker model is significantly more realistic than that of previous WF attacks [10, 35, 36]. While the conclusions made by Juarez *et al.* [24] regarding the assumptions of previous WF attacks are indeed insightful, we argue that many of these conclusions do not apply to the realm of hidden services.

[3] As pointed out by a reviewer, it is worth noting that the specific technique used in [6] has since been adressed by a change in the HS directory protocol.

7.2 Methodology

We first note here that hidden services have significantly lower uptime than a normal website on average. We found that only about 1000 hidden services were consistently up of the 2000 hidden services we tried to connect to. This makes collecting significant amounts of traces of hidden services very difficult. Furthermore, we found that hundreds of the available services were just a front page showing that it had been compromised by the FBI. This introduces significant noise to WF printing techniques: we now have hundreds of "different" pages that look exactly the same. We thus tried to group all of these hidden services as one website. This unfortunately limited our open world experiments to just 1000 websites. We also note that there may be other similar cases in our data, where a hidden service is not actually servicing any real content.

7.2.1 Data Collection

We gathered data to test OP servicing a normal user and a hidden service for both closed and open world settings. For the normal user case, we spawned an OP behind which a client connects to any website. We then used both `firefox` and `wget` to visit 50 sensitive hidden services that the attacker monitors (similar to experiments in Section 3). Our sensitive hidden service list contained a variety of websites for whistleblowing, adult content, anonymous messaging, and black markets. We collected 50 instances of the 50 pages, and 1 instance of 950 the other hidden services.

For the hidden service case, we first downloaded the contents of 1000 hidden services using a recursive `wget`. We then started our own hidden service which contains all the downloaded hidden service contents in a subdirectory. Finally, we created 5 clients who connect to our service to simulate users connecting to one server, and visiting a cached page. We then reset all the circuits, and visited a different cached page to simulate a different hidden service. We repeated this experiment 50 times for the 50 monitored hidden services, and once for the other 950 hidden services.

We argue that this setup generates realistic data for the following reasons. First, as shown in Section 3, the actual contents of hidden services changes minimally. Thus servicing older content from a different hidden service within our hidden service should not result in a significantly different trace than the real one. Second, the exact number of clients connected to the service is irrelevant once you consider the results in Section 6. An RP circuit correlates to one client, and thus allows us to consider one client trace at a time. Note that this is how a real-life adversary could generate training data to deanonymize the servers: it could run its own servers of the cached hidden services, and collect large samples of the servers' traffic patterns.

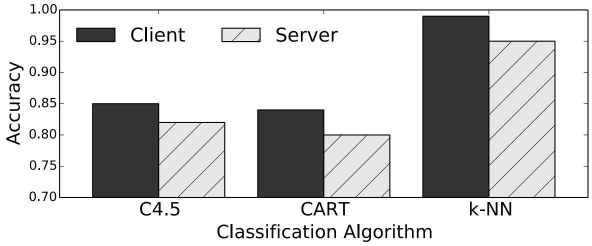


Figure 13: Accuracy of website fingerprinting attacks in closed world setting.

7.2.2 Website Fingerprinting Hidden Services

We extracted features similar to the ones presented in Wang *et al.* [35] from the data we collected.

- **General Features:** We use the total transmission size and time, and the number of incoming and outgoing packets.
- **Packer Ordering:** We record the location of each outgoing cell.
- **Bursts:** We use the number of consecutive cells of the same type. That is, we record both the incoming bursts and outgoing bursts, and use them as features.

We performed WF in closed and open world settings. In the closed world setting, the user visits/hosts a hidden service selected randomly from the list of 50 pages known to the attacker. In the open world setting, the user visits/hosts any of the 1000 pages, only 50 of which are monitored by the attacker. In either case, the attacker collects network traces of the Tor user, and tries to identify which service is associated with which network trace. We can consider the clients and the servers separately since we can identify HS-IP and Client-IP using the attack from Section 5.1 with high probability.

7.3 WF Accuracy on Hidden Services

We ran the same classifiers as the ones used in Section 6: CART, C4.5, and *k*-NN.[4] The accuracy of the classifiers in the closed world setting of both client and server is shown in Figure 13. For the open world setting, we varied the number of non-monitored training pages from 100 to 900 in 100 page increments (i.e., included exactly

[4] For *k*-NN, we tested with both Wang *et al.* [35] and the implementation in Weka, and we got inconsistent results. For consistency in our evaluation, we used the Weka version as with the other two classifiers.

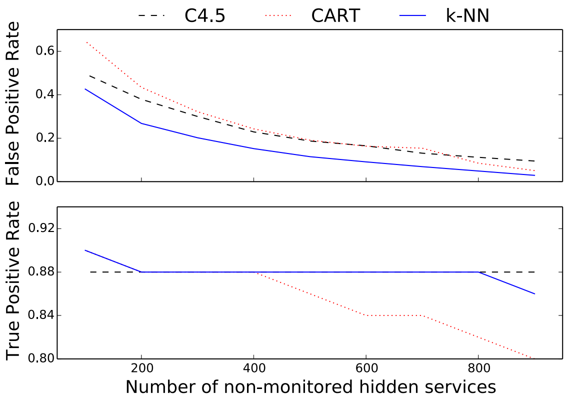


Figure 14: TPR and FPR of the client side classification for different classifiers.

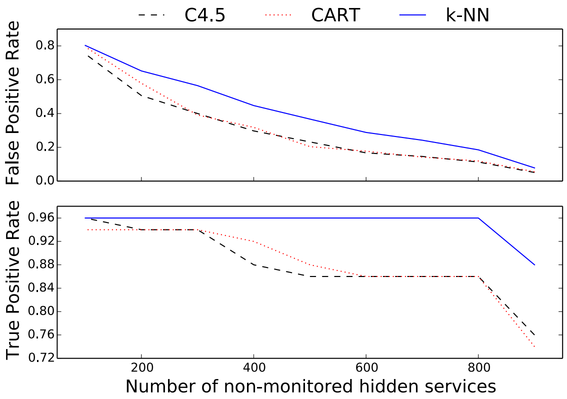


Figure 15: TPR and FPR of the server side classification for different classifiers.

one instance of the websites in the training set), and measured the TPR and FPR. The results of the open world experiment on the clients and the servers are shown in Figure 14 and Figure 15 respectively.

In all settings, we found that *k*-NN classifier works the best for classifying hidden services. We believe this is because *k*-NN considers multiple features simultaneously while the tree-based classifiers consider each feature one after another. In the closed world, the accuracy of *k*-NN was 97% for classifying the clients and 94.7% for the servers. In the open world, the *k*-NN classifier again performed the best. The TPR of *k*-NN reduced slightly as we increased the number of trained non-monitored websites. The TPR ranged from 90% to 88% and 96% to 88% for classifying clients and servers respectively. The FPR steadily decreased as we trained on more non-monitored websites: for classifying clients, it varied from 40% to 2.9% depending on the number of trained pages. Similarly, FPR of classifying servers varied from 80.3% to 7.8% for attacking servers.[5] Though the FPR is too a large for accurate classification when trained only on small number of non-monitored websites, we found that it quickly decreased as we increased the number of websites in the training set.

In general, the classifiers performed better in identifying clients' connections better than the hidden services servers; the TPR was comparable, but the FPR was significantly lower for classifying clients. We believe that this is, at least partially, due to the fact that we are using one real hidden service to emulate multiple hidden services; our data does not capture the differences in the differences in hardware, software, locations, and other characteristics real hidden service servers would have.

8 Future Possible Defenses

Our attacks rely on the special properties of the circuits used for hidden service activities. For the first attack (Section 5.1), we used three very identifiable features of the circuits: (1) DoA, (2) number of outgoing cells, and (3) number of incoming cells. To defend against this attack, Tor should address the three features. First, *all* circuits should have similar lifetime. Client IP and hidden service IP lasts either a very short or very long time, and this is very identifying. We recommend that circuits with less than 400 seconds of activity should be padded to have a lifetime of 400-800 seconds. Furthermore, we suggest that hidden services re-establish their connection to their IPs every 400-800 seconds to avoid any circuits from lasting too long. Second, hidden service and client IP should have a larger and varying number of outgoing and incoming cells. IPs are only used to establish the connection which limits the possible number of exchanged cells. We believe they should send and receive a random number of `PADDING` cells, such that their median value of incoming and outgoing cells is similar to that of a general circuit. We evaluated the effectiveness of this defense on the same dataset used in Section 6.1, and found that the true positive rate for the IPs and RPs fell below 15%. Once the features look the same, the classifiers cannot do much better than simply guessing.

To prevent the second attack (Section 5.2), we recommend that every circuit be established in a pair with the same sequence for the first few cells. If an `extend` fails for either circuit (which should be a rare occurrence), then we should restart the whole process to ensure no information is leaked. To do this efficiently, Tor could use its preemptive circuits. Tor already has the practice of building circuits preemptively for performance reasons. We can leverage this, and build the preemptive circuit

[5] The results are not directly comparable to previous WF attacks due to the differences in the settings, such as the size of the open world.

with another general circuit with the same sequence as the IP-RP pairs. This would eliminate the second attack.

For WF attacks (Section 7), defenses proposed by previous works [35, 9] will be effective here as well. Furthermore, for the clients, the results of Juarez *et al.* [24] suggest that WF attacks on hidden service would have significantly lower accuracy if an RP circuit is shared across multiple hidden service accesses.

9 Related Work

Several attacks challenging the security of Tor have been proposed. Most of the proposed attacks are based on side-channel leaks such as congestion [31, 17], throughput [30], and latency [21]. Other attacks exploit Tor's bandwidth-weighted router selection algorithm [5] or its router reliability and availability [7]. Most of these attacks are active in that they require the adversary to perform periodic measurements, induce congestion, influence routing, or kill circuits.

Our attacks on the other hand, like the various WF attacks, are passive. Other passive attacks against Tor include Autonomous Systems (AS) observers [15], where the attacker is an AS that appears anywhere between the client and his entry guard, and between the exit and the destination.

In addition, several attacks have been proposed to deanonymize hidden services. Øverlier and Syverson [32] presented attacks aiming to deanonymize hidden services as follows: the adversary starts by deploying a router in the network, and uses a client which repeatedly attempts to connect to the target hidden service. The goal is that, over time, the hidden service will choose the malicious router as part of its circuit and even as its entry guard to the client allowing the attacker to deanonymize him using traffic confirmation.

A similar traffic confirmation attack was described by Biryukov *et al.* [6]. The malicious RP sends a message towards the hidden service consisting of 50 padding cells when it receives the `rendezvous1` sent by the hidden service. This signal allows another malicious OR along the circuit from the hidden service to the RP, to identify the hidden service or its entry guard on the circuit. Biryukov *et al.* also show how it is possible for the attacker to enumerate all hidden services and to deny service to a particular target hidden service.

10 Conclusion

Tor's hidden services allow users to provide content and run servers, while maintaining their anonymity. In this paper, we present the first passive attacks on hidden services, which allow an entry guard to detect the presence of hidden service activity from the client- or the server-side. The weaker attacker, who does not have perfect circuit visibility, can exploit the distinctive features of the IP and RP circuit communication and lifetime patterns to classify the monitored circuits to five different classes.

For the stronger attacker, who has perfect circuit visibility (in the case where the client uses only one entry guard), the attacker runs a novel pairwise circuit correlation attack to identify distinctive cell sequences that can accurately indicate IP and RP circuits.

We evaluated our attacks using network traces obtained by running our own clients and hidden service on the live Tor network. We showed that our attacks can be carried out easily and yield very high TPR and very low FPR. As an application of our attack, we studied the applicability of WF attacks on hidden services, and we made several observations as to why WF is more realistic and serious in the domain of hidden services. We applied state-of-the-art WF attacks, and showed their effectiveness in compromising the anonymity of users accessing hidden services, and in deanonymizing hidden services. Finally, we propose defenses that would mitigate our traffic analysis attacks.

11 Code and Data Availability

Our data and scripts are available at `http://people.csail.mit.edu/kwonal/hswf.tar.gz`.

12 Acknowledgements

The authors thank Tao Wang and the reviewers for their useful feedback and comments. This reserach was supported in part by the QCRI-CSAIL partnership.

References

[1] Alexa The Web Information Company. `https://www.alexa.com`.

[2] Tor Hidden Service Search. `https://ahmia.fi`.

[3] Tor. Tor Metrics Portal. `https://metrics.torproject.org/`.

[4] ALSABAH, M., BAUER, K., AND GOLDBERG, I. Enhancing tor's performance using real-time traffic classification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 73–84.

[5] BAUER, K., MCCOY, D., GRUNWALD, D., KOHNO, T., AND SICKER, D. Low-Resource Routing Attacks Against Tor. In *Proceedings of the Workshop on Privacy in the Electronic Society (WPES 2007)* (October 2007), pp. 11–20.

[6] BIRYUKOV, A., PUSTOGAROV, I., AND WEINMANN, R.-P. Trawling for Tor Hidden Services: Detection, Measurement, Deanonymization. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2013), SP '13, IEEE Computer Society, pp. 80–94.

[7] Borisov, N., Danezis, G., Mittal, P., and Tabriz, P. Denial of Service or Denial of Security? How Attacks on Reliability can Compromise Anonymity. In *Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS '07* (October 2007), pp. 92–102.

[8] Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. *Classification and Regression Trees*. CRC Press, New York, 1999.

[9] Cai, X., Nithyanand, R., Wang, T., Johnson, R., and Goldberg, I. A Systematic Approach to Developing and Evaluating Website Fingerprinting Defenses. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2014), CCS '14, ACM, pp. 227–238.

[10] Cai, X., Zhang, X., Joshi, B., and Johnson, R. Touching from a Distance: Website Fingerprinting Attacks and Defenses. In *Proceedings of the 19th ACM conference on Computer and Communications Security (CCS 2012)* (October 2012).

[11] Dingledine, R. Using Tor Hidden Services for Good. `https://blog.torproject.org/blog/using-tor-good`, 2012. Accessed February 2015.

[12] Dingledine, R. Tor security advisory: "relay early" traffic confirmation attack. https://blog.torproject.org/blog/tor-security-advisory-relay-early-traffic-confirmation-attack, 2014. Accessed February 2015.

[13] Dingledine, R., Hopper, N., Kadianakis, G., and Mathewson, N. One Fast Guard for Life (or 9 months). `https://www.petsymposium.org/2014/papers/Dingledine.pdf`, 2015. Accessed February 2015.

[14] Dingledine, R., Mathewson, N., and Syverson, P. Tor: The Second-Generation Onion Router. In *Proceedings of the 13th USENIX Security Symposium* (August 2004), pp. 303–320.

[15] Edman, M., and Syverson, P. As-awareness in Tor Path Selection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (2009), CCS '09, pp. 380–389.

[16] Elahi, T., Bauer, K., AlSabah, M., Dingledine, R., and Goldberg, I. Changing of the Guards: A Framework for Understanding and Improving Entry Guard Selection in Tor. In *Proceedings of the 2012 ACM Workshop on Privacy in the Electronic Society* (2012), WPES '12, pp. 43–54.

[17] Evans, N., Dingledine, R., and Grothoff, C. A Practical Congestion Attack on Tor Using Long Paths. In *Proceedings of the 18th USENIX Security Symposium* (Berkeley, CA, USA, 2009), USENIX Association, pp. 33–50.

[18] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. The WEKA Data Mining Software: An Update. *SIGKDD Explor. Newsl. 11*, 1 (November 2009), 10–18.

[19] Hernández-Campos, F., Jeffay, K., and Smith, F. D. Tracking the Evolution of Web Traffic: 1995-2003. In *11th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2003), 12-15 October 2003, Orlando, FL, USA* (2003), pp. 16–25.

[20] Herrmann, D., Wendolsky, R., and Federrath, H. Website Fingerprinting: Attacking Popular Privacy Enhancing Technologies with the Multinomial Naive Bayes Classifier. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security* (2009), CCSW '09, pp. 31–42.

[21] Hopper, N., Vasserman, E. Y., and Chan-Tin, E. How Much Anonymity does Network Latency Leak? In *Proceedings of the 14th ACM conference on Computer and Communications Security* (October 2007), CCS '07.

[22] Jansen, R., Tschorsch, F., Johnson, A., and Scheuermann, B. The Sniper Attack: Anonymously Deanonymizing and Disabling the Tor Network. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2013* (2014).

[23] Johnson, A., Feigenbaum, J., and Syverson, P. Preventing Active Timing Attacks in Low-Latency Anonymous Communication. In *Proceedings of the 10th Privacy Enhancing Technologies Symposium (PETS 2010)* (July 2010).

[24] Juarez, M., Afroz, S., Acar, G., Diaz, C., and Greenstadt, R. A Critical Evaluation of Website Fingerprinting Attacks. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2014), CCS '14, ACM, pp. 263–274.

[25] Levine, B. N., Reiter, M. K., Wang, C., and Wright, M. K. Timing Attacks in Low-Latency Mix-Based Systems. In *Proceedings of Financial Cryptography* (February 2004), pp. 251–265.

[26] Li, W., and Moore, A. W. A Machine Learning Approach for Efficient Traffic Classification. In *15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2007), October 24-26, 2007, Istanbul, Turkey* (2007), pp. 310–317.

[27] Luo, Y., Xiang, K., and Li, S. Acceleration of Decision Tree Searching for IP Traffic Classification. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (New York, NY, USA, 2008), ANCS '08, ACM, pp. 40–49.

[28] Mathewson, N. Some Thoughts on Hidden Services. `https://blog.torproject.org/category/tags/hidden-services`, 2014. Accessed February 2015.

[29] McCoy, D., Bauer, K., Grunwald, D., Kohno, T., and Sicker, D. Shining Light in Dark Places: Understanding the Tor Network. In *Proceedings of the 8th Privacy Enhancing Technologies Symposium* (July 2008), pp. 63–76.

[30] Mittal, P., Khurshid, A., Juen, J., Caesar, M., and Borisov, N. Stealthy Traffic Analysis of Low-Latency Anonymous Communication Using Throughput Fingerprinting. In *Proceedings of the 18th ACM conference on Computer and Communications Security* (2011), CCS '11, pp. 215–226.

[31] Murdoch, S. J., and Danezis, G. Low-cost traffic analysis of tor. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2005), SP '05, IEEE Computer Society, pp. 183–195.

[32] Øverlier, L., and Syverson, P. Locating Hidden Servers. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy* (May 2006), pp. 100–114.

[33] Panchenko, A., Niessen, L., Zinnen, A., and Engel, T. Website Fingerprinting in Onion Routing Based Anonymization Networks. In *Proceedings of the ACM Workshop on Privacy in the Electronic Society (WPES)* (October 2011), pp. 103–114.

[34] Quinlan, J. R. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[35] Wang, T., Cai, X., Nithyanand, R., Johnson, R., and Goldberg, I. Effective Attacks and Provable Defenses for Website Fingerprinting. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, Aug. 2014), USENIX Association, pp. 143–157.

[36] Wang, T., and Goldberg, I. Improved Website Fingerprinting on Tor. In *Proceedings of the Workshop on Privacy in the Electronic Society (WPES 2013)* (November 2013), ACM.

SecGraph: A Uniform and Open-source Evaluation System for Graph Data Anonymization and De-anonymization

Shouling Ji
Georgia Institute of Technology

Weiqing Li
Georgia Institute of Technology

Prateek Mittal
Princeton University

Xin Hu
IBM Thomas J. Watson Research Center

Raheem Beyah
Georgia Institute of Technology

Abstract

In this paper, we analyze and systematize the state-of-the-art graph data privacy and utility techniques. Specifically, we propose and develop *SecGraph* (available at [1]), a uniform and open-source Secure Graph data sharing/publishing system. In SecGraph, we systematically study, implement, and evaluate 11 graph data anonymization algorithms, 19 data utility metrics, and 15 modern Structure-based De-Anonymization (SDA) attacks. To the best of our knowledge, SecGraph is the first such system that enables data owners to anonymize data by state-of-the-art anonymization techniques, measure the data's utility, and evaluate the data's vulnerability against modern De-Anonymization (DA) attacks. In addition, SecGraph enables researchers to conduct fair analysis and evaluation of existing and newly developed anonymization/DA techniques. Leveraging SecGraph, we conduct extensive experiments to systematically evaluate the existing graph data anonymization and DA techniques. The results demonstrate that (*i*) most anonymization schemes can partially or conditionally preserve most graph utilities while losing some application utility; (*ii*) no DA attack is optimum in all scenarios. The DA performance depends on several factors, e.g., similarity between anonymized and auxiliary data, graph density, and DA heuristics; and (*iii*) all the state-of-the-art anonymization schemes are vulnerable to several or all of the modern SDA attacks. The degree of vulnerability of each anonymization scheme depends on how much and which data utility it preserves.

1 Introduction

Many computing systems generate data with graph structure, e.g., social networks, collaboration networks, and email networks [2–4]. Even mobility traces, e.g., WiFi traces, Bluetooth traces, instant message traces, and check-ins, can be modeled by graphs via applying sophisticated techniques [3–5]. Generally, those data are called *graph data*. For research purposes, data and network mining tasks, and commercial applications, these graph data are often transferred, shared, and/or provided to the public, research community, and/or commercial partners. Since graph data carry a lot of sensitive private information of users/systems who generated them [2, 3], it is critical to protect users' privacy during the data transferring, sharing, and/or publishing.

To protect users' privacy, several anonymization techniques have been proposed to anonymize graph data, which can be classified into six categorizes: Naive ID Removal, Edge Editing (EE) based techniques [6], *k*-anonymity based techniques [7–11], Aggregation/Class/Cluster based techniques [12–14], Differential Privacy (DP) based techniques [15–19], and Random Walk (RW) based techniques [20]. Fundamentally, these techniques try to protect users' privacy by perturbing the original graph's structure while preserving as much data utility as possible.

Following Narayanan and Shmatikov's work [2], many new Structure-based De-Anonymization (SDA, we use DA and SDA interchangeably in this paper) attacks on graph data have been proposed, which can be categorized into two classes: *seed-based attacks*, e.g., Narayanan-Shmatikov's attack [2], and *seed-free attacks*, e.g., Ji et al.'s attack [3]. For both types of attacks, the goal is to de-anonymize anonymized users using their uniquely distinguishable structural characteristics.

Surprisingly, although we already have many sophisticated anonymization techniques (e.g., [6, 7, 12, 15, 20]) and powerful SDA attacks (e.g., [2,3,5,21–24]), whether state-of-the-art anonymization techniques can defend against modern SDA attacks is still an open problem. This is because of the incomplete evaluation of existing anonymization and DA techniques. For anonymization works, they usually only evaluate the data utility performance of their proposed techniques (although some works provide a theoretical security guarantee, these guarantees usually do not hold due to improper assumptions or incomplete considerations as analyzed in Section 4). For DA works, they usually evaluate their attacks' performance without applying state-of-the-art anonymization techniques (e.g., k-anonymity based schemes, DP based schemes) to their test data.

Contributions. To address the above open problem, we systematically study, implement, and evaluate existing graph data anonymization techniques and DA attacks. Specifically, our main contributions are as follows.

(a) We design and implement a Secure Graph data publishing/sharing (SecGraph) system (available at [1]). SecGraph enables data owners to anonymize their data using state-of-the-art anonymization techniques, measure the anonymized data's graph and application utilities, and comprehensively evaluate their data's actual vulnerability against modern DA attacks. To the best of our knowledge, SecGraph is the first such system publicly available to both academia and industry. More importantly, SecGraph provides the first *uniform platform* that enables researchers to conduct accurate comparative studies of anonymization/DA techniques, and to comprehensively understand the resistance/vulnerability of existing or newly developed anonymization techniques, the effectiveness of existing or newly developed DA attacks, and graph and application utilities of anonymized data.

(b) In SecGraph, we systematically analyze, implement, and evaluate 11 state-of-the-art graph data anonymization schemes and 19 graph and application utility metrics. We also analyze the 11 anonymization schemes with respect to the 19 utility metrics, both analytically and experimentally. The evaluation results demonstrate that most existing anonymization algorithms can partially or conditionally preserve most graph utilities. However, all the anonymization schemes lose one or more application utility.

(c) We summarize and analyze the fundamental properties of existing SDA attacks. Then, we systematically implement and evaluate 15 modern SDA attacks on real-world graph datasets. Our results show that modern SDA attacks are powerful and robust to seed mapping errors. Furthermore, no attack is optimum in all scenarios. The DA performance of an attack depends on the similarity between the anonymized and auxiliary data, graph density, DA heuristics, etc.

(d) We analytically and experimentally evaluate the performance of existing graph data anonymization schemes on defending against modern SDA attacks. We find that existing anonymization techniques are vulnerable to modern SDA attacks. Their degree of vulnerability depends on how much data utility is preserved in the anonymized data.

Abbreviations. For convenient reference, we summarize the used abbreviations in Table 1.

Roadmap. In Section 2, we study existing graph data anonymization schemes and their utility performance. In Section 3, we study modern SDA attacks. In Section 4, the effectiveness of existing anonymization schemes against modern DA attacks is analyzed. We systematically implement and evaluate SecGraph in Section 5. The future research directions are discussed in Section 6. We conclude this paper in Section 7.

Table 1: Abbreviations and acronyms.

Terms	SDA	Structure-based De-anonymization
	DA	De-anonymization
	SF	Seed-Free
Anonymizatoin	EE	Edge Editing
	DP	Differential Privacy
	RW	Random Walk
	k-NA	k-Neighborhood Anonymity
	k-DA	k-Degree Anonymity
	k-auto	k-automorphism
	k-iso	k-isomorphism
Utility metrics	Deg.	Degree
	JD	Joint Degree
	ED	Effective Diameter
	PL	Path Length
	LCC	Local Clustering Coefficient
	GCC	Global Clustering Coefficient
	CC	Closeness Centrality
	BC	Betweenness Centrality
	EV	Eigenvector
	NC	Network Constraint
	NR	Network Resilience
	Infe.	Infectiousness
	RX	Role extraction
	RE	Reliable Email
	IM	Influence Maximization
	MINS	Minimum-sized Influential Node Set
	CD	Community Detection
	SR	Secure Routing
	SD	Sybil Detection
De-anonymization	DV	Distance Vector [5]
	RST	Randomized Spanning Tress [5]
	RSM	Recursive Subgraph Matching [5]
	DeA	De-Anonymization [25]
	ADA	Adaptive De-Anonymization [25]
	BDK	Backstrom et al.'s attacks [26]
	NS	Narayanan-Shmatikov's attack [2]
	NSR	Narayanan et al.'s attack [21]
	NKA	Nilizadeh et al.'s attack [22]
	PFG	Pedarsani et al.'s attack [23]
	YG	Yartseva-Grossglauser's attack [27]
	KL	Korula-Lattanzi's attack [24]
	JLSB	Ji et al.'s attack [3]

2 Graph Anonymization

2.1 Status Quo

Generally, existing graph data anonymization techniques can be classified into six categories. We discuss each category as follows.

Naive ID Removal. To anonymize graph data, a straightforward method is *naive ID removal*. Although this method has been demonstrated to be extremely vulnerable to SDA attacks, it is still widely used because of its simplicity, ease of applicability, and scalability [2, 3, 5, 26, 28].

Edge Editing based Anonymization. To protect graph data's privacy, Ying and Wu proposed spectrum preserved Edge Editing (EE) based schemes *Add/Del* and *Switch* [6]. Under *Add/Del*, k randomly chosen edges will be added followed by the deletion of another k randomly chosen edges. Under *Switch*, k random *edge switches* are conducted.

***k*-anonymity.** k-anonymity has been widely used to anonymize relational data [29, 30]. Similarly, much effort has been spent to extend k-anonymity to graph data [7–11]. To defend against neighborhood attacks, Zhou and Pei proposed *k-Neighborhood Anonymity* (k-NA) for graph data [7]. In another work, Liu and Terzi considered *degree attacks* and proposed *k-Degree Anonymity* (k-DA) for graph data, under which for each user, there exists at least $k-1$ other users with the same degree [8]. In [9], Zou et al. simultaneously considered four types of structural attacks on graph data and proposed *k-automorphism* (k-auto), where each user always has $k-1$ other symmetric users with respect to $k-1$ automorphic functions. Another similar work is [10], where Cheng et al. proposed *k-isomorphism* (k-iso) to defend against structural attacks. Under k-iso, a graph is partitioned and anonymized into k disjoint isomorphic subgraphs. In [11], Yuan et al. considered personalized privacy protection for anonymizing graph data in terms of both semantic and structural information.

Aggregation/Class/Cluster based Anonymization. Another popular idea to protect graph data is to group users into *clusters* (equivalently, *groups, classes*). In [12], Hay et al. proposed an *aggregation based graph anonymization* algorithm, which first partitions users and then describes the graph at the level of partitions. Another work, at the semantics level, is [13], where Bhagat et al. designed a class-based anonymization algorithm. In [14], Thompson and Yao presented two *cluster-based anonymization* schemes for graph data.

Differential Privacy. Differential Privacy (DP) is an emerging anonymization technique with a strong privacy guarantee [31, 32]. Initially, DP was proposed for statistical databases [31]. Recently, there have been works that seek to enable differentially private graph data release. Aiming at protecting *edge/link privacy*, defined as the privacy of users' relationship in graph data, in [15], Sala et al. introduced *Pygmalion*, a differentially-private graph model. To bypass many difficulties encountered when working with the worst-case sensitivity [15], Proserpio recently presented a general platform, named *wPING*, for differentially private data analysis and publishing [16, 17]. Similar to [15], Wang and Wu also employed the dK-graph generation model for enforcing edge DP in graph anonymization [18]. Another recent work for edge DP is [19], where Xiao et al. proposed a Hierarchical Random Graph (HRG) model based scheme to meet edge DP.

Random Walk based Anonymization. In [20], Mittal et al. proposed a *Random Walk (RW) based anonymization* technique for preserving link (edge) privacy. In this technique, an edge in the original graph is replaced by a RW path.

2.2 Anonymization and Utility

Generally, an anonymization scheme can be evaluated from two perspectives: *data utility preservation* and *resistance to DA attacks*. However, most, if not all, existing graph anonymization works have not been significantly evaluated with respect to their utility or resistance to DA attacks. On one hand, most existing graph anonymization works only conducted limited evaluations on their utility preservation, e.g., degree distribution, path length distribution, which are insufficient to understand their value for high-level data mining tasks and applications, e.g., sense-making, search for similar users, user classification, reliable email, influence maximization. On the other hand and more seriously, to the best of our knowledge, no work (including existing DA works) actually evaluated the resistance of state-of-the-art anonymization techniques against modern SDA attacks.

To address these concerns, we comprehensively analyze the utility of existing graph data anonymization algorithms in this subsection and defer the detailed resistance analysis to Section 4. Before performing the analysis, we first present the used utility metrics, which can be classified as *graph utility metrics* or *application utility metrics*.

2.2.1 Graph Utility Metrics

Graph utility captures how the anonymized data preserves fundamental structural properties of the original graph after applying an anonymization technique. Particularly, we examine 12 graph utility metrics of existing anonymization schemes as follows[1].

Degree (Deg.), which refers to the degree distribution; *Joint Degree (JD)*, which refers to the joint degree distribution of a graph; *Effective Diameter (ED)*, which is defined as the minimum number of hops in which 90% of all connected pairs of nodes can reach each other; *Path Length (PL)*, which refers to the distribution of the

[1] Without of causing confusion, we interchangeably use node and user in this paper.

shortest path lengths between all pairs of users; *Local Clustering Coefficient (LCC) and Global Clustering Coefficient (GCC). Clustering coefficient* measures the degree to which users in graph data tend to cluster together. *Closeness Centrality (CC)*, which is defined as the *inverse of the farness* of a user within a graph and measures how long it takes to spread information from a user to all other users sequentially; *Betweenness Centrality (BC)*, which quantifies the number of times a user acts as a bridge along the shortest path between two other users; *EigenVector (EV)*. The EV of the adjacency matrix A of a graph G is a non-zero vector $\mathbf{v}$ such that $A\mathbf{v} = \lambda\mathbf{v}$, where λ is some scalar multiplier; *Network Constraint (NC)*, which measures the extent to which a user links to others that are already linked to each other; *Network Resilience (NR)* [33], which measures how robust a graph is and is defined as the number of users in the *largest connected component* when users are removed from the graph in the degree decreasing order; and *Infectiousness (Infe.)* [34], which measures the number of users infected by a disease, given that a randomly chosen user is infected and each infected user transmits this disease to its neighbors with some infection rate.

2.2.2 Application Utility Metrics

In reality, most data is published/shared for data/network mining tasks, high-level applications, etc. Therefore, besides examining data's fundamental structural utility, it is also crucial to ensure that the anonymized data is useful for practical applications. Toward this objective, we evaluate 7 popular application utility metrics for anonymization schemes as follows.

(a) *Role eXtraction (RX) [35].* Based on users' structural behavior, users in a graph can be labeled as having different roles, e.g., *clique members*, *periphery-nodes*. RX is an important operation for graph data that is useful for many network mining tasks such as sense-making. We measure the RX utility of an anonymization scheme using the method in [35].

(b) *Reliable Email (RE) [36].* RE is a whitelisting system leveraging users' neighborhoods to filter and block spam emails. To evaluate the structural utility of an anonymization scheme with respect to RE, we take a similar method as in [15] to compute the number of users who can be spammed by a fixed number of compromised neighbors in a graph.

(c) *Influence Maximization (IM) [37].* The IM problem seeks to find a set of θ users such that these θ users have the maximum influence to the network under some influence propagation model. IM is important for many real world applications, e.g., advertisements.

(d) *Minimum-sized Influential Node Set (MINS) [38].* MINS is another popular and important application utility metric that leverages a graph's structure to identify the minimum-sized set of influential nodes, such that all other nodes in the network could be influenced with a probability above a threshold. MINS can be used in many meaningful applications, e.g., social problems alleviation, new products promotion.

(e) *Community Detection (CD) [39].* CD is a popular application on graph data which enables comprehensive analysis of a network structure and supports other applications, e.g., classification, routing (information propagation). To measure the CD utility of an anonymization scheme, we employ the hierarchical agglomeration algorithm proposed in [39].

(f) *Secure Routing (SR) [40].* The structure of graph data can also be used to improve the performance of secure routing for systems such as P2P systems. For our purpose, we evaluate the SR application utility of an anonymization scheme using the method designed in [40].

(g) *Sybil Detection (SD) [41].* Sybil attacks are a serious threat to both centralized and distributed systems, e.g., recommendation systems, anonymity systems. For our purpose, we evaluate the SD application utility of an anonymization scheme using the method in [41].

2.2.3 Anonymization vs Utility

We are ready to analyze the utility performance of existing graph data anonymization techniques. We summarize the graph and application utilities, and Resistance to SDA attacks (R2SDA) (e.g., [2, 3, 25, 27]) of existing graph anonymization schemes in Table 2. We analyze the results in Table 2 as follows.

For the Naive ID removal scheme, it is straightforward that it preserves all the data utility. However, it is also the most vulnerable scheme to SDA attacks.

Since *Add/Del* randomly adds and deletes edges, which is an global edge edition operation and thus it may change many fundamental structural properties of a graph. It follows that it can conditionally or partially preserve both graph and application utilities. However, utilities like JD, GCC, NC, CD, and MINS would be destroyed if too many existing edges are deleted while new edges are added. For *Switch*, it switches two randomly selected qualified edges, which preserves the degree of each user. Consequently, *Switch* can preserve Deg. and partially preserve most other utilities. Furthermore, compared to *Add/Del*, *Switch* can conditionally preserve the RX and CD utilities which are destroyed in *Add/Del*. This is because that *Add/Del* randomly changes users' degree in the global edge edition process and thus some global structure-sensitive application utility is lost or significantly affected. Furthermore, *Add/Del* and *Switch* cannot defend against modern SDA attacks as shown in [2, 3, 5].

Table 2: Analysis of existing graph anonymization techniques. ✓ = preserving the utility, ◐= partially preserving the utility, ♦ = conditionally preserving the utility depending on parameters and considered data (based on our analysis, it is necessary to distinguish *partially* and *conditionally* preserving a data utility. For instance, k-DA conditionally preserves the Deg. utility depending on k while *Add/Del* can partially preserve the Deg. utility for an arbitrary k), ✗ = not preserving the utility, and **n/a** = evaluation not available in existing works.

	graph utility												application utility							R2SDA
	Deg.	JD	ED	PL	LCC	GCC	CC	BC	EV	NC	NR	Infe.	RX	RE	IM	MINS	CD	SR	SD	
Naive	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗
Add/Del [6]	◐	♦	◐	◐	♦	♦	◐	◐	◐	♦	◐	◐	✗	◐	◐	♦	✗	♦	♦	✗
Switch [6]	✓	♦	♦	◐	♦	♦	◐	◐	◐	◐	♦	◐	♦	◐	◐	♦	♦	◐	◐	✗
k-NA [7]	♦	♦	♦	♦	♦	♦	◐	♦	♦	◐	◐	◐	✗	◐	♦	♦	♦	♦	♦	**n/a**
k-DA [8]	♦	♦	♦	♦	♦	♦	◐	♦	♦	◐	◐	◐	✗	◐	♦	♦	♦	♦	♦	**n/a**
k-auto [9]	♦	♦	♦	♦	♦	♦	◐	♦	♦	◐	◐	◐	✗	◐	♦	♦	♦	♦	♦	**n/a**
k-iso [10]	♦	♦	✗	✗	♦	✗	✗	✗	✗	♦	✗	✗	✗	✗	✗	♦	♦	✗	♦	**n/a**
Aggregation [12]	♦	♦	♦	♦	♦	♦	◐	♦	♦	◐	◐	◐	✗	◐	♦	♦	♦	♦	♦	**n/a**
Cluster [14]	♦	♦	♦	♦	♦	♦	◐	♦	♦	◐	◐	◐	✗	◐	♦	♦	♦	♦	♦	**n/a**
DP [15]	♦	♦	♦	◐	♦	♦	◐	♦	♦	♦	♦	◐	✗	◐	♦	♦	✗	♦	♦	**n/a**
DP [16, 17]	♦	♦	♦	◐	♦	♦	◐	♦	♦	♦	♦	◐	✗	◐	♦	♦	✗	♦	♦	**n/a**
DP [18]	♦	♦	♦	◐	♦	♦	◐	♦	♦	♦	♦	◐	✗	◐	♦	♦	✗	♦	♦	**n/a**
DP [19]	♦	♦	♦	◐	♦	♦	◐	♦	♦	♦	♦	◐	✗	◐	♦	♦	✗	♦	♦	**n/a**
RW [20]	✓	♦	♦	♦	♦	✗	◐	♦	♦	◐	♦	◐	✗	◐	♦	✗	✗	◐	◐	**n/a**

The k-anonymity based anonymization schemes k-NA [7], k-DA [8], and k-auto [9] can partially/conditionally preserve the graph and most application utilities except for the RX utility. This is because the fundamental idea of k-anonymity based schemes is to make k users/subgraphs structurally similar. Therefore, there is a tradeoff between anonymity and utility. If k is large, more users will be structurally similar while more utility will be lost. On the other hand, if k is chosen to be small, more utility will be preserved at the cost of lower anonymity guarantee. Furthermore, since every user is guaranteed to be structurally similar to at least $k-1$ other users while the RX utility tries to distinguish users based on their structural differences, it turns out k-anonymity based schemes cannot preserve the RX utility. As we discussed before, k-iso achieves structure anonymization by partitioning the original graph into k isomorphic subgraphs. Therefore, several fundamental properties of a graph will be destroyed, e.g., connectivity. It follows that several important graph and application utilities are lost in k-iso, e.g., PL, GCC, NR, Infe., RX, RE, IM, and SR. Finally, compared with other schemes, k-NA, k-auto, and k-iso have higher computational complexities.

Similar to k-anonymity based schemes, the cluster based schemes [12, 14] can conditionally/partially preserve graph and application utilities except for RX. This is because the fundamental idea of cluster based schemes is to make the users within a cluster structurally indistinguishable. Therefore, to what extent these schemes can preserve data utility depends on the cluster size setting. Again, since RX is achieved based on users' structural difference, this utility is not preserved in cluster based schemes.

For DP based schemes (e.g., [15, 19]), their main objective is to protect link privacy by perturbing the edges of a graph. The fundamental idea of these schemes is to make an anonymized graph structurally similar to its neighboring graphs and thus an adversary cannot infer the existence of an edge. Therefore, they can conditionally/partially preserve most graph and application utilities. However, if a high level of privacy is guaranteed, many edges in the graph are changed. Furthermore, similar to *Add/Del*, the edge perturbation in DP also belongs to global edge edition. Therefore, the global structure-sensitive high-level application utilities, e.g., RX, MINS, and CD, are destroyed or significantly reduced in DP based schemes.

In the RW based scheme [20], link privacy is achieved by replacing a random walk path with an edge, and thus this scheme, theoretically, will not change the degree distribution of the original data. It follows that several utilities, e.g., Deg., RX, SD, NR, Infe., can be preserved or partially preserved. However, some other global utilities, e.g. JD, GCC, are lost in the RW based scheme due to the significant change of the overall graph structure.

From Table 2, no existing work evaluates the resistance of state-of-the-art anonymization schemes against modern SDA attacks. Although most of the schemes have nice theoretical privacy guarantees, unfortunately, that privacy analysis cannot guarantee that they can defend against modern SDA attacks due to the improper model of the adversary's auxiliary information, problematic assumptions, etc. Therefore, aiming to address this open problem, we evaluate the effectiveness of existing graph data anonymization schemes against modern SDA attacks in Sections 4 and 5.

3 Graph De-anonymization

3.1 Graph Data DA

3.1.1 Seed-based DA

When de-anonymizing graph data, it is intuitive to identify some users first as seeds. Then, the large scale DA is bootstrapped from these seeds. In [26], Backstrom et al. presented both active attacks and passive attacks to graph data. However, the attacks in [26] have several limitations, e.g., they are not scalable and they leverage sybil users that can be detected by modern sybil defense techniques [41]. To improve the attacks in [26], Narayanan and Shmatikov presented a scalable two-phase DA attack to social networks [2]. In the first phase, some seed users are identified between the anonymized graph and the auxiliary graph. In the second phase, starting from the identified seeds, a self-reinforcing DA propagation process is iteratively conducted based on both graphs' structural characteristics, e.g., node degrees, nodes' eccentricity, edge directionality. Later, Narayanan et al. employed a simplified version of the attack in [2] (using less DA heuristics) for link prediction [21]. In [22], Nilizadeh et al. extended Narayanan and Shmatikov's attack by proposing a community-enhanced DA scheme of social networks. Actually, the community-level DA in [22] can also be applied to enhance other seed-based DA attacks (e.g., [5, 25]).

In [5], Srivatsa and Hicks presented three attacks to de-anonymize mobility traces, which can be modeled as contact graphs by applying multiple preprocessing techniques (e.g., [5]). Similar to Narayanan et al.'s attacks [2, 21], Srivatsa-Hicks' attacks also consist of two phases, where the first phase is for seed identification and the second phase is for mapping (DA) propagation. To achieve mapping propagation, Srivatsa and Hicks proposed three heuristics based on Distance Vector (DV), Randomized Spanning Trees (RST), and Recursive Subgraph Matching (RSM). In [25], Ji et al. proposed two two-phase DA attack frameworks, namely De-Anonymization (DeA) and Adaptive De-Anonymization (ADA), which are workable when the auxiliary data only has partial overlap with the anonymized data.

In [24,27], besides quantifying the de-anonymizability of graph data, the authors also proposed DA attacks. In [27], Yartseva and Grossglauser proposed a simple percolation-based DA algorithm to graph data. Given a seed mapping set, the algorithm incrementally maps every pair of users (from the anonymized and auxiliary graphs respectively) with at least *r* neighboring mapped pairs, where *r* is a predefined mapping threshold. Another similar attack was presented by Korula and Lattanzi [24], which also starts from a seed set and iteratively maps a pair of users with the most number of neighboring mapped pairs.

Table 3: Analysis of existing graph DA techniques. SF = seed-free, AGF = auxiliary graph-free, SemF = semantics-free, A/P = active/passive attack, Scal. = scalable, Prac. = practical, Rob. = robust to noise, ✓ = true, ◐= partially true, ♦ = conditionally true, and ✗ = false.

	SF	AGF	SemF	A/P	Scal.	Prac.	Rob.
BDK [26]	✓	✓	✓	A, P	✗	◐	✗
NS [2]	✗	✗	✓	P	✓	✓	✓
NSR [21]	✗	✗	✓	P	✓	✓	✓
NKA [22]	♦	✗	✓	P	♦	♦	♦
DV [5]	✗	✗	✓	P	♦	♦	✓
RST [5]	✗	✗	✓	P	♦	♦	✓
RSM [5]	✗	✗	✓	P	♦	♦	✓
PFG [23]	✓	✗	✓	P	✓	♦	♦
YG [27]	✗	✗	✓	P	✓	♦	✓
DeA [25]	✗	✗	✓	P	✓	✓	✓
ADA [25]	✗	✗	✓	P	✓	✓	✓
KL [24]	✗	✗	✓	P	✓	♦	✓
JLSB [3]	✓	✗	✓	P	✓	✓	✓

3.1.2 Seed-free DA

Taking another approach, some powerful seed-free DA attacks on graph data have been proposed. Using degrees and distances to other nodes as each node's fingerprints, Pedarsani et al. proposed a Bayesian model based seed-free algorithm for graph data DA [23]. Another seed-free DA attack to graph data was presented by Ji et al. [3]. Unlike previous attacks, Ji et al.'s attack is an optimization based single-phase cold start algorithm.

3.2 Graph DA Analysis

In this subsection, we analyze the performance of existing graph data DA algorithms. For convenience, in the rest of this paper, we denote Backstrom et al.'s attacks [26] by BDK (the initials of the authors), Narayanan-Shmatikov's attack [2] by NS, Narayanan et al.'s attack [21] by NSR, Nilizadeh et al.'s attack [22] by NKA, Srivatsa-Hicks' three attacks [5] by DV, RST, and RSM, respectively, Pedarsani et al.'s attack [23] by PFG, Yartseva-Grossglauser's attack [27] by YG, Ji et al.'s two attacks [25] by DeA and ADA, respectively, Korula-Lattanzi's attack [24] by KL, and Ji et al.'s attack [3] by JLSB. We show our analytical results in Table 3 and discuss the result as follows.

Except for BDK, all the existing SDA attacks are passive attacks and require auxiliary graphs to perform the attack, i.e., they employ the structural similarity between the the anonymized graph and the auxiliary graph to break the anonymity. However, when we examine the anonymization schemes in Table 2, we find that none properly consider such auxiliary information in their threat models.

To perform BDK attacks [26], an adversary either has to insert some Sybil users in the dataset before the actual anonymized data release, or has to be an internal user that knows its neighborhoods. In either case, such attacks can only de-anonymize some users but cannot de-anonymize users in large scale. Furthermore, the attacks cannot tolerate any topological change of the original data. Therefore, BDK attacks are not scalable or robust. These attacks require that an adversary successfully launches Sybil users or be an internal user that obtains his neighborhoods.

All the examined DA attacks are semantics-free. This is because the structural information itself is sufficient to perfectly or partially de-anonymize graph users. Furthermore, compared to semantics information, structural information is widely available in large scale, resilient to noise, and easily computable [2, 3, 5]. Following this fact, all the attacks except for BDK are (conditionally) scalable, practical, and robust.

Specifically, DV, RST, and RSM [5] are conditionally scalable and practical. This is because they are not computationally feasible when the number of seeds is large. PFG [23] is conditionally practical and robust. This is because it is very sensitive to the graph density of the anonymized data. Generally, this attack is suitable for sparse graphs however it has a significant performance degradation as the graph density increases. YG [27] is conditionally practical because it is designed to de-anonymize users of degree no less than 4 in the anonymized data. In many real world graph datasets, the users with degree less than 4 could dominate or take a significant portion of graph data based on the statistics in [3]. The conditional practicability of KL [24] comes from its improper assumption that $\Theta(\iota \cdot n)$ ($\iota \in (0,1]$ is a constant and n is the number of nodes in a graph) seeds are available, which is too strong to hold for real world DA attacks. Note that, the community-level DA of NKA [22] is scalable (with complexity of $O(n^2)$). However, the NKA [22] is conditionally scalable, practical, and robust. This is because, if the community-level DA of NKA [22] is employed to enhance DV, RST, RSM, YG, and/or KL, it is conditionally scalable, practical, and/or robust. NS [2], NSR [21], DeA, ADA, and JLSB [3, 25] adaptively perform DA employing several heuristics based on a graph's local and global structural characteristics. It follows that they are scalable, practical, and robust as long as similarity exists between anonymized graphs and auxiliary graphs.

Both seed-based attacks (e.g., NS, DV) and seed-free attacks (e.g., PFG, JLSB) have advantages depending on the application scenarios. On one hand, seed-based attacks are more stable with respect to de-anonymizing arbitrary anonymized graphs. The reason is straightforward since seed knowledge provides more auxiliary information to an adversary. On the other hand, it is possible that in some scenarios seeds are not available, and thus seed-free attacks are more general. Furthermore, if there is some error in the seed seeking phase (which is possible in real world attacks), seed-based attacks will suffer performance de-gradation or will possibly fail.

Table 4: DA attacks vs anonymization techniques. Naive = naive ID removal, EE = EE based schemes [6], *k*-anony. = *k*-anonymity based schemes [7]- [10], Cluster = cluster based schemes [12, 14], DP = DP based schemes [15]- [19], RW = the random walk based scheme [20], and ✓, ♦, and ✗ = the anonymization scheme is vulnerable, conditionally vulnerable, and invulnerable (i.e., resistant) to the DA attack, respectively.

	Naive	EE	*k*-anony.	Cluster	DP	RW
BDK [26]	✓	✗	✗	✗	✗	✗
NS [2]	✓	✓	♦	♦	✓	✓
NSR [21]	✓	✓	♦	♦	✓	✓
NKA [22]	✓	♦	♦	♦	✗	✗
DV [5]	✓	✓	♦	♦	✓	✓
RST [5]	✓	✓	♦	♦	✓	✓
RSM [5]	✓	✓	♦	♦	✓	✓
PFG [23]	✓	✓	♦	♦	✓	✓
YG [27]	✓	✓	♦	♦	✓	✓
DeA [25]	✓	✓	♦	♦	✓	✓
ADA [25]	✓	✓	♦	♦	✓	✓
KL [24]	✓	✓	♦	♦	✓	✓
JLSB [3]	✓	✓	♦	♦	✓	✓

4 Anonymization vs DA Analysis

As we analyzed in Tables 2 and 3, understanding the vulnerability/resistance of state-of-the-art graph data anonymization schemes against modern SDA attacks is still an open problem. After carefully analyzing existing anonymization and DA techniques, we summarize the ***vulnerability*** of existing anonymization schemes in Table 4. We further experimentally validate our analysis in Section 5. Below, we analyze and discuss the results in Table 4.

It has been shown in both academia and in practice that the naive ID removal anonymization cannot protect graph data's privacy. Therefore, naive anonymization is vulnerable to all the existing SDA attacks.

As we analyzed before, all other state-of-the-art anonymization schemes (e.g., EE, *k*-anony., Cluster, DP, and RW) are resistant to BDK attacks. Again, this is because an assumption of BDK attacks is that data is anonymized by the naive ID removal technique.

For EE based anonymization schemes ([6]), they are conditionally vulnerable to NKA [22] and vulnerable to all the other modern SDA attacks [2,3,25,27]. This is because although EE can partially modify the structure of a graph, to preserve data utility, many structural properties, e.g., neighborhood, degree distribution, closeness/betweenness centrality distribution, and path length distribution, are generally preserved. Therefore, given an auxiliary graph consisting of the same or overlapping group of users with the anonymized graph, powerful DA heuristics can be designed based on these structural properties to break the privacy of EE based anonymization schemes. Furthermore, the availability of seed users make such heuristics more robust to the noise introduced by EE. For instance, NS breaks EE by employing degree and neighborhood similarity [2], DV, RST, and RSM break EE by employing path length and neighborhood similarity [5], DeA and ADA break EE by employing centrality similarity [25], etc. As we analyzed in Table 2, EE based anonymization schemes (e.g., *Add/Del*) may destroy graphs' community utility, and thus they are conditionally vulnerable to NKA [22].

k-anonymity based anonymization schemes ([7]-[10]) are conditionally vulnerable to modern SDA attacks [2, 3, 25, 27]. The reasons are as follows: *k*-anonymity is initially designed for traditional relational data, which makes a user semantically indistinguishable with $k-1$ other users. Unlike relational data, which are structurally independent of each other, users in graph data have strong structural correlation in addition to semantic similarity. When researchers extended *k*-anonymity to graph data, they extended the concept of traditional semantics to graph data as different structural properties (e.g., degree, neighborhood, and subgraph), and designed schemes to make *k* users structurally indistinguishable with respect to some structural semantics, i.e., degree, neighborhood, subgraph, etc. However, even if users in graph data cannot be distinguished with respect to some structural semantics, e.g., degree, neighborhood, subgraph, they can be de-anonymized by other structural semantics, e.g., path length distribution, closeness centrality, betweenness centrality, or the combinations of several structural semantics. Theoretically, the only way to make users indistinguishable with respect to all structural semantics is to make a graph *completely connected* or *disconnected*, which also implies that all the data utility is destroyed. Therefore, as long as some data utility is preserved in the anonymized data, *k*-anonymity based schemes are vulnerable to modern SDA attacks. The degree of vulnerability depends on how much data utility is preserved.

Cluster based schemes ([12, 14]) are also conditionally vulnerable to modern SDA attacks [2,3,25,27]. The analysis is similar to that of *k*-anonymity. The fundamental idea of cluster based schemes is to cluster users first and then to make the users within a cluster indistinguishable with respect to neighborhoods. Again, even if users are indistinguishable by neighborhoods, they can be de-anonymized by other structural semantics or the combinations of other semantics, e.g., centralities scores, path length distribution. Consequently, cluster based schemes are vulnerable as long as some data utility, especially graph utilities, are preserved in the anonymized data, and the vulnerability depends on the amount of data utility preserved.

DP and RW based schemes ([15]- [20]) are vulnerable to modern SDA attacks except NKA [22]. The reasons are as follows: First, they are designed with the objective of protecting the link privacy of graph data and no dedicated node privacy protection techniques are considered. Second, to protect link privacy, the edges are perturbed in DP based schemes and random walk paths are replaced by edges in the RW based scheme, both with a nice theoretical privacy guarantee. However, after the edge anonymization process, many data utilities, e.g., degree, path length distribution, are still preserved. This implies that, given an auxiliary graph, users are still de-anonymizable based on several structural semantics under DP and RW based schemes. Furthermore, as shown by Narayanan et al. in [21], link privacy can be breached after de-anonymizing the users in an anonymized graph (we also employ the same approach to break users' link privacy [1]). Again, as we analyzed in Table 2, since DP and RW based schemes cannot preserve data's community utility, they are resistant to NKA.

In summary, based on our analysis, state-of-the-art anonymization schemes are still vulnerable to modern DA attacks. The fundamental reasons are: first, existing anonymization schemes only ensure that graph data users are indistinguishable with respect to some structural semantics (properties). However, other structural semantics, especially global ones, and the combinations of multiple structural semantics can still enable effective DA of users; and second, as one of the main objectives, all the anonymization schemes try to preserve as much data utility as possible. However, data utility from the adversary's perspective is equivalent to structural information, which can be used along with an auxiliary graph for conducting powerful DA attacks.

5 SecGraph

As we found when discussing existing anonymization and DA techniques, they all have limitations when evaluating the techniques' performance. For instance, it is still an open problem to understand the resistance/vulnerability of state-of-the-art anonymization schemes against modern DA attacks. To address this open problem, we implement a Secure Graph data publishing/sharing (SecGraph) system.

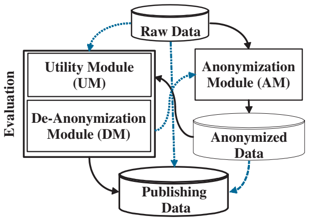


Figure 1: SecGraph: system overview.

5.1 System Overview

The overview of SecGraph is shown in Fig.1. SecGraph consists of three main modules: Anonymization Module (AM), Utility evaluation Module (UM), and DA evaluation Module (DM). The main functions of each module are briefly summarized as follows.

AM: the main function of this module is to anonymize raw graph data and generate anonymized data. In this module, we implement 11 state-of-the-art graph data anonymization schemes, including EE based algorithms [6], *k*-anonymity based algorithms and its variants [7–11], aggregation/class/cluster based algorithms [12–14], differential privacy based algorithms [15–17,19], and the random walk based algorithm [20].

UM: in this module, we evaluate raw/anonymized data's utility with respect to the 12 graph utility metrics and 7 application utility metrics as defined in Section 2.2. With the UM, we can determine whether the data to be published/shared (e.g., the anonymized data) satisfies required utility requirements. We can also evaluate how an anonymization algorithm preserves data utility.

DM: in this module, we implement 15 SDA algorithms (all the existing SDA algorithms, to the best of our knowledge). By this module, the security of data to be published/shared can be evaluated with real-world SDA attacks. More importantly, the effectiveness of an anonymization algorithm can be examined by this module, i.e., whether the anonymized data of an anonymization algorithm is resistant to modern SDA attacks.

We make further remarks on SecGraph and its modules and functions as follows.

(a) From Fig.1, raw data can be published/shared in multiple forms depending on the data owners' requirements on the security/privacy and utility of the data to be published. Each path in Fig.1 represents a data publishing scenario. For instance, the path *raw data* → *publishing data* means to publish the raw data directly. The path *raw data* → *AM* → *anonymized data* → *evaluation* → *publishing data* means that the raw data is anonymized first. Then, the anonymized data will be evaluated with respect to utility and/or practical de-anonymizability before actual publishing. The anonymization and evaluation process may be repeated several times until certain security and utility requirements are met.

(b) To the best of our knowledge, SecGraph is the first implemented uniform secure graph data publishing system, which systematically and comprehensively integrates state-of-the-art anonymization schemes, DA schemes, and graph/application utility measurements. The significance of SecGraph to the graph data anonymization and DA area lies in the following aspects. First, SecGraph enables data owners to conveniently and freely choose any modern anonymization algorithm to anonymize their data. They can also employ different evaluation modules to examine whether the anonymized data meets their security/privacy and utility requirements. Second, SecGraph is a uniform platform for testing and comparing different anonymization and DA algorithms. Previously, due to the lack of a uniform system, existing anonymization/DA algorithms are often proposed and implemented on separate platforms and different environments/settings. Consequently, a number of implementation and evaluation differences (e.g., particular assumptions, models, evaluation datasets, programming, testing environments, parameter settings) limit researchers' understanding of the performance of existing anonymization and DA algorithms in different scenarios. However, as a uniform platform, SecGraph can reduce the evaluation bias caused by implementation and testing differences as much as possible. Therefore, SecGraph allows data owners to choose and compare the actual performance of different data anonymization algorithms on their data and thus to make the best decision. Additionally, SecGraph allows data anonymization researchers to compare their anonymization schemes to existing solutions as well as to examine their schemes' resistance against modern DA attacks. SecGraph also allows data DA researchers to evaluate the performance of new DA attacks by de-anonymizing the anonymized data of state-of-the-art anonymization schemes. Therefore, SecGraph is helpful to both data owners and researchers in conveniently applying existing schemes, comprehensively understanding existing algorithms, and effectively developing new anonymization/DA techniques.

(c) Besides providing a uniform platform, SecGraph is an easily portable and extendable system. First, the algorithms in SecGraph are implemented in Java and thus it is system independent. Second, all the modules of SecGraph are independent of each other, which means that each module can work individually. Additionally, as shown in Fig.1, multiple modules can also work together to perform data anonymization, utility evaluation, and de-anonymizability evaluation. Third, all the schemes/measurements within each module are independent, which means that they can be implemented, evaluated, and employed independently. Furthermore, newly developed anonymization/DA schemes and utility metrics can be easily integrated into SecGraph.

5.2 System Implementation

The implementation of SecGraph is as follows.

In the AM, we implement 11 algorithms, which cover all the categories of state-of-the-art anonymization techniques. Specifically, the implemented anonyzation algorithms are naive ID removal, two EE based algorithms *Add/Del* [6] and *Switch* [6], two *k*-anonymity based algorithms *k*-DA [8] and *k*-iso [10], two cluster based algorithms bounded *t*-means clustering [14] and union-split clustering [14], three DP based algorithms Sala et al.'s scheme [15], Proserpio et al.'s scheme [16, 17], and Xiao et al.'s scheme [19], and one RW based algorithm [20]. Note that, we do not implement all the algorithms discussed in Section 2.1 even though we cover all the categories. The implementation criteria includes representativeness, scalability, and practicality, which led us to implement the latest, scalable, and practical schemes.

In the UM, we implemented the 12 graph utility metrics and 7 application utility metrics as discussed in Section 2.2.

In the DM, we implement all the 15 SDA attacks discussed in Section 3.1. To the best of our knowledge, these are all of the existing SDA attacks.

5.3 SecGraph-based Analysis

5.3.1 Primary Datasets

The employed datasets for evaluation are *Enron*, an email network consisting of 36.7K users and .2M edges, and *Facebook*, a Facebook friendship network in the New Orleans area consisting of 63.7K users and .82M edges [3, 4].

5.3.2 Anonymization vs Utility

In this subsection, we evaluate the utility performance of anonymization algorithms. Due to the space limitation, we do not show the evaluation results of all the implemented algorithms. Particularly, we demonstrate the results of *Switch* [6], *k*-DA [8], *union-split clustering* [14], the improved version of Sala et al.'s DP scheme [15–17], and RW [20] which represent all the categories of anonymization algorithms. The evaluation methodology is that we first anonymize the original graph by an algorithm, and then measure how each data utility is preserved in the anonymized graph compared to the original graph. Specifically, when measuring utilities Deg., JD, PL, LCC, CC, BC, NC, NR, Infe., RX, and RE, we measure the *cosine similarity* between their distributions in the anonymized and original graphs; when measuring ED, GCC, and EV, we measure their *ratios* between the anonymized and original graphs; and when measuring MINS and CD, we measure their *Jaccard similarity* in the anonymized and original graphs.

We demonstrate the results in Table 5. (more results are available in [1]). The criteria for anonymization parameters settings are: (*i*) we follow the same/similar settings as in the original works of these anonymization schemes; and (*ii*) many data utilities can be preserved after anonymization. For the three graph utilities IM, SR, and SD, we only test them on small graphs, and put the results in [1]. We analyze the results in Table 5 as follows.

Generally, the evaluation results in Table 5 are consistent with our analysis in Table 2. Most anonymization algorithms can partially or conditionally preserve most graph and application utilities. Therefore, most of the anonymized data can be employed for graph analytics, data mining tasks, and graph applications.

Among all the graph utilities, JD and GCC are the most sensitive utilities to a graph's structure change, and thus they are the easiest ones to be destroyed by the anonymization algorithms. This is because these two utilities are very sensitive to edge changes. Even if the degree distribution of the anonymized data remains the same as the original data, the JD distribution and GCC may change significantly.

Compared to application utility, existing anonymization algorithms are better at preserving graph utility. For instance, most algorithms lost the RX utility and CD utility. This is because most application utilities depend on several graph utilities, e.g., the role of a user in RX depends on that user's degree, CC, BC, community attributes, and other structural characteristics. Therefore, application utilities are more easily affected than graph utilities, i.e., application utilities are more sensitive to graph's structural changes.

No anonymization scheme is optimal in preserving every data utility. For instance, *Switch* is better than *k*-DA on preserving Deg. and JD while it is worse than *k*-DA on preserving GCC and MINS, and DP is better than RW on preserving LCC and GCC while it is worse than RW on preserving Deg. Therefore, when choosing an anonymization algorithm, it is better to take into account the specific application. Furthermore, RW has the most utility loss, e.g., GCC, RX, MINS, and CD, which is also consistent with our analysis in Table 2. This is because that the graph's global structure is significantly changed in RW by replacing random walk paths with edges.

5.3.3 DA Evaluation

In this subsection, we evaluate the performance of modern DA attacks. As we analyzed before, BDK [26],

Table 5: Utility analysis of anonymization techniques. k is the number of modified edges for *Switch*, and the anonymization parameter for k-DA and Cluster, ε is the anonymization parameter for DP, t is the random walk step for RW, m is the number of edges in the original graph, and $\mathbb{D}$ is the diameter of the original graph ($\mathbb{D} = 11$ for Enron and $\mathbb{D} = 6$ for Facebook).

	Enron										**Facebook**									
Utility	*Switch* (vs. k)		k-DA (vs. k)		Cluster (vs. k)		DP (vs. ε)		RW (vs. t)		*Switch* (vs. k)		k-DA (vs. k)		Cluster (vs. k)		DP (vs. ε)		RW (vs. t)	
	.05m	.1m	5	50	5	50	300	50	2	$\mathbb{D}$	.05m	.1m	5	50	5	50	300	50	2	$\mathbb{D}$
Deg.	1	1	.9988	.9166	.9990	.9934	.9617	.8616	.9871	.9964	1	1	.9990	.9595	.9998	.9981	.9932	.9716	.9958	.9959
JD	.8725	.8338	.8928	.4183	.8216	.7055	.8496	.7363	.6972	.6438	.9941	.9804	.9947	.7328	.9872	.9024	.9755	.8263	.9678	.9362
ED	.9881	.9617	1.080	.9561	1.04	1.02	1.03	.9627	1.02	.9025	.9161	.8328	.9350	1.015	.9957	.9956	.9414	.9313	.9285	.8376
PL	.9954	.9887	.9891	.8934	.9994	.9905	.9565	.9839	.9963	.9657	.9618	.9159	.9999	.9946	.9999	1	.9960	.9653	.9706	.8965
LCC	.9830	.9631	.9972	.9809	.9966	.9797	.9528	.8328	.6785	.5985	.9204	.8303	.9998	.9983	.9968	.9947	.9793	.9437	.6239	.5543
GCC	.8967	.8013	.9921	.9283	.9774	.9097	.7755	.4609	.3107	.5383	.5180	.2241	.9847	.9986	.9766	.9937	.9522	.8702	.2552	.0334
CC	.9986	.9965	.9985	.9955	.9999	.9947	.9759	.9666	.9885	.9994	1	.9999	1	1	1	1	1	.9998	1	.9998
BC	.9859	.9812	.9691	.9019	.9936	.9733	.8360	.7406	.9613	.9246	.9787	.9494	.9790	.9515	.9983	.9897	.9779	.9518	.9935	.9669
EV	.9991	.9977	.9910	.8998	.9947	.9720	.9232	.8653	.9717	.9204	.9881	.9556	.9981	.9626	.9999	.9996	.9977	.9911	.9891	.9480
NC	.9984	.9962	.9999	.9991	.9996	.9956	.9977	.9596	.9042	.9028	.9995	.9986	1	1	1	1	.9987	.9934	.9928	.9942
NR	.9968	.9917	.9988	.9599	.9998	.9962	.9782	.8591	.9313	.8695	.9990	.9990	.9990	.9990	.9990	.9990	.9990	.9990	.9990	.9990
Infe.	.9627	.9597	.9604	.9411	.9427	.9413	.9662	.9593	.9664	.9446	.9748	.9704	.9758	.9695	.9730	.9719	.9730	.9699	.9788	.9778
PR	.9980	.9962	.9848	.8934	.9997	.9974	.9801	.9000	.8925	.9942	.9866	.9825	.9878	.9610	.9900	.9907	.9875	.9691	.9869	.9810
HS	.9991	.9977	.9910	.8998	.9947	.9720	.9232	.8653	.9717	.9204	.9326	.8780	.9711	.9789	.9648	.9625	.9626	.9322	.9283	.8655
AS	.9991	.9977	.9910	.8998	.9947	.9720	.9232	.8653	.9717	.9204	.9920	.9656	.9946	.9498	.9978	.9986	.9970	.9965	.9943	.9594
RX	.6575	.6009	.4561	.3173	.4512	.3685	.4196	.4116	.2955	.2680	.3494	.2608	.2974	.3139	.3902	.4652	.3483	.3134	.3250	.2772
RE	.9997	.9997	.9999	.9954	.9999	.9996	.9994	.9985	.9994	.9990	.9999	.9997	1	.9999	1	1	1	.9996	.9999	.9997
MINS	.7578	.6486	.9639	.9026	.9898	.9297	.7292	.3272	.1815	.1645	.6085	.4419	.9426	.9251	.9240	.9184	.8483	.7768	.2480	.1893
CD	.6251	.5411	.8454	.5339	.6794	.6692	.5095	.1028	.2531	.0569	.3536	.1986	.5043	.5887	.8558	.8523	.5027	.3213	.2860	.1205

RST [5], and RSM [5] are not scalable/practical; NSR [21] and DeA [25] are simplified versions of NS [2] and ADA [25], respectively; and NKA [22] actually depends on other attacks, e.g., NS. Therefore, here, we focus on evaluating the seven general, practical, and scalable DA attacks: NS [2], DV (we replace its seed identification phase with a scalable one) [5], PFG [23], YG [27], ADA [25], KL [24], and JLSB [3]. Furthermore, PFG and JLSB are seed-free and the other five attacks are seed-based.

First, employing the same Enron and Facebook datasets as before, we evaluate the DA performance of the seven DA attacks. The evaluation methodology is generally the same as in previous works [2, 3, 5, 22, 23, 25, 27]: we first randomly sample two graphs with probability s from the original data as the anonymized graph and auxiliary graph respectively, and then employ the auxiliary graph to de-anonymize the anonymized graph. Furthermore, for seed-based attacks, e.g., NS, DV, YG, ADA, and KL, we feed them 50 pre-identified seed mappings. The DA performance of the evaluated attacks with respect to different s is shown in Table 6. From Table 6, we have the following observations.

With the increase of s, more users can be successfully de-anonymized under each algorithm. The reason is evident. Since a large s implies that the anonymized graph and the auxiliary graph are more structurally similar, more accurate structural information can be employed by all the SDA algorithms. Hence, better DA performance can be achieved.

Generally, all the algorithms have their advantages in some specific scenarios, and no algorithm is the best in all the cases. For instance, to de-anonymize Enron, KL has the best performance when $s = .6$ while ADA has the best performance when $s = .95$. Multiple reasons are responsible for the results such as the similarity between the anonymized and auxiliary graphs, the density of the anonymized/auxiliary graph, the heuristics employed by an algorithm, etc.

According to the results, NS is more suitable for the scenarios where the anonymized and auxiliary graphs are highly similar while unsuitable when they are not sufficiently similar, e.g., it can successfully de-anonymize 95.27% Facebook users when $s = .95$ while only 0.18% users when $s = .6$. The reason is because NS mainly employs local graph structural properties to adaptively conduct user DA, and thus is sensitive to users' local structural characteristics. When s is small, most users are indistinguishable with respect to their local structures, e.g., degree, followed by poor DA performance.

Compared to NS, the other attacks, especially DV, PFG, ADA, and JLSB, are more stable even with a small s. For instance, when $s = .6$, DV, PFG, ADA, and JLSB can successfully de-anonymize 15.63%, 10.87%, 15.68%, and 14.73% Facebook users, respectively. This is because these attacks mainly employ global graph characteristics (e.g., clossness centrality, the distance vector to seeds) to perform the DA, which are more resilient to noise.

Table 6: Performance of DA attacks. s is the probability of generating the auxiliary and anonyized graphs from the original graph. Each value, e.g., 0.1277, in the table indicates the ratio of successfully de-anonymized users.

s	**De-anonymize Enron**							**De-anonymize Facebook**						
	NS	DV	PFG	YG	ADA	KL	JLSB	NS	DV	PFG	YG	ADA	KL	JLSB
.60	.0037	.1277	.0739	.0310	.1305	.1596	.1191	.0018	.1563	.1087	.2832	.1568	.0599	.1473
.65	.0039	.1601	.0937	.0410	.1651	.1814	.1460	.0020	.1998	.1402	.3346	.2005	.0747	.1799
.70	.0054	.1969	.1397	.0725	.2013	.2026	.1723	.0031	.2437	.1523	.4124	.2444	.0841	.2094
.75	.0055	.2244	.1349	.1004	.2307	.2152	.1958	.8712	.3068	.2041	.4554	.3078	.1196	.2574
.80	.0061	.2841	.1837	.1014	.2896	.2519	.2474	.9056	.3802	.2586	.4970	.3805	.1508	.3042
.85	.3420	.3481	.2180	.1531	.3522	.3123	.2971	.9231	.4561	.3073	.5402	.4576	.1817	.3559
.90	.3660	.4004	.2736	.1885	.4043	.3389	.3443	.9414	.5659	.3977	.5737	.5670	.2552	.4289
.95	.3937	.5814	.4370	.2277	.5898	.5209	.5438	.9527	.7407	.5584	.6071	.7422	.3989	.5542

For the seed-free attacks, PFG and JLSB, they can achieve comparable performance as seed-based attacks in most scenarios even without any seed information. For instance, when $s = .95$, PFG and JLSB can de-anonymize 43.7% and 54.38% Enron users, respectively, which are better than several seed-based algorithms and further demonstrate the power of structure-based attacks. The reason for the effectiveness of seed-free attacks is that in most cases, the combination of a user's local and global structural characteristics, e.g., degree, neighborhood degree distribution, closeness/betweenness centrality, is sufficient to distinguish him/her from other users.

5.3.4 Robustness of Modern SDA Attacks

The robustness of modern DA attacks with respect to graph noise (e.g., adding fake edges and deleting true edges) has been extensively evaluated in existing works [2,3,5,25]. However, to the best our knowledge, no existing work has evaluated the robustness of any seed-based de-anonymizaton attack to incorrect seed mappings. Employing Enron and Facebook, we address this open issue by conducting such an evaluation and the results are shown in Table 7. We analyze the results in Table 7 as follows.

Generally, all the DA algorithms are robust with respect to incorrect seed mappings in most scenarios. This is because during the DA process, most algorithms also employ other seed-independent structural properties, e.g., degree, closeness/betweenness centrality, in addition to relying on seed-dependent structural properties. Even for the pure seed-based DA attacks, e.g., YG and KL, they perform DA in the decreasing order of user degrees. Therefore, the negative impacts of incorrect seed mappings can be partially offset, i.e., even with some incorrect seed mappings, many users are still distinguishable with respect to their structural characteristics.

For all algorithms, when incorrect seed mappings increase, fewer users can be correctly de-anonymized. The reason is evident: more incorrect seed mappings imply more incorrect seed-dependent structural information is provided to each algorithm, followed by the degradation of the DA performance of each algorithm.

When de-anonymizing Enron, the performance of NS has a significant drop when the percentage of incorrect seed mappings is increased from 8% to 10%. This is because of the seed transitional phenomena as observed in [2], i.e., when the correct effective seed-dependent structural information is below/above some crucial threshold, NS's performance has a significant transition.

DV is much more stable than other algorithms. This is because it is a pure global structure-based attack and thus incorrect seed mappings have minimum impact on it.

5.3.5 Anonymization vs DA

Now, we evaluate the effectiveness of state-of-the-art anonymization techniques against modern DA attacks employing Enron and Facebook. The methodology is that we first employ different anonymization techniques to anonymize Enron/Facebook. Then, we sample an auxiliary graph from Enron/Facebook with probability s. Finally, we employ different DA algorithms to de-anonymize the anonymized data using the auxiliary graph. We show the results in Table 8 and analyze the results as follows.

All the state-of-the-art graph anonymization algorithms are vulnerable to some or all of the modern SDA attacks, which confirmed our analytical results in Table 4. For instance, when $s = .85$, NS can still successfully de-anonymize more than 80% Facebook users anonymized by *Switch*, k-DA, Cluster, or DP, and DV can successfully de-anonymize 15.3% Facebook users anonymized by RW ($t = 2$). Similarly, when $s = .85$, NS can successfully de-anonymize more than 35% Enron users anonymized by k-DA ($k = 5$), Cluster ($k = 5, 50$), YG can successfully de-anonymize 13.73% and 15.49% Enron users anonymized by *Switch* ($k = .05m$) and DP ($\varepsilon = 300$) respectively, and DV can successfully de-anonymize 19.23%/24.12% Enron users anonymized by RW with $t = 2/11$. Based on the results, we conclude that modern SDA attacks are very powerful. As

Table 7: DA robustness with respect to seed errors. Each algorithm is provided with 50 seed mappings, and Λ_e/Λ indicates the percentages of incorrect seed mappings. Each value in the table indicates the ratio of successfully de-anonymized users.

$\frac{\Lambda_e}{\Lambda}$	De-anonymize Enron					De-anonymize Facebook				
	NS	DV	YG	ADA	KL	NS	DV	YG	ADA	KL
4%	.341	.342	.148	.336	.302	.922	.456	.537	.442	.183
6%	.341	.342	.133	.329	.303	.917	.456	.528	.440	.183
8%	.338	.348	.135	.329	.310	.918	.456	.542	.428	.184
10%	.007	.348	.147	.323	.310	.918	.456	.536	.420	.182
12%	.007	.348	.142	.313	.311	.915	.456	.529	.414	.185
14%	.006	.348	.112	.306	.307	.916	.456	.526	.403	.186
16%	.006	.348	.129	.297	.303	.916	.456	.525	.394	.184
18%	.006	.348	.099	.293	.308	.913	.456	.533	.380	.183
20%	.006	.348	.126	.285	.306	.913	.456	.518	.356	.179
22%	.005	.348	.125	.280	.303	.912	.456	.531	.347	.182
24%	.005	.348	.116	.268	.304	.910	.456	.521	.332	.180
26%	.005	.348	.118	.255	.303	.889	.456	.528	.319	.179
28%	.004	.348	.112	.253	.300	.886	.456	.520	.309	.182
30%	.004	.348	.120	.247	.307	.884	.456	.522	.283	.180
32%	.004	.348	.106	.235	.305	.888	.456	.521	.270	.178
34%	.004	.348	.081	.230	.304	.887	.456	.521	.259	.178
36%	.004	.348	.084	.216	.300	.889	.456	.505	.245	.182
38%	.004	.347	.096	.199	.301	.888	.456	.493	.230	.178
40%	.004	.347	.065	.186	.302	.886	.456	.505	.214	.179
42%	.003	.347	.071	.182	.302	.882	.456	.516	.195	.181
44%	.003	.347	.106	.169	.303	.881	.456	.495	.185	.180
46%	.003	.347	.050	.160	.299	.881	.456	.480	.173	.177
48%	.003	.347	.059	.153	.297	.881	.456	.497	.161	.180
50%	.002	.347	.063	.146	.298	.874	.456	.475	.148	.176

Table 8: Anonymization vs DA. The seed-based algorithms are provided with 50 seeds and the anonymization parameters are chosen according to the same criteria as in Table 5.

	s	Enron										Facebook									
		Switch (k)		k-DA (k)		Cluster (k)		DP (ε)		RW (t)		*Switch* (k)		k-DA (k)		Cluster (k)		DP (ε)		RW (t)	
		5	10	5	50	5	50	300	50	2	$\mathbb{D}$	5	10	5	50	5	50	300	50	2	$\mathbb{D}$
NS	.85	.0072	.0052	.3702	.0088	.3722	.3707	.0091	.0055	.0015	.0015	.8973	.8247	.9454	.9402	.9456	.9442	.9317	.8914	.0008	.0006
	.90	.0077	.0054	.3822	.0105	.3900	.3839	.0095	.0060	.0015	.0015	.9063	.8427	.9520	.9495	.9519	.9508	.9393	.8944	.0008	.0007
	.95	.3577	.0064	.4033	.0418	.4049	.4064	.3946	.0064	.0015	.0016	.9162	.8583	.9570	.9559	.9569	.9558	.9453	.9130	.0000	.0007
DV	.85	.1261	.0813	.1433	.0437	.2120	.1408	.1160	.0701	.1923	.2412	.1716	.0926	.2411	.0588	.3340	.3368	.2324	.0736	.1530	.1271
	.90	.1546	.0956	.1765	.0517	.2564	.1637	.1394	.0733	.2129	.2169	.2124	.1147	.2999	.0758	.4113	.4090	.3623	.0802	.1604	.1322
	.95	.2121	.1366	.2548	.0753	.3745	.2215	.1821	.0858	.2072	.2190	.3006	.1586	.4210	.1161	.5767	.5656	.4087	.1016	.1591	.1332
PFG	.85	.0667	.0422	.0692	.0214	.1116	.0683	.0489	.0365	.1578	.2131	.0706	.0395	.0703	.0154	.1191	.1155	.0891	.0206	.1349	.1190
	.90	.0805	.0478	.0810	.0263	.1317	.0789	.0571	.0390	.1711	.2012	.0978	.0497	.0946	.0213	.1480	.1595	.1870	.0223	.1382	.1217
	.95	.1193	.0695	.1123	.0353	.1978	.0952	.0755	.0479	.1714	.2074	.1378	.0725	.1317	.0332	.2034	.2330	.1756	.0295	.1397	.1216
YG	.85	.1373	.0969	.1646	.0289	.1576	.1570	.1549	.0664	.0394	.0323	.5437	.5056	.5816	.5086	.5897	.5805	.5404	.4347	.0356	.0210
	.90	.1716	.1037	.1612	.0253	.1868	.1710	.1577	.0736	.0404	.0342	.5681	.5182	.6089	.5129	.6036	.5980	.5702	.4818	.0372	.0222
	.95	.1730	.1197	.2155	.3785	.1971	.2064	.1884	.0838	.0418	.0348	.5821	.5439	.6208	.5504	.6223	.6190	.5716	.4538	.0346	.0231
ADA	.85	.1262	.0820	.1468	.0445	.2130	.1418	.1160	.0701	.0771	.0731	.1724	.0926	.2425	.0603	.3358	.3379	.2337	.0749	.0985	.0725
	.90	.1543	.0964	.1795	.0534	.2588	.1652	.1394	.0729	.0855	.0704	.2129	.1146	.3026	.0776	.4124	.4103	.3639	.0823	.1008	.0764
	.95	.2139	.1381	.2605	.0768	.3777	.2230	.1823	.0855	.0872	.0733	.3019	.1589	.4245	.1186	.5780	.5667	.4105	.1038	.1041	.0784
KL	.85	.0904	.0811	.0997	.0357	.0965	.0689	.0745	.0331	.0900	.0729	.0799	.0764	.0819	.0683	.0788	.0762	.0769	.0313	.1099	.0737
	.90	.1077	.0970	.1202	.0549	.1134	.0918	.0874	.0319	.0939	.0744	.0979	.0939	.1013	.0848	.0960	.0863	.1249	.0317	.1099	.0715
	.95	.1381	.1150	.1936	.0978	.2052	.1686	.1719	.0376	.0994	.0776	.1350	.1331	.1418	.1265	.1294	.1206	.1450	.0600	.1171	.0754
JLSB	.85	.0692	.0440	.0798	.0234	.1248	.0854	.0886	.0656	.0709	.0720	.1453	.0786	.2025	.0595	.2618	.2673	.1958	.0768	.0901	.0681
	.90	.0886	.0536	.1046	.0296	.1618	.1135	.1070	.0664	.0767	.0728	.1673	.0911	.2335	.0708	.3001	.3094	.3050	.0777	.0911	.0699
	.95	.1846	.1189	.2381	.0746	.3317	.2319	.1449	.0814	.0838	.0740	.2180	.1174	.3111	.1096	.3983	.3924	.3142	.0950	.0940	.0734

we analyzed in Table 4, two fundamental reasons make state-of-the-art graph anonymization algorithms vulnerable. First, in existing graph anonymization schemes, graph users are only indistinguishable with respect to some structural properties/semantics. However, several other structural properties or the combinations of them can still enable effective graph user DA. Furthermore, the design philosophy of existing anonymization schemes is to preserve as much data utility as possible. However, data utility can be used to conduct powerful SDA attacks. Therefore, it is still an open problem to design effective graph data anonymization algorithms which can defend against modern SDA attacks.

Generally, when *s* is large and the anonymization level (e.g., *k* for *Switch* and *k*-DA) is low, more users can be correctly de-anonymized. The reason is straightforward. A large *s* implies more structural information of the original graph can be preserved in the auxiliary graph and thus more accurate structural characteristics can be employed for DA. Meanwhile, a low anonymization level implies less perturbation applied to the original graph's structure followed by the anonymized graph is more structurally similar to the original graph and thus is easier to be de-anonymized.

Among all the DA attacks, NS, YG, and ADA perform better than other attacks in most scenarios. This is because they mainly employ the combinations of several local structural characteristics to conduct the DA. According to our utility analysis in Table 2 and evaluation results in Table 5, most existing anonymization algorithms can preserve most graph utilities, especially the local graph utilities, e.g., Deg., LCC. It turns out that the graph utility preserved by anonymization algorithms can be used by DA attacks to conduct effective DA. Therefore, in the scenarios where an anonymization algorithm preserves more data utility, the corresponding dataset is more vulnerable to modern SDA attacks.

Among all the anonymization techniques, RW has better performance than others in most of the cases. The reason is that, a random walk path of length *t* is replaced by an edge in RW. It follows that the original graph structure is significantly changed. Therefore, a RW-anonymized graph is more resistant to DA attacks. However, RW achieves such DA resistance at the cost of sacrificing more data utility compared with other anonymization techniques, which is consistent with our utility analysis and evaluation results in Tables 2 and 5. Furthermore, we can also find that in most scenarios, existing anonymization techniques can degrade the performance of SDA attacks. Again, as shown in Tables 2 and 5, some data utilities are also degraded/lost.

6 Future Research and Challenges

In this section, we discuss the future research directions and challenges of graph data anonymization and DA.

Graph Data Anonymization. According to our analytical results in Table 4 and evaluation results in Table 8, all the state-of-the-art anonymization techniques, e.g., *k*-anonymity based schemes, DP based schemes, are vulnerable to modern SDA attacks. Their vulnerability depends on how much data utility is preserved in the anonymized data. Therefore, it is very difficult, if not impossible, to develop effective and universal graph data anonymization techniques to defend against modern SDA attacks. The main challenges are two-folds. First, guaranteeing data utility is one of the primary objectives when publishing/sharing graph data. However, as we explained before, the preserved graph and application utilities enable adversaries to conduct large-scale DA attacks. Therefore, it is a big challenge to effectively anonymize graph data with desired data utility preservation and without enabling adversaries to utilize these data utilities. Second, many local and global structural characteristics (or, structural scemantics), e.g., Deg., LCC, CC, BC, are embedded in graph data's structure. Existing anonymization techniques can only make graph users structurally indistinguishable with respect to one or several semantics, e.g., degree and neighborhood. However, as we explained before, in many scenarios, several other structural semantics and their combinations are sufficient to enable a SDA attack to de-anonymize graph users. Therefore, it is also a key challenge to make graph users structurally indistinguishable with respect to most, if not all, structural semantics.

Considering that it is difficult to seek a tradeoff between generic utility and anonymity, a promising research direction could be developing *application-oriented* anonymization techniques. Instead of preserving as much data utility as possible, one only considers some specific application-aware utility when designing the anonymization techniques. For instance, although RW loses more data utility than most existing graph anonymization techniques, it achieves better anonymity and meanwhile supports some application utility, e.g., sybil detection [20].

Graph Data DA. Based on our DA evaluation results, future DA research may follow two directions.

First, it is interesting to study how to combine the advantages of different algorithms and develop new stable and improved DA schemes. To achieve this, the challenge is to decide which structural characteristics should be employed and how to use these characteristics during the DA process. This is because some structural characteristics are local (e.g., Deg.) while others are global (e.g., CC and BC). It is better to seek a balance between the employed local and global structural semantics. Additionally, some structural characteristics may carry similar structural semantics, and thus simultaneously employing such characteristics will not lead to too much improvement. Furthermore, according to our evaluation experience, the sequence and weights of applying different structural characteristics may induce very different DA performance.

Second, instead of trying to design a uniformly optimal DA algorithm, it is better to develop some anonymization technique-oriented and application-aware DA schemes. This is because, for some anonymization algorithms, e.g., most *k*-anonymity based schemes, they mainly achieve anonymity by local graph perturbation. In this scenario, the global graph characteristics based DA algorithms will be more effective. On the other hand, for some anonymization algorithms, e.g., *Add/Del* and RW, they mainly achieve anonymity through global graph perturbation. Therefore, the local graph characteristics based DA schemes will be better at de-anonymizing the data anonymized by these techniques. Furthermore, according to our DA evaluation experience, some DA attacks are more effective to de-anonymize dense graphs, e.g., NS and JLSB, while some other attacks are more effective to de-anonymize sparse graphs, e.g., DV, PFG. Therefore, when developing new DA algorithms, it is helpful to take into account both the attacked anonymization technique and the attacked application.

More Future Work. In this paper, we focus on implementing and evaluating graph data anonymization and DA techniques. It is also interesting to integrate the anonymization and DA techniques for other data types, e.g., relational data. In the future, we propose to develop a uniform and open-source evaluation system supporting multi-type data anonymization and DA.

7 Conclusion

In this paper, we propose, implement, and evaluate SecGraph (available at [1]), an *open-source* secure graph data publishing/sharing system. Within SecGraph, we systematically analyze, implement, and evaluate 11 graph data anonymization algorithms, 19 data utility metrics, and 15 modern SDA attacks. To the best of our knowledge, SecGraph is the first such system that provides a *uniform platform* enabling data owners to anonymize and evaluate the security of their data, and simultaneously enabling researchers to conduct fair studies of existing or newly developed anonymization/DA techniques. Leveraging SecGraph, we conduct extensive experimental evaluations. The results demonstrate that (*i*) most anonymization schemes can partially or conditionally preserve most graph utility but lose some application utility; (*ii*) no DA attack is optimum in all scenarios. The actual DA performance depends on several factors; and (*iii*) all the state-of-the-art anonymization schemes are vulnerable to modern SDA attacks. Based on our findings and analysis, we discuss the future research directions and challenges of graph data anonymization and DA.

8 Acknowledgments

The authors are very grateful to the anonymous reviewers for their time and valuable comments. The authors are also grateful to the following researchers in developing SecGraph: Ada Fu, Michael Hay, Davide Proserpio, Qian Xiao, Shirin Nilizadeh, Jing S. He, Wei Chen, and Stanford SNAP developers.

This work was partly supported by NSF-CAREER-CNS-0545667. Prateek Mittal was supported in part by the NSF under the grant CNS-1409415.

References

[1] S. Ji, W. Li, P. Mittal, X. Hu, and R. Beyah. Secgraph. *http://www.ece.gatech.edu/cap/secgraph/.*

[2] A. Narayanan and V. Shmatikov. De-anonymizing social networks. *S&P*, 2009.

[3] S. Ji, W. Li, M. Srivatsa, and R. Beyah. Structural data de-anonymization: Quantification, practice, and implications. *CCS*, 2014.

[4] S. Ji, W. Li, N. Gong, P. Mittal, and R. Beyah. On your social network de-anonymizablity: Quantification and large scale evaluation with seed knowledge. *NDSS*, 2015.

[5] M. Srivatsa and M. Hicks. Deanonymizing mobility traces: Using social networks as a side-channel. *CCS*, 2012.

[6] X. Ying and X. Wu. Randomizing social networks: a spectrum preserving approach. *SDM*, 2008.

[7] B. Zhou and J. Pei. Preserving privacy in social networks against neighborhood attacks. *ICDE*, 2008.

[8] K. Liu and E. Terzi. Towards identity anonymization on graphs. *SIGMOD*, 2008.

[9] L. Zou, L. Chen, and M. T. Özsu. K-automorphism: A general framework for privacy preserving network publication. *VLDB*, 2009.

[10] J. Cheng, A. Fu, and J. Liu. K-isomorphism: Privacy preserving network publication against structural attacks. *SIGMOD*, 2010.

[11] M. Yuan, L. Chen, and P. Yu. Personalized privacy protection in social networks. *VLDB*, 2010.

[12] M. Hay, G. Miklau, D. Jensen, D. Towsley, and P. Weis. Resisting structural re-identification in anonymized social networks. *VLDB*, 2008.

[13] S. Bhagat, G. Cormode, B. Krishnamurthy, and D. Srivastava. Class-based graph anonymization for social network data. *VLDB*, 2009.

[14] B. Thompson and D. Yao. The union-split algorithm and cluster-based anonymization of social networks. *ASIACCS*, 2009.

[15] A. Sala, X. Zhao, C. Wilson, H. Zheng, and B. Zhao. Sharing graphs using differentially private graph models. *IMC*, 2011.

[16] D. Proserpio, S. Goldberg, and F. McSherry. A workflow for differentially-private graph synthesis. *WOSN*, 2012.

[17] D. Proserpio, S. Goldberg, and F. McSherry. Calibrating data to sensitivity in private data analysis. *VLDB*, 2014.

[18] Y. Wang and X. Wu. Preserving differential privacy in degree-correlation based graph generation. *TDP*, 2013.

[19] Q. Xiao, R. Chen, and K. Tan. Differentially private network data release via structural inference. *KDD*, 2014.

[20] P. Mittal, C. Papamanthou, and D. Song. Preserving link privacy in social network based systems. *NDSS 2013*.

[21] A. Narayanan, E. Shi, and B. Rubinstein. Link prediction by de-anonymization: How we won the kaggle social network challenge. *IJCNN*, 2011.

[22] S. Nilizadeh, A. Kapadia, and Y.-Y. Ahn. Community-enhanced de-anonymization of online social networks. *CCS*, 2014.

[23] P. Pedarsani, D. R. Figueiredo, and M. Grossglauser. A bayesian method for matching two similar graphs without seeds. *Allerton*, 2013.

[24] N. Korula and S. Lattanzi. An efficient reconciliation algorithm for social networks. *VLDB*, 2014.

[25] S. Ji, W. Li, M. Srivatsa, J. He, and R. Beyah. Structure based data de-anonymization of social networks and mobility traces. *ISC*, 2014.

[26] L. Backstrom, C. Dwork, and J. Kleinberg. Wherefore art thou r3579x? anonymized social networks, hidden patterns, and structural steganography. *WWW*, 2007.

[27] L. Yartseva and M. Grossglauser. On the performance of percolation graph matching. *COSN*, 2013.

[28] D. Goodin. http://arstechnica.com/tech-policy/2014/06/poorly-anonymized-logs-reveal-nyc-cab-drivers-detailed-whereabouts/.

[29] P. Samarati. Protecting respondents' identities in microdata release. *TKDE*, 2001.

[30] J. Brickell and V. Shmatikov. The cost of privacy: Destruction of data-mining utility in anonymized data publishing. *KDD*, 2008.

[31] C. Dwork. Differential privacy. *ICALP*, 2006.

[32] R. Chen, G. Acs, and C. Castelluccia. Differentially private sequential data publication via variable-length n-grams. *CCS*, 2012.

[33] R. Albert, H. Jeong, and A.-L. Barabasi. Error and attack tolerance of complex networks. *Nature*, 2000.

[34] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 1998.

[35] K. Henderson et al. Roix: Structural role extraction & mining in large graphs. *KDD*, 2012.

[36] S. Garriss, M. Kaminsky, M. J. Freedman, B. Karp, D. Mazières, and H. Yu. Re: Reliable email. *NSDI*, 2006.

[37] W. Chen, Y. Wang, and ang S. Yang. Efficient influence maximization in social networks. *KDD*, 2009.

[38] J. He, S. Ji, R. Beyah, and Z. Cai. Minimum-sized influential node set selection for social networks under the independent cascade model. *Mobihoc*, 2014.

[39] J Yang and J. Leskovec. Overlapping community detection at scale: A nonnegative matrix factorization. *WSDM*, 2013.

[40] S. Marti, P. Ganesan, and H. Garcia-Molina. Sprout: P2p routing with social networks. *P2P&DB*, 2004.

[41] H. Yu, P. B. Gibbons, M. Kaminsky, and F. Xiao. Sybillimit: A near-optimal social network defense against sybil attacks. *S&P*, 2008.

Trustworthy Whole-System Provenance for the Linux Kernel

Adam Bates, Dave (Jing) Tian,
Kevin R.B. Butler
University of Florida
{adammbates,daveti,butler}@ufl.edu

Thomas Moyer
MIT Lincoln Laboratory
thomas.moyer@ll.mit.edu

Abstract

In a provenance-aware system, mechanisms gather and report metadata that describes the history of each object being processed on the system, allowing users to understand how data objects came to exist in their present state. However, while past work has demonstrated the usefulness of provenance, less attention has been given to *securing* provenance-aware systems. Provenance itself is a ripe attack vector, and its authenticity and integrity must be guaranteed before it can be put to use.

We present Linux Provenance Modules (LPM), the first general framework for the development of provenance-aware systems. We demonstrate that LPM creates a trusted provenance-aware execution environment, collecting complete whole-system provenance while imposing as little as 2.7% performance overhead on normal system operation. LPM introduces new mechanisms for secure provenance layering and authenticated communication between provenance-aware hosts, and also interoperates with existing mechanisms to provide strong security assurances. To demonstrate the potential uses of LPM, we design a Provenance-Based Data Loss Prevention (PB-DLP) system. We implement PB-DLP as a file transfer application that blocks the transmission of files derived from sensitive ancestors while imposing just tens of milliseconds overhead. LPM is the first step towards widespread deployment of trustworthy provenance-aware applications.

1 Introduction

A provenance-aware system automatically gathers and reports metadata that describes the history of each object being processed on the system. This allows users to track, and understand, how a piece of data came to exist in its current state. The application of provenance is presently of enormous interest in a variety of disparate communities including scientific data processing, databases, software development, and storage [43, 53]. Provenance has also been demonstrated to be of great value to security by identifying malicious activity in data centers [5, 27, 56, 65, 66], improving Mandatory Access Control (MAC) labels [45, 46, 47], and assuring regulatory compliance [3].

Unfortunately, most provenance collection mechanisms in the literature exist as fully-trusted user space applications [28, 27, 41, 56]. Even kernel-based provenance mechanisms [43, 48] and sketches for trusted provenance architectures [40, 42] fall short of providing a provenance-aware system for malicious environments. The problem of whether or not to trust provenance is further exacerbated in distributed environments, or in layered provenance systems, due to the lack of a mechanism to verify the authenticity and integrity of provenance collected from different sources.

In this work, we present **Linux Provenance Modules (LPM)**, the first generalized framework for secure provenance collection on the Linux operating system. Modules capture *whole-system provenance*, a detailed record of processes, IPC mechanisms, network activity, and even the kernel itself; this capture is invisible to the applications for which provenance is being collected. LPM introduces a gateway that permits the upgrading of low integrity workflow provenance from user space. LPM also facilitates secure distributed provenance through an authenticated, tamper-evident channel for the transmission of provenance metadata between hosts. LPM interoperates with existing security mechanisms to establish a hardware-based root of trust to protect system integrity.

Achieving the goal of trustworthy whole-system provenance, we demonstrate the power of our approach by presenting a scheme for *Provenance-Based Data Loss Prevention (PB-DLP)*. PB-DLP allows administrators to reason about the propagation of sensitive data and control its further dissemination through an expressive policy system, offering dramatically stronger assurances than existing enterprise solutions, while imposing just mil-

The Lincoln Laboratory portion of this work was sponsored by the Assistant Secretary of Defense for Research & Engineering under Air Force Contract #FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

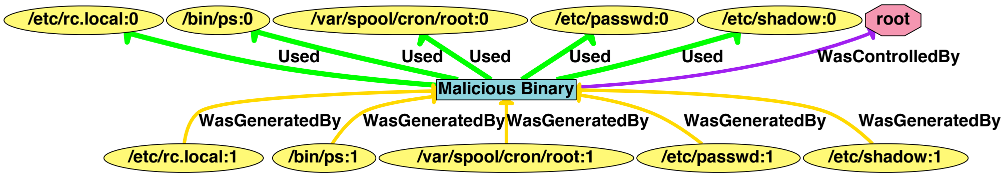


Figure 1: A provenance graph showing the attack footprint of a malicious binary. Edges encode relationships that flow backwards into the history of system execution, and writing to an object creates a second node with an incremented version number. Here, we see that the binary has rewritten `/etc/rc.local`, likely in an attempt to gain persistence after a system reboot.

liseconds of overhead on file transmission. To our knowledge, this work is the first to apply provenance to DLP.

Our contributions can thus be summarized as follows:

- **Introduce Linux Provenance Modules (LPM).** LPM facilitates secure provenance collection at the kernel layer, supports *attested disclosure* at the application layer, provides an *authenticated channel* for network transmission, and is compatible with the W3C Provenance (PROV) Model [59]. In evaluation, we demonstrate that provenance collection imposes as little as 2.7% performance overhead.

- **Demonstrate secure deployment.** Leveraging LPM and existing security mechanisms, we create a trusted provenance-aware execution environment for Linux. Through porting Hi-Fi [48] and providing support for SPADE [29], we demonstrate the relative ease with which LPM can be used to secure existing provenance collection mechanisms. We show that, in realistic malicious environments, ours is the first proposed system to offer secure provenance collection.

- **Introduce Provenance-Based Data Loss Prevention (PB-DLP).** We present a new paradigm for the prevention of data leakage that searches object provenance to identify and prevent the spread of sensitive data. PB-DLP is impervious to attempts to launder data through intermediary files and IPC. We implement PB-DLP as a file transfer application, and demonstrate its ability to query object ancestries in just tens of milliseconds.

2 Background

Data provenance, sometimes called *lineage*, describes the actions taken on a data object from its creation up to the present. Provenance can be used to answer a variety of historical questions about the data it describes. Such questions include, but are not limited to, "*What processes and datasets were used to generate this data?*" and "*In what environment was the data produced?*" Conversely, provenance can also answer questions about the successors of a piece of data, such as "*What objects on the system were derived from this object?*" Although potential applications for such information are nearly limitless, past proposals have conceptualized provenance in different ways, indicating that a one-size-fits-all solution to provenance collection is unlikely to meet the needs of all of these audiences. We review these past proposals for provenance-aware systems in Section 8.

The commonly accepted representation for data provenance is a directed acyclic graph (DAG). In this work, we use the W3C PROV-DM specification [59] because it is pervasive and facilitates the exchange of provenance between deployments. An example PROV-DM graph of a malicious binary is shown in Figure 1. This graph describes an attack in which a binary running with root privilege reads several sensitive system files, then edits those files in an attempt to gain persistent access to the host. Edges encode relationships between nodes, pointing backwards into the history of system execution. Writing to an object triggers the creation of a second object node with an incremented version number. This particular provenance graph could serve as a valuable forensics tool, allowing system administrators to better understand the nature of a network intrusion.

2.1 Data Loss Prevention

Data Loss Prevention (DLP) is enterprise software that seeks to minimize the leakage of sensitive data by monitoring and controlling information flow in large, complex organizations [1].[1] In addition to the desire to control intellectual property, another motivator for DLP systems is demonstrating regulatory compliance for personally-identifiable information (PII),[2] as well as directives such

[1] Our overview of data loss prevention is based on review of publicly available product descriptions for software developed by Bit9, CDW, Cisco, McAfee, Symantec, and Titus.

[2] See NIST SP 800-122

as PCI,[3] HIPAA,[4] SOX.[5] or E.U. Data Protection.[6] As encryption can be used to protect data at rest from unauthorized access, the true DLP challenge involves preventing leakage at the hands of authorized users, both malicious and well-meaning agents. This latter group is a surprisingly big problem in the fight to control an organization's intellectual property; a 2013 study conducted by the Ponemon Institute found that over half of companies' employees admitted to emailing intellectual property to their personal email accounts, with 41 percent admitting to doing so on a weekly basis [2]. It is therefore important for a DLP system to be able to exhaustively explain which pieces of data are sensitive, where that data has propagated to within the organization, and where it is (and is not) permitted to flow.

As DLP systems are proprietary and are marketed so as to abstract away the complex details of their internal workings, we cannot offer a complete explanation of their core features. However, some of the mechanisms in such systems are known. Many DLP products use a *regular expression-based* approach to identify sensitive data, operating similarly to a general-purpose version of Cornell's Spider[7]. For example, in PCI compliance,[3] DLP might attempt to identify credit card numbers in outbound emails by searching for 16 digit numbers that pass a Mod-10 validation check [39]. Other DLP systems use a *label-based* approach to identify sensitive data, tagging document metadata with security labels. The Titus system accomplishes this by having company employees manually annotate the documents that they create;[8] plugins for applications (e.g., Microsoft Office) then prevent the document from being transmitted to or opened by other employees that lack the necessary clearance. In either approach, DLP software is difficult to configure and prone to failure, offering marginal utility at great price.

3 Linux Provenance Modules

To serve as the foundation for secure provenance-aware systems, we present *Linux Provenance Modules (LPM)*. We provide a working definition for the provenance our system will collect in §3.1. In §3.2 we consider the capabilities and aims of a provenance-aware adversary, and identify security and design goals in §3.3. The LPM design is presented in §3.4, and in §3.5 we demonstrate its secure deployment. An expanded description of our system is available in our technical report [8].

3.1 Defining Whole-System Provenance

In the design of LPM, we adopt a model for *whole-system provenance*[9] that is broad enough to accommodate the needs of a variety of existing provenance projects. To arrive at a definition, we inspect four past proposals that collect broadly scoped provenance: SPADE [29], LineageFS [53], PASS [43], and Hi-Fi [48]. **SPADE** provenance is structured around primitive operations of system activities with data inputs and outputs. It instruments file and process system calls, and associates each call to a process ID (PID), user identifier, and network address. **LineageFS** uses a similar definition, associating process IDs with the file descriptors that the process reads and writes. **PASS** associates a process's output with references to all input files and the command line and process environment of the process; it also appends out-of-band knowledge such as OS and hardware descriptions, and random number generator seeds, if provided. In each of these systems, networking and IPC activity is primarily reflected in the provenance record through manipulation of the underlying file descriptors. **Hi-Fi** takes an even broader approach to provenance, treating non-persistent objects such as memory, IPC, and network packets as principal objects.

We observe that, in all instances, provenance-aware systems are exclusively concerned with operations on *controlled data types*, which are identified by Zhang et al. as files, inodes, superblocks, socket buffers, IPC messages, IPC message queue, semaphores, and shared memory [64]. Because controlled data types represent a super set of the objects tracked by system layer provenance mechanisms, we define whole-system provenance as *a complete description of agents (users, groups) controlling activities (processes) interacting with controlled data types during system execution.*

3.2 Threat Model & Assumptions

We consider an adversary that has gained remote access to a provenance-aware host or network. Once inside the system, the attacker may attempt to remove provenance records, insert spurious information into those records, or find gaps in the provenance monitor's ability to record information flows. A network attacker may also attempt to forge or strip provenance from data in transit. Because captured provenance can be put to use in other applications, the adversary's goal may even be to target the provenance monitor itself. The implications and methods of such an attack are domain-specific. For example:

[3] See https://www.pcisecuritystandards.org

[4] See http://www.hhs.gov/ocr/privacy

[5] Short for the Sarbanes-Oxley Act, U.S. Public Law No. 107-20

[6] See EU Directive 95/46/EC

[7] See http://www2.cit.cornell.edu/security/tools

[8] See http://www.titus.com

[9] This term is coined in [48], but not explicitly defined. We demonstrate the concrete requirements of a collection mechanism for whole-system provenance in this work.

- **Scientific Computing**: An adversary may wish to manipulate provenance in order to commit fraud, or to inject uncertainty into records to trigger a "Climategate"-like controversy [50].

- **Access Control**: When used to mediate access decisions [7, 45, 46, 47], an attacker could tamper with provenance in order to gain unauthorized access, or to perform a denial-of-service attack on other users by artificially escalating the security level of data objects.

- **Networks**: Provenance metadata can also be associated with packets in order to better understand network events in distributed systems [5, 65, 66]. Coordinating multiple compromised hosts, an attacker may attempt to send *unauthenticated* messages to avoid provenance generation and to perform data exfiltration.

We define a provenance trusted computing base (TCB) to be the kernel mechanisms, provenance recorder, and storage back-ends responsible for the collection and management of provenance. *Provenance-aware applications are not considered part of the TCB.*

We make the following assumption with regards to the TCB. In Linux, kernel modules have unrestricted access to kernel memory, meaning that there is no mechanism for protecting LPM from the rest of the kernel. The kernel code is therefore trusted; we assume that the stock kernel will not seek to tamper with the TCB. However, we do consider the possibility that the kernel could be compromised after installation through its interactions with user space applications. To facilitate host attestation in distributed environments, we also assume access to a Public Key Infrastructure (PKI) for provenance-aware hosts to publish their public signing keys.

3.3 System Goals

We set out to provide the following security assurances in the design of of our system-layer provenance collection mechanism. McDaniel et al. liken the needs of a secure provenance monitor [42] to the reference monitor guarantees laid out by Anderson [4]: complete mediation, tamperproofness, and verifiability. We define these guarantees as follows:

G1 **Complete**. Complete mediation for provenance has been discussed elsewhere in the literature in terms of assuring *completeness* [32]: that the provenance record be gapless in its description of system activity. To facilitate this, LPM must be able to observe all information flows that pass through controlled data types.

G2 **Tamperproof.** As many provenance use cases involve enhancing system security, LPM will be an adversarial target. The TCB must therefore be impervious to disabling or manipulation by processes in user space.

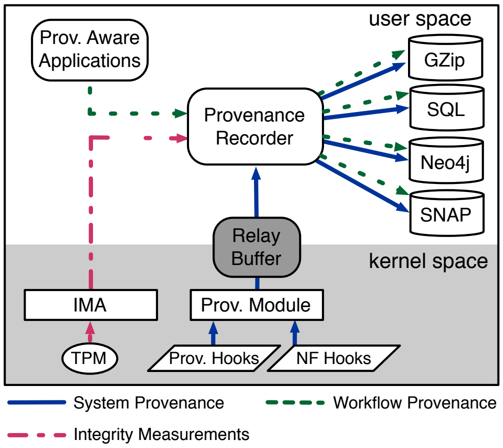


Figure 2: Diagram of the LPM Framework. Kernel hooks report provenance to a recorder in userspace, which uses one of several storage back-ends. The recorder is also responsible for evaluating the integrity of workflow provenance prior to storing it.

G3 **Verifiable**. The functionality of LPM must be verifiably correct. Additionally, local and remote users should be able to attest whether the host with which they are communicating is running the secured provenance-aware kernel.

Through surveying past work in provenance-aware systems, we identify the following additional goals to support whole-system provenance:

G4 **Authenticated Channel**. In distributed environments, provenance-aware systems must provide a means of assuring authenticity and integrity of provenance as it is communicated over open networks [7, 42, 48, 65]. *While we do not seek to provide a complete distributed provenance solution in LPM*, we do wish to provide the required building blocks within the host for such a system to exist. LPM must therefore be able to monitor every network message that is sent or received by the host, and reliably explain these messages to other provenance-aware hosts in the network.

G5 **Attested Disclosure**. Layered provenance, where additional metadata is disclosed from higher operational layers, is a desirable feature in provenance-aware systems, as applications are able to report workflow semantics that are invisible to the operating system [44]. LPM must provide a gateway for upgrading low integrity user space disclosures before logging them in the high integrity provenance record. This is consistent with the Clark-Wilson Integrity model for upgrading or discarding low integrity inputs [17].

In order to bootstrap trust in our system, we have implemented LPM as a parallel framework to Linux Security Modules (LSM) [60, 61]. Building on these results, we show in Section 4 that this approach allows LPM to inherit the formal assurances that have been verified for the LSM architecture.

3.4 Design & Implementation

An overview of the LPM architecture is shown in Figure 2. The LPM patch places a set of *provenance hooks* around the kernel; a *provenance module* then registers to control these hooks, and also registers several Netfilter hooks; the module then observes system events and transmits information via a relay buffer to a *provenance recorder* in user space that interfaces with a datastore. The recorder also accepts disclosed provenance from applications after verifying their correctness using the Integrity Measurements Architecture (IMA) [52].

In designing LPM, we first considered using an experimental patch to the LSM framework that allows “stacking” of LSM modules [10]. However, at this time, no standard exists for handling when modules make conflicting decisions, creating the potential unpredicted behavior. We also felt that dedicated provenance hooks were necessary; by collecting provenance *after* LSM authorization routines, we ensure that the provenance history is an accurate description of authorized system events. If provenance collection occurred *during* authorization, as would be the case with stacked LSMs, it would not be possible to provide this property.

3.4.1 Provenance Hooks

The LPM patch introduces a set of hook functions in the Linux kernel. These hooks behave similarly to the LSM framework’s security hooks in that they facilitate modularity, and default to taking no action unless a module is enabled. Each provenance hook is placed directly beneath a corresponding security hook. The return value of the security hook is checked prior to calling the provenance hook, thus assuring that the requested activity has been authorized prior to provenance capture; we consider the implications of this design in Section 4. A workflow for the hook architecture is depicted in Figure 3. The LPM patch places over 170 provenance hooks, one for each of the LSM authorization hooks. In addition to the hooks that correspond to existing security hooks, we also support Pohly et al.’s Hi-Fi [48] hook that is necessary to preserve Lamport timestamps on network messages [38].

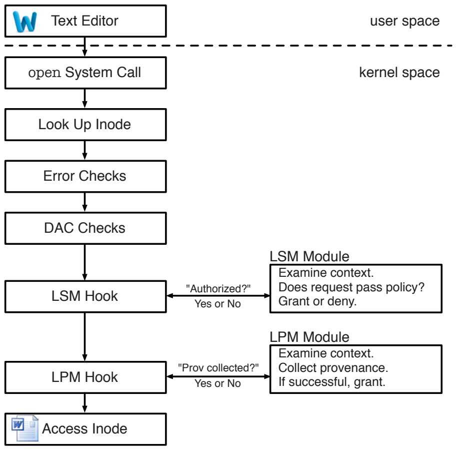


Figure 3: Hook Architecture for the `open` system call. Provenance is collected *after* DAC and LSM checks, ensuring that it accurately reflects system activity. LPM will only deny the operation if it fails to generate provenance for the event.

3.4.2 Netfilter Hooks

LPM uses Netfilter hooks to implement a cryptographic message commitment protocol. In Hi-Fi, provenance-aware hosts communicated by embedding a provenance sequence number in the IP options field [49] of each outbound packet [48]. This approach allowed Hi-Fi to communicate as normal with hosts that were not provenance-aware, but unfortunately was not secure against a network adversary. In LPM, provenance sequence numbers are replaced with Digital Signature Algorithm (DSA) signatures, which are space-efficient enough to embed in the IP Options field. We have implemented full DSA support in the Linux kernel by creating signing routines to use with the existing DSA verification function. DSA signing and verification occurs in the NetFilter `inet_local_out` and `inet_local_in` hooks. In `inet_local_out`, LPM signs over the immutable fields of the IP header, as well as the IP payload. In `inet_local_in`, LPM checks for the presence of a signature, then verifies the signature against a configurable list of public keys. If the signature fails, the packet is dropped before it reaches the recipient application, thus ensuring that there are no breaks in the continuity of the provenance log. The key store for provenance-aware hosts is obtained by a PKI and transmitted to the kernel during the boot process by writing to `securityfs`. LPM registers the Netfilter hooks with the highest prior-

[10] See https://lwn.net/Articles/518345/

ity levels, such that signing occurs just before transmission (i.e., after all other IPTables operations), and signature verification occurs just after the packet enters the interface (i.e., before all other IPTables operations).

3.4.3 Provenance Modules

Here, we introduce two of our own provenance modules (Provmon, SPADE), as well as briefly mention the work of our peers (UPTEMPO):

- **Provmon**. Provmon is an extended port of the Hi-Fi security module [48]. The original Hi-Fi code base was 1,566 lines of code, requiring 723 lines to be modified in the transition. Our extensions introduced 728 additional lines of code. The process of porting did not affect the module's functionality, although we have subsequently extended the Hi-Fi protocol to capture additional lineage information:

  File Versioning. The original Hi-Fi protocol did not track version information for files, leading to uncertainty as to the exact contents of a file at the time it was read. Accurately recovering this information in user space was not possible due to race conditions between kernel events. Because versioning is necessary to break cycles in provenance graphs [43], we have added a version field to the provenance context for inodes, which is incremented on each write.

  Network Context. Hi-Fi omitted remote host address information for network events, reasoning that source information could be forged by a dishonest agent in the network. These human-interpretable data points were replaced with an assigned random identifier for each packet. We found, however, that these identifiers could not be interpreted without remote address information, and incorporated the recording of remote IP addresses and ports into Provmon.

- **SPADE**. The SPADE system is an increasingly popular option for provenance auditing, but collecting provenance in user space limits SPADE's expressiveness and creates the potential for incomplete provenance. To address this limitation, we have created a mechanism that reports LPM provenance into SPADE's Domain-Specific Language pipe [29]. This permits the collection of whole-system provenance while simultaneously leveraging SPADE's existing storage, remote query, and visualization utilities.

- **Using Provenance to Expedite MAC Policies (UPTEMPO)**. Using LPM as a collection mechanism, Moyer et al. investigate provenance analysis as a means of administrating Mandatory Access Control (MAC) policies [54]. UPTEMPO first observes system execution in a sterile environment, aggregating LPM provenance in a centralized data store. It then recovers the implicit information flow policy through mining the provenance store to generate a MAC policy for the distributed system, decreasing both administrator effort and the potential for misconfiguration.

3.4.4 Provenance Recorders

LPM provides modular support for different storage through *provenance recorders*. To prevent an infinite provenance loop, recorders are flagged as provenance-opaque [48] using the `security.provenance` extended attribute, which is checked by LPM before creating a new event. Each recorder was designed to be as agnostic to the active LPM as possible, making them easy to adapt to new modules.

We currently provide provenance recorders that offer backend storage for *Gzip*, *PostGreSQL*, *Neo4j*, and *SNAP*. Commentary on our PostGreSQL and Neo4j reporters can be found in our technical report [8]. We make use of the Gzip and SNAP recorders during our evaluation in Section 6.

The Gzip recorder incurs low storage overheads and fast insertion speeds. On our test bed, we observed this recorder processing up to 400,000 events per second from the Provmon provenance stream. However, because the provenance is not stored in an easily queried form, this back-end is best suited for environments where queries are an offline process.

To create graph storage that was efficient enough for LPM, we used the SNAP graphing library[11] to design a recorder that maintains an in-memory graph database that is fully compliant with the W3C PROV-DM Model [59]. We have observed insertion speeds of over 150,000 events per second using the SNAP recorder, and highly efficient querying as well. This recorder is further evaluated in Section 6.

3.4.5 Workflow Provenance

To support layered provenance while preserving our security goals, we require a means of evaluating the integrity of user space provenance disclosures. To accomplish this, we extend the LPM Provenance Recorder to use the Linux Integrity Measurement Architecture (IMA) [35, 52]. IMA computes a cryptographic hash of each binary before execution, extends the measurement into a TPM Platform Control Register (PCR), and stores the measurement in kernel memory. This set of measurements can be used by the Recorder to make a decision about the integrity of the a Provenance-Aware Application (PAA) prior to accepting the disclosed provenance. When a PAA wishes to disclose provenance, it opens a

[11] See http://snap.stanford.edu

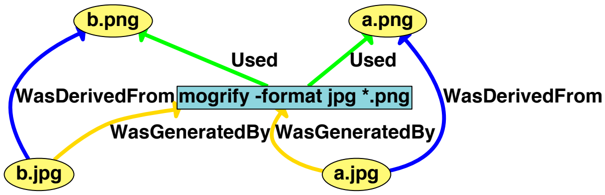


Figure 4: A provenance graph of image conversion. Here, workflow provenance (*WasDerivedFrom*) encodes a relationship that more accurately identifies the output files' dependencies compared to only using kernel layer observations (*Used*, *WasGeneratedBy*).

new UNIX domain socket to send the provenance data to the Provenance Recorder. The Recorder uses its own UNIX domain socket to recover the process's pid, then uses the `/proc` filesystem to find the full path of the binary, then uses this information to look up the PAA in the IMA measurement list. The disclosed provenance is recorded only if the signature of PAA matches a known-good cryptographic hash.

As a demonstration of this functionality, we created a provenance-aware version of the popular ImageMagick utility [12]. ImageMagick contains a batch conversion tool for image reformatting, `mogrify`. Shown in Figure 4, `mogrify` reads and writes multiple files during execution, leading to an *overtainting* problem – at the kernel layer, LPM is forced to conservatively assume that all outputs were derived from all inputs, creating false dependencies in the provenance record. To address this, we extended the Provmon protocol to support a new message, `provmsg_imagemagick_convert`, which links an input file directly to its output file. When the recorder receives this message, it first checks the list of IMA measurements to confirm that ImageMagick is in a good state. If successful, it then annotates the existing provenance graph, connecting the appropriate input and output objects with *WasDerivedFrom* relationships. Our instrumentation of ImageMagick demonstrates that LPM supports layered provenance at no additional cost over other provenance-aware systems [29, 43], and does so in a manner that provides assurance of the integrity of the provenance log.

3.5 Deployment

We now demonstrate how we used LPM in the deployment of a secure provenance-aware system. Additional background on the security technologies use in our deployment can be found in our technical report [8].

3.5.1 Platform Integrity

We configured LPM to run on a physical machine with a Trusted Platform Module (TPM). The TPM provides a root of trust that allows for a measured boot of the system. The TPM also provides the basis for remote attestations to prove that LPM was in a known hardware and software configuration. The BIOS's core root of trust for measurement (CRTM) bootstraps a series of code measurements prior to the execution of each platform component. Once booted, the kernel then measures the code for user space components (e.g., provenance recorder) before launching them, through the use of the Linux Integrity Measurement Architecture (IMA)[52]. The result is then extended into TPM PCRs, which forms a verifiable chain of trust that shows the integrity of the system via a digital signature over the measurements. A remote verifier can use this chain to determine the current state of the system using TPM attestation.

We configured the system with Intel's Trusted Boot (tboot),[13] which provides a secure boot mechanism, preventing system from booting into the environment where critical components (e.g., the BIOS, boot loader and the kernel) are modified. Intel tboot relies on the Intel TXT[14] to provide a secure execution environment. [15] Additionally, we compiled support for IMA into the provenance-aware kernel, which is necessary in order for the LPM Recorder to be able to measure the integrity of provenance-aware applications.

3.5.2 Runtime Integrity

After booting into the provenance-aware kernel, the runtime integrity of the TCB (defined in §3.2) must also be assured. To protect the runtime integrity of the kernel, we deploy a Mandatory Access Control (MAC) policy, as implemented by Linux Security Modules. On our prototype deployments, we enabled SELinux's MLS policy, the security of which was formally modeled by Hicks et al. [33]. Refining the SELinux policy to prevent Access Vector Cache (AVC) denials on LPM components required minimal effort; the only denial we encountered was when using the PostgreSQL recorder, which was quickly remedied with the `audit2allow` tool. Preserving the integrity of LPM's user space components, such as the provenance recorder, was as simple as creating a new policy module. We created a policy module to protect the LPM recorder and storage back-end using the `sepolicy` utility. Uncompiled, the policy module was only 135 lines.

[12] See http://www.imagemagick.org

[13] See http://sf.net/projects/tboot

[14] See https: //www.kernel.org/doc/Documentation/intel_txt.txt

[15] For virtual environments, similar functionality can be provided on Xen via TPM *sealing* and the virtual TPM (vTPM), which is bound to the physical TPM of the host system.

4 Security

In this section, we demonstrate that our system meets all of the required security goals for trustworthy whole-system provenance. In this analysis, we consider an LPM deployment on a physical machine that was enabled with the Provmon module and has been configured to the conditions described in Section 3.5.

Complete (G1). We defined whole-system provenance as a complete description of agents (users, groups) controlling activities (processes) interacting with controlled data types during system execution (§ 3.1). LPM attempts to track these system objects through the placement of provenance hooks (§3.4.1), which directly follow each LSM authorization hook. The LSM's complete mediation property has been formally verified [20, 64]; in other words, there is an authorization hook prior to every security-sensitive operation. Because every interaction with a controlled data type is considered security-sensitive, we know that a provenance hook resides on all control paths to the provenance-sensitive operations. LPM is therefore capable of collecting complete provenance on the host.

It is important to note that, as a consequence of placing provenance hooks beneath authorization hooks, LPM is unable to record failed access attempts. However, inserting the provenance layer beneath the security layer ensures accuracy of the provenance record. Moreover, failed authorizations are a different kind of metadata than provenance because they do not describe processed data; this information is better handled at the security layer, e.g., by the SELinux Access Vector Cache (AVC) Log.

Tamperproof (G2). The runtime integrity of the LPM trusted computing base is assured via the SELinux MLS policy, and we have written a policy module that protects the LPM user space components (§3.5.2). Therefore, the only way to disable LPM would be to reboot the system into a different kernel; this action can be disallowed through secure boot techniques,[13] and is detectable by remote hosts via TPM attestation (§3.5.1).

Verifiable (G3). While we have not conducted an independent formal verification of LPM, our argument for its correctness is as follows. A provenance hook follows each LSM authorization hook in the kernel. The correctness of LSM hook placement has been verified through both static and dynamic analysis techniques [20, 25, 34]. Because an authorization hook exists on the path of every sensitive operation to controlled data types, and LPM introduces a provenance hook behind each authorization hook, LPM inherits LSM's formal assurance of complete mediation over controlled data types. This is sufficient to ensure that LPM can collect the provenance of every sensitive operation on controlled data types in the kernel (i.e., whole-system provenance).

Authenticated Channel (G4). Through use of Netfilter hooks [57], LPM embeds a DSA signature in every outbound network packet. Signing occurs immediately prior to transmission, and verification occurs immediately after reception, making it impossible for an adversary-controlled application running in user space to interfere. For both transmission and reception, the signature is invisible to user space. Signatures are removed from the packets before delivery, and LPM feigns ignorance that the options field has been set if `get_options` is called. Hence, LPM can enforce that all applications participate in the commitment protocol.

Prior to implementing our own message commitment protocol in the kernel, we investigated a variety of existing secure protocols. The integrity and authenticity of provenance identifiers could also be protected via IPsec [36], SSL tunneling,[16] or other forms of encapsulation [5, 65]. We elected to move forward with our approach because 1) it ensures the monitoring of all *all* processes and network events, including non-IP packets, 2) it does not change the number of packets sent or received, ensuring that our provenance mechanism is minimally invasive to the rest of the Linux network stack, and 3) it preserves compatibility with non-LPM hosts. An alternative to DSA signing would be HMAC [9], which offers better performance but requires pairwise keying and sacrifices the non-repudiation policy; BLS, which approaches the theoretical maximum security parameter per byte of signature [12]; or online/offline signature schemes [15, 23, 26, 55].

Authenticated Disclosures (G5). We make use of IMA to protect the channel between LPM and provenance-aware applications wishing to disclose provenance. IMA is able to prove to the provenance recorder that the application was unmodified at the time it was loaded into memory, at which point the recorder can accept the provenance disclosure into the official record. If the application is known to be correct (e.g., through formal verification), this is sufficient to establish the runtime integrity of the application. However, if the application is compromised after execution, this approach is unable to protect against provenance forgery.

A separate consideration for all of the above security properties are Denial of Service (DoS) attacks. *DoS attacks on LPM do not break its security properties.* If an attacker launches a resource exhaustion attack in order to prevent provenance from being collected, all kernel operations will be disallowed and the host will cease to function. If a network attacker tampers with a packet's provenance identifier, the packet will not be delivered to the recipient application. In all cases, the provenance record remains an accurate reflection of system events.

[16] See http://docs.oracle.com/cd/E23823_01/html/816-5175/kssl-5.html

Algorithm 1 Summarizes a's propagation through the system.

```
Require: $a$ is an entity
procedure REPORT($a$)
    $Locations = [\,]$                                ▷ Assigns an empty list.
    for each $s$ in $a, FindSuccessors(a)$ do
        if $s.type$ is File then
            $Locations.Add(< s.disk, s.directory >)$
        else if $s.type$ is Network Packet then
            $Locations.Add(< s.remote\_ip, s.port >)$
        end if
    end for
    return $Locations$
end procedure
```

5 LPM Application: Provenance-Based Data Loss Prevention

To further demonstrate the power of LPM, we now introduce a set of mechanisms for Provenance-Based Data Loss Prevention (PB-DLP) that offer dramatically simplified administration and improved enforcement over existing DLP systems. A provenance-based approach is a novel and effective means of handling data loss prevention; to our knowledge, we are the first in the literature to do so. The advantage of our approach when compared to existing systems is that LPM-based provenance-aware systems already perform system-wide capture of information flows between kernel objects. Data loss prevention in such a system therefore becomes a matter of preventing all derivations of a sensitive source entity, e.g., a Payment Card Industry (PCI) database, from being written to a monitored destination entity (e.g., a network interface).

We begin by defining a policy format for PB-DLP. Individual rules take the form

$$< Srcs = [src_1, src_2, \ldots, src_n], dst >$$

where *Srcs* is a list of entities representing persistent data objects, and *dst* is a single entity representing either a persistent data object such as a file or interface or an abstract entity such as a remote host. The goal for PB-DLP is as follows – an entity e_1 with ancestors A is written to entity e_2 if and only if $A \not\supseteq Srcs$ for *all* rules in the rule set where $e_2 = dst$. The reason that sources are expressed as sets is that, at times, the union of information is more sensitive than its individual components. For example, sharing a person's last name *or* birthdate may be permissible, while sharing the last name *and* birthdate is restricted as PII.[2]

Below, we define the functions that realize this goal. First, we define two provenance-based functions as the basis for a DLP *monitoring phase*, which allows administrators to learn more about the propagation of sensitive data on their systems. Then, we define mechanisms for a DLP *enforcement phase*.

Algorithm 2 Mediates request to write e to d given $Rules$.

```
Require: $e, d$ are entities
Require: $Rules$ is a PB-DLP policy
procedure PROVWRITE($e, d, Rules$)
    for each $rule$ in $Rules$ do
        if $d = rule.dst$ then
            $A = FindAncestors(e)$
            $NumSrcs = length(rule.Srcs)$
            for each $src$ in $rule.Srcs$ do
                if $src$ in $A$ then
                    $NumSrcs--$
                end if
            end for
            if $NumSrcs = 0$ then                     ▷ $A \supseteq Srcs$, deny.
                return PB-DLP_DENY
            end if
        end if
    end for
    return PB-DLP_PERMIT                              ▷ $A \not\supseteq Srcs$, permit.
end procedure
```

5.1 Monitoring Phase

The goal of monitoring is to allow administrators to reason about how sensitive data is stored and put to use on their systems. The end product of the monitor phase is a set of rules (a policy) that restrict the permissible flows for sensitive data sources. Monitoring is an ongoing process in DLP, where administrators attempt to iteratively improve protection against data leakage. The first step is to identify the data that needs protection. Identifying the source of such information is often quite simple; for example, a database of PCI or PII data. However, reliably finding data objects that were derived from this source is extraordinarily complicated using existing solutions, but is simple now with LPM. To begin, we define a helper function for system monitoring:

1. *FindSuccessors(Entity)*: This function performs a provenance graph traversal to obtain the list of data objects derived from *Entity*.

FindSuccessors can then be used as the basis for a function that summarizes the spread of sensitive data:

2. *Report(Entity)*: List the locations that a target object and its successors have propagated. This function is defined in Algorithm 1.

The information provided by *Report* is similar to the data found in the Symantec DLP Dashboard [1], and could be used as the backbone of a PB-DLP user interface. Administrators can use this information to write a PB-DLP policy or revise an existing one.

5.2 Enforcement Phase

Possessing a PB-DLP policy, the goal of the enforcement phase is to prevent entities that were derived from sensitive sources from being written to restricted locations. To do so, we need to inspect the object's provenance to discover the entities from which it was derived. We define the following helper function:

3. *FindAncestors(Entity)*: This function performs a provenance graph traversal to obtain the list of data objects used in the creation of *Entity*.

FindAncestors can be then used as the basis for a function that prevents the spread of sensitive data:

4. *ProvWrite(Entity, Destination, Rules)*: Write the target entity to the destination if and only if it is valid to the provided rule set, as defined in Algorithm 2.

5.3 File Transfer Application

In many enterprise networks that are isolated from the Internet via firewalls and proxies, it is desirable to share files with external users. File transfer services are one way to achieve this, and provide a single entry/exit point to the enterprise network where files being transferred can be examined before being released.[17] In the case of incoming files, scans can check for known malware, and in some cases, check for other types of malicious behavior from unknown malware.

We implemented PB-DLP as a file transfer application for provenance-aware systems using LPM's Provmon module. The application interfaced with LPM's SNAP recorder using a custom API. Before permitting a file to be transmitted to a remote host, the application ran a query that traversed *WasDerivedFrom* edges to return a list of the file's ancestors, permitting the transfer only if the file was not derived from a restricted source. PB-DLP allows internal users to share data, while ensuring that sensitive data is not exfiltrated in the process.

Because provenance graphs continue to grow indefinitely over time, in practice the bottleneck of this application is the speed of provenance querying. We evaluate the performance of PB-DLP queries in Section 6.3.

5.4 PB-DLP Analysis

Below, we select two open source systems that approximate *label based* and *regular expression (regex) based* DLP solutions, and compare their benefits to PB-DLP.

5.4.1 Label-Based DLP

The SELinux MLS policy [31] provides information flow security through a label-based approach, and could be used to approximate a DLP solution without relying on commercial products. Proprietary label-based DLP systems rely on manual annotations provided by users, requiring them to provide correct labeling based on their knowledge of data content. Using SELinux as an exemplar labeling system is therefore an *extremely conservative* approach to analysis.

Within an MLS system, each subject, and object, is assigned a classification level, and categories, or compartments. Consider an example system, with classification levels, $\{A,B\}$ with A dominating B, and compartments $\{\alpha,\beta\}$. We can model our policy as a lattice, where each node in the lattice is a tuple of the form $\{< level >, \{compartments\}\}$. Once the policy is defined, it is possible to enforce the simple and *-properties. If a user has access to data with classification level A, and compartment α, he cannot read anything in compartment $\{\beta\}$ (*no read-up*). Furthermore, when data is accessed in A,$\{\alpha\}$, the user cannot write anything to B,$\{\alpha\}$ (*no write-down*).

In order to use SELinux's MLS enforcement as a DLP solution, the administrator configures the policy to enforce the constraint that no data of specified types can be sent over the network. However, this is difficult in practice. Consider an example system that processes PII. The users of the system may need to access information, such as last names, and send these to the payroll department to ensure that each employee receives a paycheck. Separately, the user may need to send a list of birthdays to another user in the department to coordinate birthday celebrations for each month. Either of these activities are acceptable (Figure 5, Decision Condition 2). However, it is common practice for organizations to have stricter sharing policies for data that contains multiple forms of PII, so while either of these identifiers could be transmitted in isolation, the two pieces of information combined could not be shared (Figure 5, Decision Condition 3).

The MLS policy cannot easily handle this type of data fusion. In order to provide support for correctly labeling fused data, an administrator would need to define the power set of all compartments within the MLS policy. In the example above, the administrator would define the following compartments: $\{\}$, $\{\alpha\}$, $\{\beta\}$, $\{\alpha,\beta\}$. In the default configuration SELinux supports 256 unique categories, meaning an SELinux DLP policy could only support eight types of data. Furthermore, the MLS policy does not support defining multiple categories within a single sensitivity level[18]. This implies that the MLS policy cannot support having a security level for A,$\{\alpha\}$ and for A,$\{\alpha,\beta\}$. Instead, the most restrictive labeling must be defined to protect the data on the system. In contrast, PB-DLP can support an arbitrary number of data fusions.

[17] Two examples of vendors that provide this capability are FireEye (http://www.fireeye.com) and Accellion (http://www.accellion.com/)

[18] See the definition of level statements at `http://selinuxproject.org/page/MLSStatements`

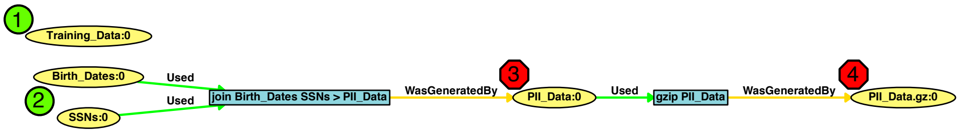


Figure 5: A provenance graph of PII data objects that are first fused and then transformed. The numbers mark DLP decision conditions. Objects marked by green circles *should not* be restricted, while red octagons *should* be restricted. Label-Based DLP correctly handles data resembling PII (1,2) and data transformations (4), but struggles with data fusions (3). Regex-Based DLP correctly identifies data fusions (3), but is prone to incorrect handling of data resembling PII (1) and fails to identify data transformations (4). PB-DLP correctly handles all conditions.

5.4.2 Regex-Based DLP

The majority of DLP software relies on pattern matching techniques to identify sensitive data. While enterprise solutions offer greater sophistication and customizability, their fundamental approach resembles that of Cornell's Spider [7], a forensics tools for identifying sensitive personal data (e.g., credit card or social security numbers). Because it is open source, we make use of Spider as an exemplar application for regex-based DLP.

Regex approaches are prone to *false positives*. Spider is pre-configured with a set of regular expressions for identifying potential PII, e.g., `(\d{3}-\d{2}-\d{4})` identifies a social security number. However, it is common practice for developers to generate and distribute training datasets to aid in software testing (Figure 5, Decision Condition 1). Spider is oblivious to information flows, instead searching for content that bears structural similarity to PII, and therefore would be unable to distinguish between true PII and training data. PB-DLP tracks the propagation of data from its source onwards, and could trivially differentiate between true PII and training sets.

Regex approaches are also prone to *false negatives*. Even after the most trivial data transformations, PII and PCI data is no longer identifiable to the Spider system (Figure 5, Decision Condition 4), permitting its exfiltration. To demonstrate, we generated a file full of random valid social security numbers that Spider was able to identify. We then ran `gzip` on the file and stored it in a second file. Spider was unable to identify the second file, but PB-DLP correctly identified both files as PII since the `gzip` output was derived from a sensitive input.

6 Evaluation

We now evaluate the performance of LPM. Our benchmarks were run on a bare metal server machine with 12 GB memory and 2 Intel Xeon quad core CPUs. The Red Hat 2.6.32 kernel was compiled and installed under 3 different configurations: all provenance disabled (*Vanilla*), LPM scaffolding installed but without an enabled module (*LPM*), and LPM installed with the Provmon module enabled (*Provmon*).

Test Type	Vanilla	LPM	Provmon
Process tests, times in μseconds (smaller is better)			
null call	0.14	0.14 (0%)	0.14 (0%)
null I/O	0.21	0.21 (0%)	0.32 (52%)
stat	1.57	1.6 (2%)	2.8 (78%)
open/close file	2.75	2.42 (-12%)	3.91 (42%)
signal install	0.25	0.25 (0%)	0.25 (0%)
signal handle	1.37	1.29 (-6%)	1.39 (1%)
fork process	380	396 (4%)	401 (6%)
exec process	873	879 (1%)	911 (4%)
shell process	2990	3000 (0%)	3113 (4%)
File and memory latencies in μseconds (smaller is better)			
file create (0k)	11.5	11.2 (-3%)	15.8 (37%)
file delete (0k)	8.51	8.12 (-5%)	11.8 (39%)
file create (10k)	23.4	21.6 (-8%)	28.8 (23%)
file delete (10k)	12.5	12 (-4%)	14.7 (18%)
mmap latency	1062	1053 (-1%)	1120 (5%)
protect fault	0.32	0.3 (-6%)	0.346 (8%)
page fault	0.016	0.016 (0%)	0.016 (0%)
100 fd select	1.53	1.53 (0%)	1.53 (0%)

Table 1: LMBench measurements for LPM kernels. All times are in microseconds. Percent overhead for modified configurations are shown in parenthesis.

6.1 Collection Performance

We used LMBench to microbenchmark LPM's impact on system calls as well as file and memory latencies. Table 1 shows the averaged results over 10 trials for each kernel, with a percent overhead calculation against Vanilla. For most measures, the performance differences between LPM and Vanilla are negligible. Comparing Vanilla to Provmon, there are several measures in which overhead is noteworthy: *stat, open/close, file creation and deletion*. Each of these benchmarks involve LMBench manipulating a temporary file that resides in `/usr/tmp/lat_fs/`. Because an absolute path is provided, before each system call occurs LMBench first traverses the path to the file, resulting in the creation of 3 different provenance events in Provmon's `inode_permission` hook, each of which is transmitted to user space via the kernel relay. While

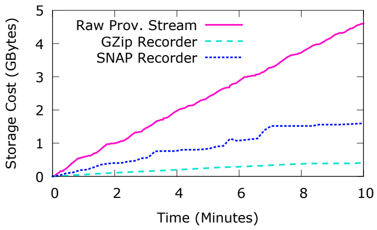


Figure 6: Growth of provenance storage overheads during kernel compilation.

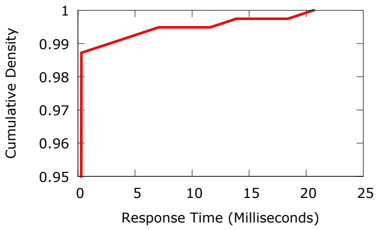


Figure 7: Performance of ancestry queries for objects created during kernel compilation.

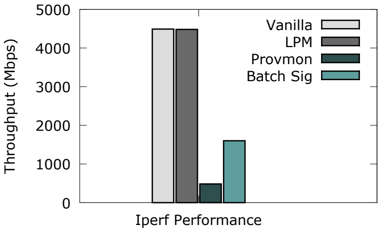


Figure 8: LPM network overhead can be reduced with batch signature schemes.

the overheads seem large for these operations, the logging of these three events only imposes approximately 1.5 microseconds per traversal. Moreover, the overhead for opening and closing is significantly higher than the overhead than reads and writes (*Null I/O*); thus, the higher open/close costs are likely to be amortized over the course of regular system use. A provenance module could further reduce this overhead by maintaining state about past events within the kernel, then blocking the creation of redundant provenance records.

Test	Vanilla	Provmon	Overhead
Kernel Compile	598 sec	614 sec	2.7%
Postmark	25 sec	27 sec	7.5%
Blast	376 sec	390 sec	4.8%

Table 2: Benchmarking Results. Our provenance module imposed just 2.7% overhead on kernel compilation.

To gain a more practical sense of the costs of LPM, we also performed multiple benchmark tests that represented realistic system workloads. Each trial was repeated 5 times to ensure consistency. The results are summarized in Table 2. For the kernel compile test, we recompiled the kernel source (in a fixed configuration) while booted into each of the kernels. Each compilation occurred on 16 threads. The LPM scaffolding (without an enabled module) is not included, because in both tests it imposed less than 1% overhead. In spite of seemingly high overheads for file I/O, Provmon imposes just 2.7% overhead on kernel compilation, or 16 seconds. The Postmark test simulates the operation of an email server. It was configured to run 15,000 transactions with file sizes ranging from 4 KB to 1 MB in 10 subdirectories, with up to 1,500 simultaneous transactions. The Provmon module imposed just 7.5% overhead on this task. To estimate LPM's overhead for scientific applications, we ran the BLAST benchmarks [19], which simulates biological sequence workloads obtained from analysis of hundreds of thousands of jobs from the National Institutes of Health.

For kernel compile and postmark, Provmon outperforms the PASS system, which exacted 15.6% and 11.5% overheads on kernel compilation and postmark, respectively [43]. Provmon introduces comparable kernel compilation overhead to Hi-Fi (2.8%) [48]. It is difficult to compare our Blast results to SPADE and PASS, as both used a custom workload instead of a publicly available benchmark. SPADE reports an 11.5% overhead on a large workload [29], while PASS reports just an 0.7% overhead. Taken as a whole, though, LPM collection either meets or exceeds the performance of past systems, while providing additional security assurances.

6.2 Storage Overhead

A major challenge to automated provenance collection is the storage overhead incurred. We plotted the growth of provenance storage using different recorders during the kernel compilation benchmark, shown in Figure 6. LPM generated 3.7 GB of raw provenance. This required only 450 MB of storage with the Gzip recorder, but provenance cannot be efficiently queried in this format. The SNAP recorder builds an in-memory provenance graph. We approximated the storage overhead through polling the virtual memory consumed by the recorder process in the `/proc` filesystem. The SNAP graph required 1.6 GB storage; the reduction from the raw provenance stream is due to the fact that redundant events did not lead to the creation of new graph components. In contrast, the PASS system generates 1.3 GB of storage overhead during kernel compilation while collecting less information (e.g., shared memory). LPM's storage overheads are thus comparable to other provenance-aware systems.

6.3 Query Performance (PB-DLP)

We evaluated query performance using our exemplar PB-DLP application and LPM's SNAP recorder. The provenance graph that was populated using the routine from the kernel compile benchmark. This yielded a raw provenance stream of 3.7 GB, which was translated by the recorder into a graph of 6,513,398 nodes and 6,754,059

[19] See http://fiehnlab.ucdavis.edu/staff/kind/Collector/Benchmark/Blast_Benchmark

edges. We were then able to use the graph to issue *ancestry queries*, in which the subgraphs were traversed to find the data object ancestors of a given file. Because we did not want ephemeral objects with limited ancestries to skew our results, we only considered the results of objects with more than 50 ancestors.

In the *worst case*, which was a node that had 17,696 ancestors, the query returned in just 21 milliseconds. Effectively, we were able to query object ancestries at line speed for network activity. We are confident that this approach can scale to large databases through precomputation of expensive operations at data ingest, making it a promising strategy for provenance-aware distributed systems; however, we note that these results are highly dependent on the size of the graph. Our test graph, while large, would inevitably be dwarfed by the size of the provenance on long-lived systems. Fortunately, there are also a variety of techniques for reducing these costs. Bates et al. show that the results from provenance graph traversals can be extensively cached when using a fixed security policy, which would allow querying to amortize to a small constant cost [7]. LPM could also be extended to support deduplication [63, 62] and policy-based pruning [6, 13], both of which could further improve performance by reducing the size of provenance graphs.

6.4 Message Commitment Protocol

Under each kernel configuration, we performed *iperf* benchmarking to discover LPM's impact on TCP throughput. iperf was launched in both client and server mode over `localhost`. The client was launched with 16 threads, two per each CPU. Our results can be found in Figure 8. The Vanilla kernel reached 4490 Mbps. While the LPM framework imposed negligible cost compared to the vanilla kernel (4480 Mbps), Provmon's DSA-based message commitment protocol reduced throughput by an order of magnitude (482 Mbps). Through use of `printk` instrumentation, we found that the average overhead per packet signature was 1.2 ms.

This result is not surprising when compared to IPsec performance. IPsec's Authentication Header (AH) mode uses an HMAC-based approach to provide similar guarantees as our protocol. AH has been shown to reduce throughput by as much as half [16]. An HMAC approach is a viable alternative to establish integrity and data origin authenticity and would also fit into the options field, but would require the negotiation of IPsec security associations. Our message commitment protocol has the benefit of being fully interoperable with other hosts, and does not require a negotiation phase before communication occurs. Another option for increasing throughput would be to employ CPU instruction extensions [30] and security co-processor [19] to accelerate the speed of DSA. Yet another approach to reducing our impact on network performance would be to employ a batch signature scheme [11]. We tested this by transmitting a signature over every 10 packets during TCP sessions, and found that throughput increased by 3.3 times to approximately 1600 Mbps. Due to the fact that this overhead may not be suitable for some environments, Provmon can be configured to use Hi-Fi identifiers [48], which are vulnerable to network attack but impose negligible overhead. LPM's impact on network performance is specific to the particular module, and can be tailored to meet the needs of the system.

7 Discussion

Without the aid of provenance-aware applications, LPM will struggle to accurately track dependencies through workflow layer abstractions. The most obvious example of such an abstraction is the copy/paste buffer in windowing applications like Xorg. This is a known side channel for kernel layer security mechanisms, one that has been addressed by the Trusted Solaris project [10], Trusted X [21, 22], the SELinux-aware X window system [37], SecureView [20], and General Dynamics' TVE [21]. Without provenance-aware windowing, LPM will conservatively assume that all files opened for reading are dependencies of the paste buffer, leading to false dependencies. LPM is also unable to observe system side channels, such as timing channels or L2 cache measurements [51], a limitation shared by many other security solutions [18].

Although we have not presented a secure distributed provenance-aware system in this work, LPM provides the foundation for the creation of such a system. In the presented modules, provenance is stored locally by the host and retrieved on an as-needed basis from other hosts. This raises availability concerns as hosts inevitably begin to fail. Availability could be improved with minimal performance and storage overheads through Gehani et al.'s approach of duplicating provenance at k neighbors with a limited graph depth d [27, 28].

Finally, LPM does not address the matter of provenance confidentiality; this is an important challenge that is explored elsewhere in the literature [14, 46]. LPM's Recorders provide interfaces that can be used to introduce an access control layer onto the provenance store.

8 Related Work

While myriad provenance-aware systems have been proposed in the literature, the majority *disclose* provenance within an application [32, 41, 65] or workflow [24, 58]. It

[20] See http://www.ainfosec.com/secureview

[21] See http://gdc4s.com/tve.html

is difficult or impossible to obtain *complete* provenance in this manner. This is because systems events that occur outside of the application, but still effect its execution, will not appear in the provenance record.

The alternative to disclosed systems are *automatic* provenance-aware systems, which collect provenance transparently within the operating system. Gehani et al.'s SPADE is a multi-platform system for eScience and grid computing audiences, with an emphasis on low latency and availability in distributed environments [29]. SPADE's *provenance reporters* make use of familiar application layer utilities to generate provenance, such as polling `ps` for process information and `lsof` for network information. This gives rise to the possibility of *incomplete* provenance due to race conditions. The PASS project collects the provenance of system calls at the virtual filesystem (VFS) layer. PASSv1 provides base functions for provenance collection that observe processes' file I/O activity [43]. Because these basic functions are manually placed around the kernel, there is no clear way to extend PASSv1 to support additional collection hooks; we address this limitation in the modular design of LPM. PASSv2 introduces a Disclosed Provenance API for tighter integration between provenance collected at different layers of abstraction, e.g., at the application layer [44]. PASSv2 assumes that disclosing processes are benign, while LPM provides a secure disclosure mechanism for attesting the correctness of provenance-aware applications. Both SPADE and PASS are designed for benign environments, making no attempt to protect their collection mechanisms from an adversary.

Previous work has considered the security of provenance under relaxed threat models relative to LPM's. In SProv, Hasan et al. introduce *provenance chains*, cryptographic constructs that prevent the insertion or deletion of provenance inside of a series of events [32]. SProv effectively demonstrates the authentication properties of this primitive, but is not intended to serve as a secure provenance-aware system; attackers can still append false records to the end of the chain, delete the whole chain, or disable the library altogether. Zhou et al. consider provenance corruption an inevitability, and show that provenance can detect *some* malicious hosts in distributed environments provided that a critical mass of correct hosts still exist [65]. They later strengthen these assurances through use of provenance-aware software-defined networking [5]. These systems consider only network events, and are unable to speak to the internal state of hosts. Lyle and Martin sketch the design for a secure provenance monitor based on trusted computing [40]. However, they conceptualize provenance as a TPM-aided proof of code execution, overlooking interprocess communication and other system activity that could inform execution results, and therefore offer information that is too coarse-grained to meet the needs of some applications. Moreover, to the best of our knowledge their system is unimplemented.

The most promising model to date for secure provenance collection is Pohly et al.'s Hi-Fi system [48]. Hi-Fi is a Linux Security Module (LSM) that collects *whole-system provenance* that details the actions of processes, IPC mechanisms, and even the kernel itself (which does not exclusively use system calls). Hi-Fi attempts to provide a provenance reference monitor [42], but remains vulnerable to the provenance-aware adversary that we describe in Section 3.2. Enabling Hi-Fi blocks the installation of other LSM's, such as SELinux or Tomoyo, or requires a third party patch to permit module stacking. This blocks the installation of MAC policy on the host, preventing runtime integrity assurances. Hi-Fi is also vulnerable to adversaries in the network, who can strip the provenance identifiers from packets in transit, resulting in irrecoverable provenance. Unlike LPM, Hi-Fi does not attempt to provide layered provenance services, and therefore does not consider the integrity and authenticity of provenance-aware applications.

Provenance collection is a form of information flow monitoring that is related, but fundamentally distinct, from past areas of study. Due to space constraints, our discussion of Information Flow Control (IFC) systems has been relegated to our technical report [8].

9 Conclusion

In this work, we have presented LPM, a platform for the creation of trusted provenance-aware execution environments. Our system imposes as little as 2.7% performance overhead on normal system operation, and can respond to queries about data object ancestry in tens of milliseconds. We have used LPM as the foundation of a provenance-based data loss prevention system that can scan file transmissions to detect the presence of sensitive ancestors in just tenths of a second. The Linux Provenance Module Framework is an exciting step forward for both provenance- and security-conscious communities.

Acknowledgements

We would like to thank Rob Cunningham, Alin Dobra, Will Enck, Jun Li, Al Malony, Patrick McDaniel, Daniela Oliveira, Nabil Schear, Micah Sherr, and Patrick Traynor for their valuable comments and insight, as well as Devin Pohly for his sustained assistance in working with Hi-Fi, and Mugdha Kumar for her help developing LPM SPADE support. This work was supported in part by the US National Science Foundation under grant numbers CNS-1118046, CNS-1254198, and CNS-1445983.

Availability

The LPM code base, including all user space utilities and patches for both Red Hat and the mainline Linux kernels, is available at http://linuxprovenance.org.

References

[1] Symantec Data Loss Prevention Customer Brochure. http://www.symantec.com/data-loss-prevention.

[2] What's Yours is Mine: How Employees are Putting Your Intellectual Property at Risk. https://www4.symantec.com/mktginfo/whitepaper/WP_WhatsYoursIsMine-HowEmployeesarePutting YourIntellectualPropertyatRisk_dai211501_cta69167.pdf.

[3] R. Aldeco-Pérez and L. Moreau. Provenance-based Auditing of Private Data Use. In *Proceedings of the 2008 International Conference on Visions of Computer Science*, VoCS'08, Sept. 2008.

[4] J. P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, Air Force Electronic Systems Division, 1972.

[5] A. Bates, K. Butler, A. Haeberlen, M. Sherr, and W. Zhou. Let SDN Be Your Eyes: Secure Forensics in Data Center Networks. In *NDSS Workshop on Security of Emerging Network Technologies*, SENT, Feb. 2014.

[6] A. Bates, K. R. B. Butler, and T. Moyer. Take Only What You Need: Leveraging Mandatory Access Control Policy to Reduce Provenance Storage Costs. In *7th Workshop on the Theory and Practice of Provenance*, TaPP'15, July 2015.

[7] A. Bates, B. Mood, M. Valafar, and K. Butler. Towards Secure Provenance-based Access Control in Cloud Environments. In *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 277–284, New York, NY, USA, 2013. ACM.

[8] A. Bates, D. Tian, K. R. B. Butler, and T. Moyer. Linux Provenance Modules: Trustworthy Whole-System Provenance for the Linux Kernel. Technical Report REP-2015-578, University of Florida CISE Dept, 2015.

[9] M. Bellare, R. Canetti, and H. Krawczyk. Keyed Hash Functions and Message Authentication. In *Proceedings of Crypto'96*, volume 1109 of *LNCS*, pages 1–15, 1996.

[10] M. Bellis, S. Lofthouse, H. Griffin, and D. Kucukreisoglu. Trusted Solaris 8 4/01 Security Target. 2003.

[11] A. Bittau, D. Boneh, M. Hamburg, M. Handley, D. Mazieres, and Q. Slack. Cryptographic protection of TCP Streams (tcpcrypt). https://tools.ietf.org/html/draft-bittau-tcp-crypt-01.

[12] D. Boneh, B. Lynn, and H. Shacham. Short Signatures from the Weil Pairing. In C. Boyd, editor, *Advances in Cryptology – ASIACRYPT 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 514–532. Springer Berlin Heidelberg, 2001.

[13] U. Braun, S. L. Garfinkel, D. A. Holland, K.-K. Muniswamy-Reddy, and M. I. Seltzer. Issues in Automatic Provenance Collection. In *International Provenance and Annotation Workshop*, pages 171–183, 2006.

[14] U. Braun and A. Shinnar. A Security Model for Provenance. Technical Report TR-04-06, Harvard University Computer Science Group, 2006.

[15] D. Catalano, M. Di Raimondo, D. Fiore, and R. Gennaro. Off-line/On-line Signatures: Theoretical Aspects and Experimental Results. In *PKC'08: Proceedings of the Practice and theory in public key cryptography, 11th international conference on Public key cryptography*, pages 101–120, Berlin, Heidelberg, 2008. Springer-Verlag.

[16] S. Chaitanya, K. Butler, A. Sivasubramaniam, P. McDaniel, and M. Vilayannur. Design, Implementation and Evaluation of Security in iSCSI-based Network Storage Systems. In *Proceedings of the Second ACM Workshop on Storage Security and Survivability*, StorageSS '06, pages 17–28, New York, NY, USA, 2006. ACM.

[17] D. D. Clark and D. R. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *IEEE S&P*, Oakland, CA, USA, Apr. 1987.

[18] D. Cock, Q. Ge, T. Murray, and G. Heiser. The Last Mile: An Empirical Study of Some Timing Channels on seL4. In *ACM Conference on Computer and Communications Security*, pages 570–581, Scottsdale, AZ, USA, nov 2014.

[19] J. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, and S. Smith. Building the IBM 4758 Secure Coprocessor. *Computer*, 34(10):57–66, Oct 2001.

[20] A. Edwards, T. Jaeger, and X. Zhang. Runtime Verification of Authorization Hook Placement for the Linux Security Modules Framework. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS'02, 2002.

[21] J. Epstein and J. Picciotto. Trusting X: Issues in Building Trusted X Window Systems -or- WhatâĂŹs Not Trusted About X. In *Proceedings of the 14th Annual National Computer Security Conference*, 1991.

[22] J. Epstein and M. Shugerman. A Trusted X Window System Server for Trusted Mach. In *USENIX MACH Symposium*, pages 141–156, 1990.

[23] S. Even, O. Goldreich, and S. Micali. On-line/off-line Digital Signatures. In *Proceedings on Advances in cryptology*, CRYPTO '89, pages 263–275, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[24] I. T. Foster, J.-S. Vöckler, M. Wilde, and Y. Zhao. Chimera: AVirtual Data System for Representing, Querying, and Automating Data Derivation. In *Proceedings of the 14th Conference on Scientific and Statistical Database Management*, SSDBM'02, July 2002.

[25] V. Ganapathy, T. Jaeger, and S. Jha. Automatic placement of authorization hooks in the linux security modules framework. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 330–339, New York, NY, USA, 2005. ACM.

[26] C.-z. Gao and Z.-a. Yao. A Further Improved Online/Offline Signature Scheme. *Fundam. Inf.*, 91:523–532, August 2009.

[27] A. Gehani, B. Baig, S. Mahmood, D. Tariq, and F. Zaffar. Fine-grained Tracking of Grid Infections. In *Proceedings of the 11th IEEE/ACM International Conference on Grid Computing*, GRID'10, Oct 2010.

[28] A. Gehani and U. Lindqvist. Bonsai: Balanced Lineage Authentication. In *Proceedings of the 23rd Annual Computer Security Applications Conference*, ACSAC'07, Dec 2007.

[29] A. Gehani and D. Tariq. SPADE: Support for Provenance Auditing in Distributed Environments. In *Proceedings of the 13th International Middleware Conference*, Middleware '12, Dec 2012.

[30] S. Gueron and V. Krasnov. Speed Up Big-Number Multiplication Using Single Instruction Multiple Data (SIMD) Architectures, June 7 2012. US Patent App. 13/491,141.

[31] C. Hanson. SELinux and MLS: Putting The Pieces Together. In *In Proceedings of the 2nd Annual SELinux Symposium*, 2006.

[32] R. Hasan, R. Sion, and M. Winslett. The Case of the Fake Picasso: Preventing History Forgery with Secure Provenance. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies*, FAST'09, San Francisco, CA, USA, Feb. 2009.

[33] B. Hicks, S. Rueda, L. St.Clair, T. Jaeger, and P. McDaniel. A Logical Specification and Analysis for SELinux MLS Policy. *ACM Trans. Inf. Syst. Secur.*, 13(3):26:1–26:31, July 2010.

[34] T. Jaeger, A. Edwards, and X. Zhang. Consistency Analysis of Authorization Hook Placement in the Linux Security Modules Framework. *ACM Trans. Inf. Syst. Secur.*, 7(2):175–205, May 2004.

[35] T. Jaeger, R. Sailer, and U. Shankar. PRIMA: Policy-reduced Integrity Measurement Architecture. In *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies*, SACMAT '06, pages 19–28, New York, NY, USA, 2006. ACM.

[36] S. Kent and R. Atkinson. RFC 2406: IP Encapsulating Security Payload (ESP). 1998.

[37] D. Kilpatrick, W. Salamon, and C. Vance. Securing the X Window System with SELinux. Technical report, Jan. 2003.

[38] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

[39] H. Luhn. Computer for Verifying Numbers, Aug. 23 1960. US Patent 2,950,048.

[40] J. Lyle and A. Martin. Trusted Computing and Provenance: Better Together. In *2nd Workshop on the Theory and Practice of Provenance*, TaPP'10, Feb. 2010.

[41] P. Macko and M. Seltzer. A General-purpose Provenance Library. In *4th Workshop on the Theory and Practice of Provenance*, TaPP'12, June 2012.

[42] P. McDaniel, K. Butler, S. McLaughlin, R. Sion, E. Zadok, and M. Winslett. Towards a Secure and Efficient System for End-to-End Provenance. In *Proceedings of the 2nd conference on Theory and practice of provenance*, San Jose, CA, USA, Feb. 2010. USENIX Association.

[43] K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-Aware Storage Systems. In *Proceedings of the 2006 USENIX Annual Technical Conference*, 2006.

[44] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. Maclean, D. Margo, M. Seltzer, and R. Smogor. Layering in Provenance Systems. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, ATC'09, June 2009.

[45] D. Nguyen, J. Park, and R. Sandhu. Dependency Path Patterns As the Foundation of Access Control in Provenance-aware Systems. In *Proceedings of the 4th USENIX Conference on Theory and Practice of Provenance*, TaPP'12, pages 4–4, Berkeley, CA, USA, 2012. USENIX Association.

[46] Q. Ni, S. Xu, E. Bertino, R. Sandhu, and W. Han. An Access Control Language for a General Provenance Model. In *Secure Data Management*, Aug. 2009.

[47] J. Park, D. Nguyen, and R. Sandhu. A Provenance-Based Access Control Model. In *Proceedings of the 10th Annual International Conference on Privacy, Security and Trust (PST)*, pages 137–144, 2012.

[48] D. Pohly, S. McLaughlin, P. McDaniel, and K. Butler. Hi-Fi: Collecting High-Fidelity Whole-System Provenance. In *Proceedings of the 2012 Annual Computer Security Applications Conference*, ACSAC '12, Orlando, FL, USA, 2012.

[49] J. Postel. RFC 791: Internet protocol. 1981.

[50] A. C. Revkin. Hacked E-mail is New Fodder for Climate Dispute. *New York Times*, 20, 2009.

[51] T. Ristenpart, E. Tromer, H. Shacham, and S. 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'09)*, pages 199–212, Chicago, IL, USA, Oct. 2009. ACM.

[52] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association.

[53] C. Sar and P. Cao. Lineage File System. http://crypto.stanford.edu/~cao/lineage.html.

[54] J. Seibert, G. Baah, J. Diewald, and R. Cunningham. Using Provenance To Expedite MAC Policies (UPTEMPO) (Previously Known as IPDAM). Technical Report USTC-PM-015, MIT Lincoln Laboratory, October 2014.

[55] A. Shamir and Y. Tauman. Improved Online/Offline Signature Schemes. In J. Kilian, editor, *Advances in Cryptology — CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 355–367. Springer Berlin / Heidelberg, 2001.

[56] D. Tariq, B. Baig, A. Gehani, S. Mahmood, R. Tahir, A. Aqil, and F. Zaffar. Identifying the Provenance of Correlated Anomalies. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, Mar. 2011.

[57] The Netfilter Core Team. The Netfilter Project: Packet Mangling for Linux 2.4. http://www.netfilter.org/, 1999.

[58] J. Widom. Trio: A System for Integrated Management of Data, Accuracy, and Lineage. Technical Report 2004-40, Stanford InfoLab, Aug. 2004.

[59] World Wide Web Consortium. PROV-Overview: An Overview of the PROV Family of Documents. `http://www.w3.org/TR/prov-overview/`, 2013.

[60] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux Security Module Framework. In *Ottawa Linux Symposium*, page 604, 2002.

[61] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, pages 17–31, Berkeley, CA, USA, 2002. USENIX Association.

[62] Y. Xie, K.-K. Muniswamy-Reddy, D. Feng, Y. Li, and D. D. E. Long. Evaluation of a Hybrid Approach for Efficient Provenance Storage. *Trans. Storage*, 9(4):14:1–14:29, Nov. 2013.

[63] Y. Xie, K.-K. Muniswamy-Reddy, D. D. E. Long, A. Amer, D. Feng, and Z. Tan. Compressing Provenance Graphs, June 2011.

[64] X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for Static Analysis of Authorization Hook Placement. In *Proceedings of the 11th USENIX Security Symposium*, 2002.

[65] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr. Secure Network Provenance. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP'11, Oct. 2011.

[66] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao. Efficient Querying and Maintenance of Network Provenance at Internet-Scale. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, June 2010.

Securing Self-Virtualizing Ethernet Devices

Igor Smolyar Muli Ben-Yehuda Dan Tsafrir
Technion – Israel Institute of Technology
{igors,muli,dan}@cs.technion.ac.il

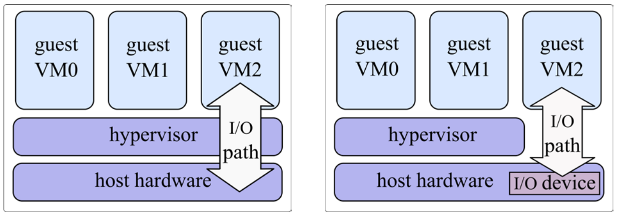
Abstract

Single root I/O virtualization (SRIOV) is a hardware/software interface that allows devices to “self virtualize” and thereby remove the host from the critical I/O path. SRIOV thus brings near bare-metal performance to untrusted guest virtual machines (VMs) in public clouds, enterprise data centers, and high-performance computing setups. We identify a design flaw in current Ethernet SRIOV NIC deployments that enables untrusted VMs to completely control the throughput and latency of other, unrelated VMs. The attack exploits Ethernet ”pause” frames, which enable network flow control functionality. We experimentally launch the attack across several NIC models and find that it is effective and highly accurate, with substantial consequences if left unmitigated: (1) to be safe, NIC vendors will have to modify their NICs so as to filter pause frames originating from SRIOV instances; (2) in the meantime, administrators will have to either trust their VMs, or configure their switches to ignore pause frames, thus relinquishing flow control, which might severely degrade networking performance. We present the Virtualization-Aware Network Flow Controller (VANFC), a software-based SRIOV NIC prototype that overcomes the attack. VANFC filters pause frames from malicious virtual machines without any loss of performance, while keeping SRIOV and Ethernet flow control hardware/software interfaces intact.

1 Introduction

A key challenge when running untrusted virtual machines is providing them with efficient and secure I/O. Environments running potentially untrusted virtual machines include enterprise data centers, public cloud computing providers, and high-performance computing sites.

There are three common approaches to providing I/O services to guest virtual machines: (1) the hypervisor *emulates* a known device and the guest uses an unmodified driver to interact with it [71]; (2) a *paravirtual driver* is installed in the guest [20, 69]; (3) the host assigns a real device to the guest, which then controls the device directly [22, 52, 64, 74, 76]. When emulating a device or using a paravirtual driver, the hypervisor intercepts all interactions between the guest and the I/O device, as shown in Figure 1a, leading to increased overhead and significant performance penalty.



(a) Traditional Virtualization (b) Direct I/O Device Assignment

Figure 1: Types of I/O Virtualization

The hypervisor can reduce the overhead of device emulation or paravirtualization by assigning I/O devices directly to virtual machines, as shown in Figure 1b. Device assignment provides the best performance [38,53,65,76], since it minimizes the number of I/O-related world switches between the virtual machine and its hypervisor. However, assignment of standard devices is not scalable: a single host can generally run an order of magnitude more virtual machines than it has physical I/O device slots available.

One way to reduce I/O virtualization overhead further and improve virtual machine performance is to offload I/O processing to scalable self-virtualizing I/O devices. The PCI Special Interest Group (PCI-SIG) on I/O Virtualization proposed the Single Root I/O Virtualization (SRIOV) standard for scalable device assignment [60]. PCI devices supporting the SRIOV standard present themselves to host software as multiple virtual interfaces. The host can assign each such partition directly to a different virtual machine. With SRIOV devices, virtual machines can achieve bare-metal performance even for the most demanding I/O-intensive workloads [38, 39]. We describe how SRIOV works and why it improves performance in Section 2.

New technology such as SRIOV often provides new capabilities but also poses new security challenges. Because SRIOV provides untrusted virtual machines with unfettered access to the physical network, such machines can inject malicious or harmful traffic into the network. We analyze the security risks posed by using SRIOV in environments with untrusted virtual machines in Section 3. We find that SRIOV NIC, as currently deployed, suffers from a major design flaw and cannot be used securely together with network flow control.

We make two contributions in this paper. The **first contribution** is to show how a malicious virtual machine with access to an SRIOV device can use the Ethernet flow control functionality to attack and completely control the bandwidth and latency of other unrelated VMs using the same SRIOV device, without their knowledge or cooperation. The malicious virtual machine does this by transmitting a small number of Ethernet pause or Priority Flow Control (PFC) frames on its host's link to the edge switch. If Ethernet flow control is enabled, the switch will then shut down traffic on the link for a specified amount of time. Since the link is shared between multiple untrusted guests and the host, none of them will receive traffic. The details of this attack are discussed in Section 4. We highlight and experimentally evaluate the most notable ramifications of this attack in Section 5.

Our **second contribution** is to provide an understanding of the fundamental cause of the design flaw leading to this attack and to show how to overcome it. We present and evaluate (in Section 6 and Section 7) the Virtualization-Aware Network Flow Controller (VANFC), a software-based prototype of an SRIOV NIC that successfully overcomes the described attack without any loss in performance.

With SRIOV, a single physical endpoint includes both the host (usually trusted) and multiple untrusted guests, all of which share the same link to the edge switch. The edge switch must either trust all the guests and the host or trust none of them. The former leads to the flow control attack we show; the latter means doing without flow control and, consequently, giving up on the performance and efficient resource utilization flow control provides.

With SRIOV NICs modeled after VANFC, cloud users could take full advantage of lossless Ethernet in SRIOV device assignment setups without compromising their security. By filtering pause frames generated by the malicious virtual machine, VANFC keeps these frames from reaching the edge switch. The traffic of virtual machines and host that share the same link remains unaffected; thus VANFC is 100% effective in eliminating the attack. VANFC has no impact on throughput or latency compared to the baseline system not under attack.

VANFC is fully backward compatible with the current hardware/software SRIOV interface and with the Ethernet flow control protocol, with all of its pros and cons. Controlling Ethernet flow by pausing physical links has its fundamental problems, such as link congestion propagation, also known as the ”congestion spreading” phenomenon [13]. The attack might also be prevented by completely redesigning the Ethernet flow control mechanism, making it end-to-end credit-based, as in InfiniBand [18], for example. But such a pervasive approach is not practical to deploy and remains outside the scope of this work. Instead, VANFC specifically targets the design flaw in SRIOV NICs that enables the attack. VANFC prevents the attack without any loss of performance and without requiring any changes to either Ethernet flow control or to the SRIOV hardware/software interfaces.

One could argue that flow control at the Ethernet level is not necessary, since protocols at a higher level (e.g., TCP) have their own flow control. We show why flow control is required for high performance setups, such as those using Converged Enhanced Ethernet, in Section 8.

In Section 9 we provide some notes on the VANFC implementation and on several aspects of VM-to-VM traffic security. We present related work in Section 10. We offer concluding remarks on SRIOV security as well as remaining future work in Section 11.

2 SRIOV Primer

Hardware emulation and paravirtualized devices impose a significant performance penalty on guest virtual machines [15, 16, 21, 22, 23]. Seeking to improve virtual I/O performance and scalability, PCI-SIG proposed the SRIOV specification for PCIe devices with self-virtualization capabilities. The SRIOV spec defines how host software can partition a single SRIOV PCIe device into multiple PCIe “virtual” devices.

Each SRIOV-capable physical device has at least one Physical Function (PF) and multiple virtual partitions called Virtual Functions (VFs). Each PF is a standard PCIe function: host software can access it as it would any other PCIe device. A PF also has a full configuration space. Through the PF, host software can control the entire PCIe device as well as perform I/O operations. Each PCIe device can have up to eight independent PFs.

VFs, on the other hand, are “lightweight” (virtual)

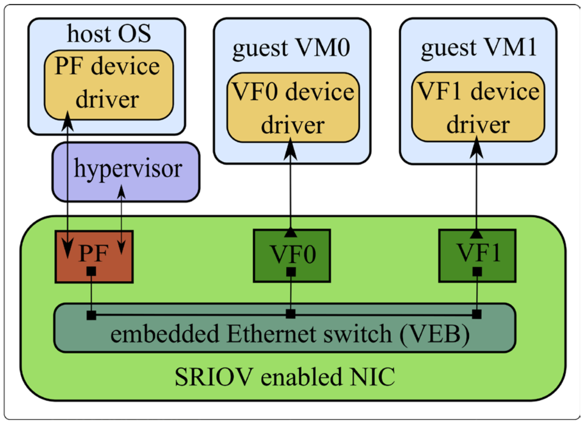


Figure 2: SRIOV NIC in a virtualized environment

PCIe functions that implement a subset of standard PCIe device functionalities. Virtual machines driving VFs perform only I/O operations through them. For a virtual machine to use a VF, the host software must configure that VF and assign it to the virtual machine. Host software often configures a VF through its PF. VFs have a partial configuration space and are usually presented to virtual machines as PCIe devices with limited capabilities. In theory, each PF can have up to 64K VFs. Current Intel implementations of SRIOV enable up to 63 VFs per PF [42] and Mellanox ConnectX adapters usually have 126 VFs per PF [57].

While PFs provide both control plane functionality and data plane functionality, VFs provide only data plane functionality. PFs are usually controlled by device drivers that reside in the trusted, privileged, host operating system or hypervisor. As shown in Figure 2, in virtualized environments each VF can be directly assigned to a VM using device assignment, which allows each VM to directly access its corresponding VF, without hypervisor involvement on the I/O path.

Studies show that direct assignment of VFs provides virtual machines with nearly the same performance as direct assignment of physical devices (without SRIOV) while allowing the same level of scalability as software-based virtualization solutions such as device emulation or paravirtualization [33, 38, 41, 77]. Furthermore, two VMs that share the same network device PF can communicate efficiently since their VM-to-VM traffic can be switched in the network adapter. Generally, SRIOV devices include embedded Ethernet switch functionality capable of efficiently routing traffic between VFs, reducing the burden on the external switch. The embedded switch in SRIOV capable devices is known as a Virtual Ethernet Bridge (VEB) [51].

SRIOV provides virtual machines with I/O performance and scalability that is nearly the same as bare metal. Without SRIOV, many use cases in cloud computing, high-performance computing (HPC) and enterprise data centers would be infeasible. With SRIOV it is possible to virtualize HPC setups [24, 37]. In fact, SRIOV is considered the key enabling technology for fully virtualized HPC clusters [54]. Cloud service providers such as Amazon Elastic Compute Cloud (EC2) use SRIOV as the underlying technology in EC2 HPC services. Their Cluster Compute-optimized virtual machines with high performance enhanced networking rely on SRIOV [2]. SRIOV is important in traditional data centers as well. Oracle, for example, created the Oracle Exalogic Elastic Cloud, an integrated hardware and software system for data centers. Oracle Exalogic uses SRIOV technology to share the internal network [40].

3 Analyzing SRIOV Security

Until recently, organizations designed and deployed Local Area Networks (LANs) with the assumption that each end-station in the LAN is connected to a dedicated port of an access switch, also known as an edge switch.

The edge switch applies the organization's security policy to this dedicated port according to the level of trust of the end-station connected to the port: some machines and the ports they connect to are trusted and some are not. But given a port and the machine connected to it, the switch enforcing security policy must know how trusted that port is.

With the introduction of virtualization technology, this assumption of a single level of trust per port no longer holds. In virtualized environments, the host, which is often a trusted entity, shares the same physical link with untrusted guest VMs. When using hardware emulation or paravirtualized devices, the trusted host can intercept and control all guest I/O requests to enforce the relevant security policy. Thus, from the point of view of the network, the host makes the port trusted again.

Hardware vendors such as Intel or Mellanox implement strict VF management or configuration access to SRIOV devices. Often they allow VFs driven by untrusted entities to perform only a limited set of management or configuration operations. In some implementations, the VF performs no such operations; instead, it sends requests to perform them to the PF, which does so after first validating them.

On the data path, the situation is markedly different. SRIOV's raison d'être is to avoid host involvement on the data path. Untrusted guests with directly assigned VFs perform data path operations—sending and receiving network frames—directly against the device. Since the device usually has a single link to the edge switch, the device aggregates all traffic, both from the trusted host and from the untrusted guests, and sends it on the single shared link. As a result, untrusted guests can send any network frames to the edge switch.

Giving untrusted guests uncontrolled access to the edge switch has two implications. First, since the edge switch uses its physical resources (CAM tables, queues, processing power) to process untrusted guests' traffic, the switch becomes vulnerable to various denial of service attacks. Second, sharing the same physical link between trusted and untrusted entities exposes the network to many Ethernet data-link layer network attacks such as Address Resolution Protocol (ARP) poisoning, Media Access Control (MAC) flooding, ARP spoofing, MAC address spoofing, and Spanning Tree Protocol (STP) attacks [14, 17, 47, 56, 73, 75]. Therefore, the edge switch must never trust ports connected to virtualized hosts with an SRIOV device.

Although the problem of uncontrolled access of untrusted end-points is general to Ethernet networks, using SRIOV devices imposes additional limitations. As we will see in the next few subsections, not trusting the port sometimes means giving up the required functionality.

3.1 Traditional Lossy Ethernet

Traditional Ethernet is a lossy protocol; it does not guarantee that data injected into the network will reach its destination. Data frames can be dropped for different reasons: because a frame arrived with errors or because a received frame was addressed to a different end-station. But most data frame drops happen when the receiver's buffers are full and the receiving end-station has no memory available to store incoming data frames. In the original design of the IEEE 802.3 Ethernet standard, reliability was to be provided by upper-layer protocols, usually TCP [63], with traditional Ethernet networks providing best effort service and dropping frames whenever congestion occurs.

3.2 Flow Control in Traditional Ethernet

Ethernet Flow Control (FC) was proposed to control congestion and create a lossless data link medium. FC enables a receiving node to signal a sending node to temporarily stop data transmission. According to the IEEE 802.3x standard [6], this can be accomplished by sending a special Ethernet pause frame. The IEEE 802.3x pause frame is defined in Annex 31B of the spec [9] and uses the MAC frame format to carry pause commands.

link speed, Gbps	single frame pause time, ms	frame rate required to stop transmission, frames/second
1	33.6	30
10	3.36	299
40	0.85	1193

Table 1: Pause frame rate for stopping traffic completely

When a sender transmits data faster than the receiver can process it and the receiver runs out of space, the receiver sends the sender a MAC control frame with a pause request. Upon receiving the pause frame, the sender stops transmitting data.

The pause frame includes information on how long to halt the transmission. The `pause_time` is a two byte MAC Control parameter in the pause frame that is measured in units of `pause_quanta`. It can be between 0 to 65535 `pause_quanta`. The receiver can also tell the sender to resume transmission by sending a special pause frame with the `pause_time` value set to 0.

Each `pause_quanta` equals 512 "bit times," defined as the time required to eject one bit from the NIC (i.e., 1 divided by the NIC speed). The maximal pause frame `pause_time` value can be 65535 `pause_quanta`, which is $65535 \times 512 = 33.6$ million bit times.

For 1Gbps networks, one pause frame with `pause_time` value of 65535 `pause_quanta` will tell the sender to stop transmitting for 33.6 million bit times, i.e., 33.6 ms. A sender operating at 10 Gbps speed will pause for 3.36 ms. A sender operating at 40 Gbps speed will pause for 0.85 ms.

Table 1 shows the rate at which a network device should receive pause frames to stop transmission completely. The `pause_time` value of each frame is 0xFFFF. Sending 30 pause frames per second will tell the sender to completely stop transmission on a 1Gbps link. For a sender operating at 10 Gbps speed to stop transmission requires sending 299 frames/second. For a sender operating at 40 Gbps speed to stop transmission requires sending 1193 frames/second.

3.3 Priority Flow Control in Ethernet

To improve the performance and reliability of Ethernet and make it more suitable for data centers, the IEEE 802.1 working group proposed a new set of standards, known as Data Center Bridging (DCB) or Converged Enhanced Ethernet (CEE).

In addition to the IEEE 802.3x Ethernet pause, the new IEEE 802.1Qbb standard proposed to make Ethernet truly “lossless” in data center environments by adding Priority-based Flow Control (PFC) [8].

Similar to the 802.3x FC, PFC is a link-level flow control mechanism, but it is implemented on a per traffic-class basis. While 802.3x FC pauses all traffic on the link, PFC makes it possible to pause a specific class of traffic using the same pause frame structure. PFC operates on individual traffic classes, as defined by Annex I of the IEEE 802.1Q standard [7]. Up to 8 traffic classes can be defined for PFC per link.

3.4 Attacking VMs via Flow Control

Direct device assignment enables malicious guests to attack the Ethernet network via well-known Layer 2 attacks [14, 17, 47, 56, 73, 75]. Even when using virtualization-aware switching extensions such as the Virtual Edge Port Aggregator (VEPA) [30, 31], all guests with direct access to the VFs of the same PF still share the same physical link to the edge switch, and the edge switch still allocates processing resources per link.

Since both 802.3x and 802.1Qbb perform flow control on a link-level basis, and the link is shared between VMs, any flow control manipulation by a single VM will affect the PF and all VFs associated with this PF. This means that a malicious VM is capable of controlling the bandwidth and latency of all VMs that share the same adapter.

The malicious VM can pause all traffic on the link by sending 802.3x pause frames and can stop a specific traffic class by sending 802.1Qbb pause frames. To stop all traffic on a 10 Gbps Ethernet link, an attacker needs to transmit pause frames at a rate of 300 frames/second, which is about 155 Kbps of bandwidth. The attacker can fully control the bandwidth and latency of all tenant VMs with minimal required resources and without any cooperation from the host or from other guest VMs.

4 Attack Evaluation

4.1 Experimental Setup

We constructed a lab setup in which we perform and evaluate the flow-control attack described in the previous section. We use a Dell PowerEdge R420 server, which is a dual socket with six cores per socket, with Intel Xeon E5-2420 CPUs running at 1.90GHz. The chipset is the Intel C600 series. The server includes 16GBs of memory and an SRIOV-capable Intel NIC (10GbE 82599 or 1GbE I350) installed in PCIe generation 3 slots with two VFs enabled.

We use the KVM Hypervisor [50] and Ubuntu server 13.10 with 3.11.0 x86_64 kernel for the host, guest VMs, and the client. Each guest is created with 2GBs of memory, two virtual CPUs, and one VF directly assigned to it. Client and host machines are identical servers connected to the same dedicated switch, as shown in Figure 3.

To achieve consistent results, the server’s BIOS profile is performance optimized, all power optimizations are tuned off, and Non-Uniform Memory Access (NUMA) is enabled. The guest virtual CPUs are pinned to the cores on the same NUMA node to which the Intel PF is connected. The host allocates to the guest memory from the same NUMA node as well.

For our 1GbE environment, we use an Intel Ethernet I350-T2 network interface connected to a Dell PowerConnect 6224P 1Gb Ethernet switch. For our 10GbE environment, we use an Intel 82599 10 Gigabit TN network interface connected to an HP 5900AF 10Gb Ethernet switch.

Host and client use their distribution’s default drivers with default configuration settings. Guest VMs use version 2.14.2 of the `ixgbevf` driver for the Intel 10G 82599 Ethernet controller virtual function and the default `igbvf` version 2.0.2-k for the Intel 1G I350 Ethernet controller virtual function. Ethernet flow control IEEE 802.3x is enabled on switch ports. We set the Ethernet Maximal Transfer Unit (MTU) to 1500 bytes on all Ethernet switches and network interfaces in our tests.

4.2 Benchmark Methodology

We conduct a performance evaluation according to the methodology in RFC 2544 [25]. For throughput tests, we use an Ethernet frame size of 1518 bytes and measure maximal throughput without packet loss. Each throughput test runs for at least 60 seconds and we take the average of 5 test cycles. To measure latency, we use 64 and 1024 byte messages. Each latency test runs at least 120 seconds and we measure the average of at least 15 test cycles. (While RFC 2544 dictates running 20 cycles, we obtained plausible results after 15 cycles; thus, we decided to reduce test runtime by running each test only 15 cycles.)

Benchmark Tools: We measure throughput and latency with two well-known network benchmark utilities: `iperf` [3] and `netperf` [45]. We use the `iperf` TCP stream test to measure throughput and the `netperf` TCP_RR test to measure latency. The `iperf` and `netperf` clients are run on the client machine, while

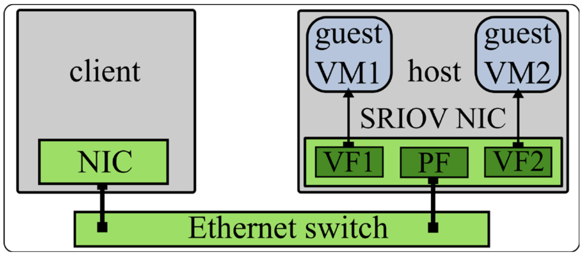


Figure 3: Setup scheme

the iperf and netperf servers are run on VM1. We measure on the client the bandwidth and latency from the client to VM1.

Traffic Generators: In addition to the traffic generated by the benchmark tools, we use tcpdump [44] to capture traffic and tcpreplay [5] to send previously captured and modified frames at the desired rate.

Testbed Scheme: The testbed scheme is shown in Figure 3. Our testbed consists of two identical servers, one acting as client and the other as the host with SRIOV capable NIC. We configure two VFs on the host's SRIOV PF. We assign VF1 to guest VM1 and VF2 to guest VM2. Client and host are connected to the same Ethernet switch. We generate traffic between VM1 and the client using iperf and netperf. VM2 is the attacking VM.

4.3 Flow-Control Attack Implementation

We use tcpreplay [5] to send specially crafted 802.3x pause frames at the desired rate from the malicious VM2.[1] When the switch receives a pause frame from VM2, it inhibits transmission of any traffic on the link between the switch and the PF, including the traffic between the client and VM1, for a certain number of pause_time quanta. Sending pause frames from VM2, we can manipulate the bandwidth and latency of the traffic between VM1 and the client. The value of pause_time of each pause frame is 0xFFFF pause_quanta units. Knowing the link speed, we can calculate the pause frame rate, as described in Section 3, and impose precise bandwidth limits and latency delays on VM1. The results of the attack in both 1GbE and 10GbE environments are presented in Section 4.4.

4.4 Attack Results

Figures 4 and 5 show the results of the pause frame attack on victim throughput in the 1GbE and 10GbE environments respectively. Figures 4a and 5a show victim (VM1) throughput under periodic attack of VM2. Every 10 seconds, VM2 transmits pause frames for 10 seconds at 30 frames/second (as shown in Figure 4a) and at 300 frames/second (as shown in Figure 5a). In this test we measure the throughput of the victim system, VM1. The figures clearly show that VM2 can gain complete control over VM1 throughput: starting from the tenth second, the attacker completely stops traffic on the link for ten seconds.

Figure 6 shows the results of the pause frame attack on victim latency in the 10GbE environment. Figure 6a shows victim latency under the same periodic attack described above. In this test we use 64B and 1024B messages. For better result visualization, we lowered the attack rate to 150 pause frames/second. Figure 6a shows that the attacker can increase victim latency to 250% by running the attack at a rate of only 150 frames/second.

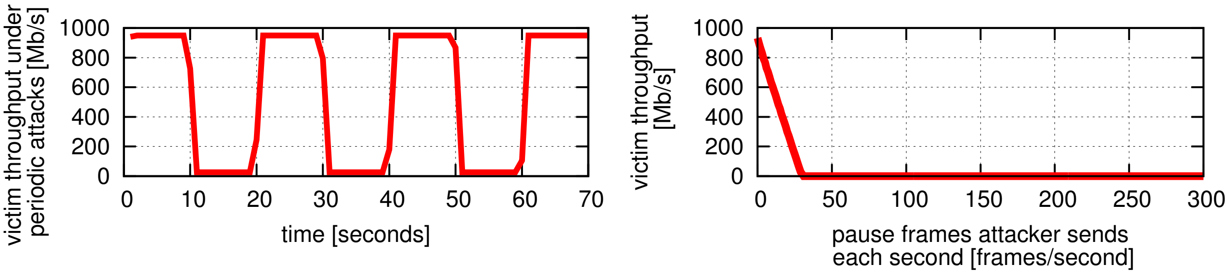
Victim throughput Figures 4b and 5b display throughput of VM1 as a function of the rate of pause frames VM2 sends. From Figure 4b we can see that VM2 can pause all traffic on the 1GbE link with almost no effort, by sending pause frames at a rate of 30 frames/second. For the 10GbE link, VM2 needs to work a little bit harder and raise its rate to 300 frames/second. This test's results confirm the calculations shown in Table 1. Figures 7a and 7b confirm that the measured victim throughput is exactly as predicted. In other words, it is easily and completely controlled by the attacker.

These tests show that a malicious VM can use the pause frame attack to control the throughput of other VMs with precision. Furthermore, we see that the pause frame attack requires minimal effort from the attacker and will be hard to detect amid all the other network traffic. To halt all transmissions on the 10GbE link, the attacker only needs to send 64B pause frames at 300 frames/second. 300 frames/second is approximately 0.002% of the 14.88 million frames/second maximum frame rate for 10GbE.[2] Discovering such an attack can be quite challenging, due to the low frame rate involved, especially on a busy high-speed link such as 10GbE or 40GbE.

Victim latency Figure 6b shows the victim's latency as a function of the attacker's pause frame rate. In this test we measure the latency of 64 byte messages and 1024 byte messages. We see that the figures for both 64B

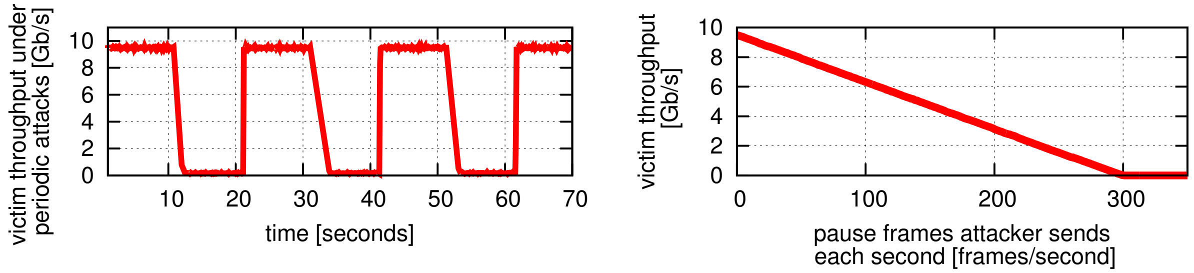
[1] We use 802.3x pause frames for the sake of simplicity, but we could have used PFC frames instead. PFC uses exactly the same flow control mechanism and has the same MAC control frame format. The only difference between PFC frames and pause frames is the addition of seven pause_time fields in PFC that are padded in 802.3x frames.

[2] The maximum frame rate equals the link speed divided by the sum of sizes of the preamble, frame length and inter-frame gap.



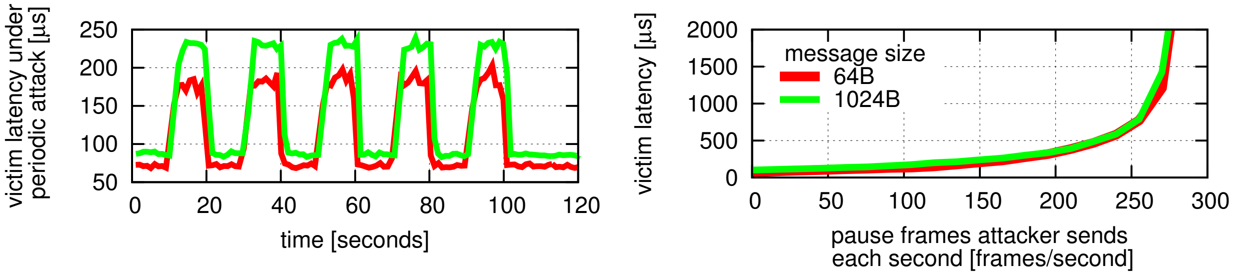
(a) (b)

Figure 4: Pause frame attack: victim throughput in 1GbE environment



(a) (b)

Figure 5: Pause frame attack: victim throughput in 10GbE environment



(a) (b)

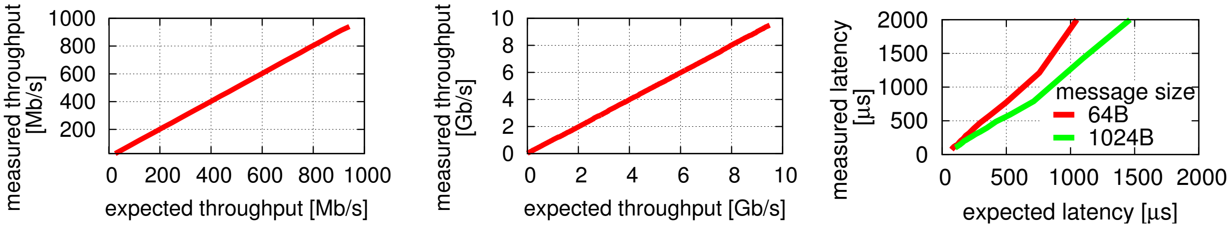
Figure 6: Pause frame attack: victim latency in 10GbE environment

and 1024B are barely distinguishable and almost converge; the latency is the same for small and large size messages under attack.

In Figure 7c we see that measured latency and expected latency differ somewhat. In practice, this difference means that an attacker can control the victim's latency with slightly less precision than it can control its throughput, but it can still control both with high precision and relatively little effort.

In back-to-back configuration, without a switch, latency behaves as expected. We believe this difference is caused by the switch's internal buffering methods—in addition to storing frames internally, the Ethernet switch prevents the possible side effects of such buffering e.g., head-of-line blocking [70] and congestion spreading [13]. To accurately explain this phenomenon, we need access to the switch internals; unfortunately, the Ethernet switch uses proprietary closed software and hardware.

Experiments with Non-Intel Devices We performed an identical experiment on same setup with an SRIOV Broadcom NetXtreme II BCM57810 10GbE NIC [26] and got the same results. Our attack is valid for this NIC as well.



(a)

(b)

(c)

Figure 7: Pause frame attack: expected vs. measured throughput and latency

We also tried the attack described above on another vendor's 40GbE SRIOV adapter. Whenever the attacking VM transmitted MAC control frames (pause frames) through its VF, the adapter completely locked up and became unresponsive. It stopped generating both transmit and receive interrupts, and required manual intervention to reset it, by reloading the PF driver on the host. This lockup appears to be a firmware issue and has been communicated to the adapter vendor.

Clearly, with this adapter and this firmware issue, a malicious VM could trivially perform a straightforward denial of service attack against its peer VMs that use this adapter's VFs and against the host. But since this attack is trivial to discover, we focus instead on the stealthier pause frame attack, which is much harder to discover and protect against.

5 Attack Ramifications

The consequences of the attack are substantial. If Ethernet flow control is enabled on the SRIOV device, the host's VMs' security is compromised and the VM's are susceptible to the attack.

The attack cannot be prevented using the filtering capabilities of currently available SRIOV Ethernet devices due to their minimal filtering capability. At best, modern SRIOV NICs are capable of enforcing anti-spoofing checks based on the source MAC address or VLAN tag of the VM, to prevent one VM from pretending to be another. In the attack we describe, the adversary generates flow control frames with the malicious VM's source MAC and VLAN tag, so anti-spoofing features cannot block the attack.

Since the attack cannot be prevented with current NICs and switches, cloud providers must either be content with flawed security and fully trust the guest VMs or disable the Ethernet flow control in their networks. Neither option is palatable. The former is unrealistic for the public cloud and unlikely be acceptable to private cloud providers. The latter means giving up the performance benefits of lossless Ethernet, increasing overall resource utilization, and reducing performance. We discuss in greater detail the performance advantages that Ethernet flow control provides in Section 8.

6 Improving SRIOV Security

The attack described in the previous sections is the result of a fundamental limitation of SRIOV: from the network point of view, VFs and their associated untrusted VMs are all lumped together into a single end-station. To secure SRIOV and eliminate the attack while keeping flow control functionality, we propose to extend SRIOV Ethernet NIC filtering capability to filter traffic transmitted by VFs, not only on the basis of source MAC and VLAN tags—the method currently employed by anti-spoofing features—but also on the basis of the MAC destination and Ethernet type fields of the frame. This filtering cannot be done by the host without some loss of performance [39] and has to be done before traffic hits the edge switch. Hence it must be done internally in the SRIOV NIC. We built a software-based prototype of an SRIOV Ethernet NIC with pause frame filtering. Before presenting the prototype, we begin by describing the internals of an SRIOV NIC.

6.1 SRIOV NIC Internals

Figure 8a shows a detailed schema of an SRIOV Ethernet NIC. The SRIOV device is connected to the external adjacent Ethernet switch on the bottom side and to the host's PCIe bus, internal to the host, on the top side.

Switching The NIC stores frames it receives from the external switch in its internal buffer. The size of this buffer is on the order of hundreds of KBytes, depending on the NIC model: 144KB in Intel I350 [43] and 512KB in Intel 82599 [42]. After receiving a packet, the SRIOV NIC looks up the frame's MAC in its MAC address table, finds the destination VF according to the frame's destination MAC address, and copies the frame (using DMA) over the PCIe bus to the VF's buffer ring, which is allocated in the host's RAM. This is analogous to a standard Ethernet switch that receives a packet on an ingress port, looks up its MAC address, and chooses the right egress port to send it to. The data path of the frame is marked with a red dashed line in Figure 8a. In addition, SRIOV NIC is able to perform VM-to-VM switching internally, without sending the frames to the external switch.

Internal Buffer When Ethernet flow control is enabled, the SRIOV NIC starts monitoring its internal buffer. If the NIC cannot process received frames fast enough, for example due to an overloaded or slow PCIe link, the buffer fills up. Once it reaches a predefined threshold, the SRIOV NIC generates and sends pause frames to the external Ethernet switch. The switch then holds transmissions to the NIC for the requested time, storing these frames in its own internal buffers. While the switch is buffering frames, the NIC should continue copying the frames it buffered into the each VF's ring buffer, clearing up space in its internal buffer.

Ring Buffer The final destination for a received frame is in its VF's ring buffer, located in host RAM. The network stack in the VM driving the VF removes frames from its ring buffers at a rate that is limited by the CPU. If the VM does not get enough CPU cycles or is not efficient enough, the NIC may queue frames to the ring buffer faster than the VM can process them. When the ring buffer fills up, most Ethernet NICs (e.g., those of Intel's and Mellanox's) will simply drop incoming frames. Less commonly, a few NICs, such as Broadcom's NetXtreme II BCM57810 10GbE, can monitor each VF's ring buffer. When the ring buffer is exhausted, the NIC can send pause frames to the external switch to give the host CPU a chance to catch up with the sender. When available, this functionality is usually disabled by default.

Outbound Security Some SRIOV Ethernet NICs (e.g., Intel 82599 10GbE [42] or I350 1GbE [43] NICs) include anti-spoofing functionality. They can verify that the source MAC address and/or VLAN tag of each frame transmitted by the VF belongs to the transmitting VF. To this end, these NICs have an internal component that can inspect and even change frames transmitted from the VF. In addition, Intel SRIOV NICs have advanced inbound filtering capabilities, storm control, rate limiting, and port mirroring features, very much like any standard Ethernet switch.

As we can see, Ethernet SRIOV devices implement on-board a limited form of Ethernet switching. That is why such devices are also known as virtual Ethernet bridges (VEBs).

6.2 The VANFC design

The key requirement from VANFC is to filter outbound traffic transmitted by a VF. Ideally, VANFC would be implemented in a production SRIOV NIC. Unfortunately, all tested SRIOV NICs are proprietary with closed firmware. Furthermore, most frame processing logic is implemented in high speed ASIC hardware.

We opted instead to build VANFC as a software-based prototype of an SRIOV NIC that filters outbound traffic. VANFC takes advantage of the following two observations: `(1)` VEB embedded into the SRIOV NIC device replicates standard Ethernet switching behavior and can be considered as a virtual Ethernet switch; `(2)` all valid pause frames are generated by the NIC's hardware and have the PF's source MAC address, whereas invalid—malicious—pause frames are sent with source address of a VF. Should the adversary VM attempt to generate pause frames with the PF's source MAC address, the NIC's anti-spoofing will find and drop these frames.

In order to filter transmitted malicious pause frames, we first need to identify pause frames. In such frames the Ethernet type field is `0x8808` (MAC control type), the MAC opcode field is `0x0001` (pause opcode), and the destination MAC address is multicast `01-80-C2-00-00-01`. For any such packet, the VANFC filter should drop the frame if the source MAC is different than the PF's MAC address.

As mentioned previously, most SRIOV NICs already have a component that can filter outbound traffic; this component is a part of the SRIOV device's internal Ethernet switch and cannot be modified. Our prototype extends this switch in software by running the extension on the wire between the SRIOV NIC and the external switch.

Filtering Component For our Ethernet filtering device we use the standard Linux bridge configured on an x86-based commodity server running Ubuntu server 13.10 and equipped with two Intel 82599 10 Gigabit TN Network controllers installed in PCIe gen 2 slots. One NIC is connected to the host PF and the other is connected to the external Ethernet switch, as displayed in Figure 8b. Ethernet switching is performed by the Linux bridge [4] and filtering is done by the `ebtables` [32].

Performance model Bridge hardware is fast enough not to be a bottleneck for 10Gb Ethernet speed. However, by adding to the setup an Ethernet device implemented in software, we increased latency by a constant delay of approximately 55µs. An eventual implementa-

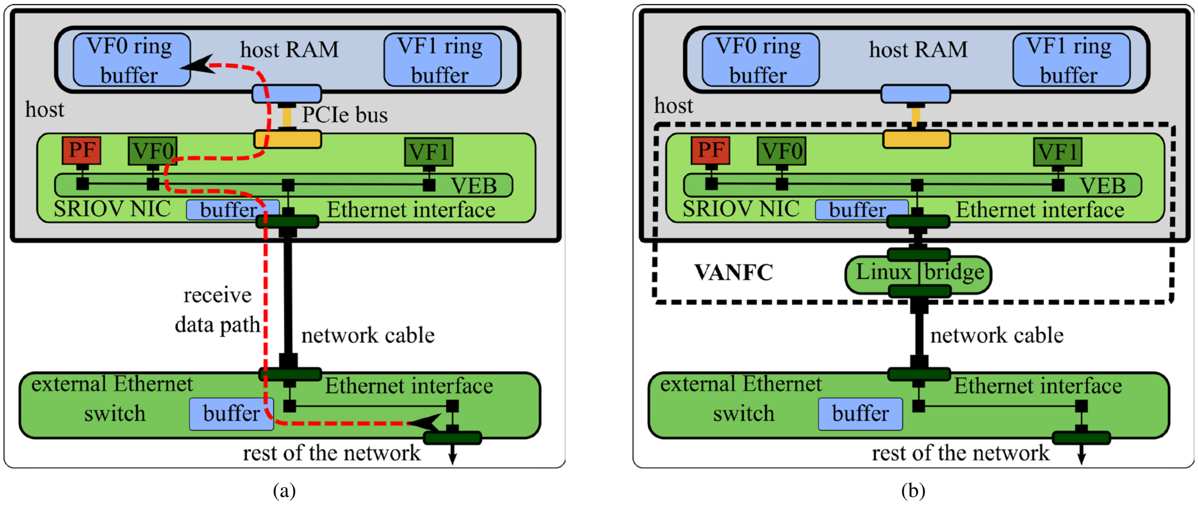


Figure 8: Fig. (a) shows schema of current SRIOV NIC internals; Fig. (b) shows VANFC schema.

tion of VANFC in hardware will eliminate this overhead; we therefore discount it in latency oriented performance tests.

We wanted to make this bridge completely transparent and not to interfere with passing traffic: the host should keep communicating with the external switch and the switch should keep communicating with the host as in the original setup without VANFC. VANFC should change neither the SRIOV software/hardware interface nor the Ethernet flow control protocol. To ensure this, we made a few modifications in Linux bridge code and in the Intel 82599 device driver used by the bridge device.

Bridge Modification The standard Ethernet bridge should not forward MAC control frames that are used to carry pause commands since MAC control frames are designed to be processed by Ethernet devices. Since we want the bridge to deliver all of the traffic between the SRIOV device and the external switch, including the pause frames sent by the PF, we modify the Linux bridging code to forward MAC control frames and use `ebtables` to filter pause frames not sent from the PF. Our experiments use a static configuration for `ebtables` and for the Linux bridge.

Device Driver Modification We use a modified `ixgbe` driver version 3.21.2 for Intel 10G 82599 network controllers on the bridge machine. According to the Intel 82599 controller data-sheet [42], the flow control mechanism of the device receives pause frames when flow control is enabled; otherwise the device silently drops pause frames. In our setup, we disable the flow control feature of Intel NICs installed in the bridge machine and we configure them to forward pause frames up to the OS, where they should be processed by the bridge and `ebtables`. We do this by enabling the Pass MAC Control Frames (PMCF) bit of the MAC Flow Control (MFLCN) register, as described in section 3.7.7.2 of the Intel 82599 data-sheet [42].

Ring Buffer Exhaustion As mentioned, some SRIOV devices are capable of monitoring a VF's ring buffer and automatically generating pause frames when it is exhausted. In such a scenario, pause frames will be generated with the source MAC address of the PF and will not be recognized by the VANFC. We argue that such pause frame generation should be disabled in **any** SRIOV based setup, regardless of whether the VMs are trusted. Since the VM fully controls the VF's ring buffer, a malicious VM can modify its software stack (e.g., the VF device driver) to manipulate the ring buffer so that the SRIOV device monitoring the ring buffer will generate pause frames on the VM's behalf. Such pause frames will reach the external switch, which will stop its transmissions to the host and other VMs, leaving us with the same attack vector.

Automatic generation of pause frames on VF ring buffer exhaustion is problematic even if all VMs are trusted. Consider, for example, a VM that does not have enough CPU resources to process all incoming traffic and exhausts the VF's ring buffer. Sending pause frames to the switch may help this VM process the buffer but will halt the traffic to other VMs. Thus, to keep the SRIOV device secure, an SRIOV NIC should not automatically send pause frames when the VF's ring buffer is exhausted regardless of whether the VM is trusted.

Nevertheless, monitoring VF ring buffers can be useful for keeping the Ethernet network lossless and avoiding dropped frames. We propose that the SRIOV device monitor ring buffers, but instead of automatically generating pause frames on ring buffer exhaustion, it should notify the hypervisor. The hypervisor, unlike the device, could then carefully consider whether the VM is malicious or simply slow. If the VM is simply slow, the hypervisor could give it a scheduling boost or assign more CPU resources to it, thereby giving it a chance to process its ring buffer before it fills up. We plan to explore this avenue in future work.

7 Evaluating VANFC

We evaluate VANFC in several scenarios. The **baseline** scenario includes an unprotected system, as shown in Figure 3, and no attack is performed during the test. In this scenario we measure the system's baseline throughput and latency. The **baseline system under attack** includes the same unprotected system but here VM2 runs the attack during the test, sending pause frames at a constant rate of 150 frames/sec. In this scenario we measure the effectiveness of the attack on an unprotected system.

In the **protected system** scenario, VANFC, shown in Figure 8b, replaces the unprotected system. In this scenario VM2 does not perform any attack during the test. We use this scenario to measure the performance overhead introduced by VANFC compared to the baseline. In the **protected system under attack** scenario, we also use VANFC, but here the attacker VM2 sends pause frames at a constant rate of 150 frames/sec. In this scenario we verify that VANFC indeed overcomes the attack.

We perform all tests on the 10GbE network with the same environment, equipment, and methodology as described in Section 4.1.

As explained in Section 6.2, to filter malicious pause frames, our solution uses a software-based filtering device, which adds constant latency of 55µs. A production solution would filter these frames in hardware, obviating this constant latency overhead of software-based model. Thus, in latency-oriented performance tests of the VANFC, we reduced 55µs from the results.

Evaluation Tests To evaluate the performance of the described scenarios, we test throughput and latency using `iperf` and `netperf`, as previously described.

In addition, we configure the `apache2` [34] web server on VM1 to serve two files, one sized 1KB and one sized 1MB. We use `apache2` version 2.4.6 installed from the Ubuntu repository with the default configuration. We run the `ab` [1] benchmark tool from the client to test the performance of the web server on VM1.

VM1 also runs `memcached` [35] server version 1.4.14, installed from the Ubuntu repository with the default configuration file. On the client we run the `memslap` [78] benchmark tool, part of the `libmemcached` client library, to measure the performance of the `memcached` server on VM1.

Figure 9 displays normalized results of the performed tests. We group test results into two categories: throughput oriented and latency oriented. Throughput oriented tests are `iperf` running pure TCP stream and `apache2` serving a 1MB file. These tests are limited by the 10GbE link bandwidth. During the tests, the client and server CPUs are almost idle.

From Figure 9 we conclude that VANFC completely blocks VM2's attack and introduces no performance penalty.

8 Necessity of Flow Control

One can argue that flow control is not required for proper functionality of high level protocols such as TCP. It then follows from this argument that SRIOV can be made "secure" simply by disabling flow control.

The TCP protocol does provide its own flow control mechanism. However, many studies have shown that TCP's main disadvantage is high CPU utilization [28, 36, 46, 55, 66]. Relying on TCP alone for flow control leads to increased resource utilization.

In public cloud environments, users pay for computational resources. Higher CPU utilization results in higher charges. In enterprise data centers and high-performance computing setups, resource consumption matters as well. Ultimately, someone pays for it. In clouds, especially, effective resource utilization will become increasingly more important [12].

Certain traffic patterns that use the TCP protocol in high-bandwidth low-latency data center environments may suffer from catastrophic TCP throughput collapse, a phenomenon also known as the *incast* problem [58]. This problem occurs when many senders simultaneously transmit data to a single receiver, overflowing the network buffers of the Ethernet switches and the receiver, thus causing significant packet loss. Studies show that Ethernet flow control functionality, together with congestion control protocol, can mitigate the *incast* problem, thus improving the TCP performance [27, 62].

As part of a recent effort to converge current network infrastructures, many existing protocols were implemented over Ethernet, e.g., Remote DMA over Converged Ethernet (RoCE) [19]. RoCE significantly reduces CPU utilization when compared with TCP.

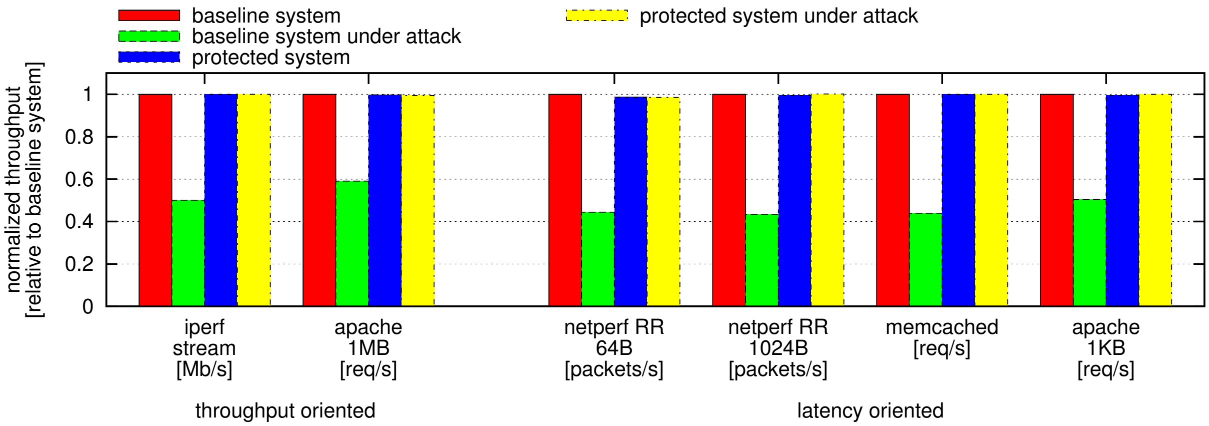
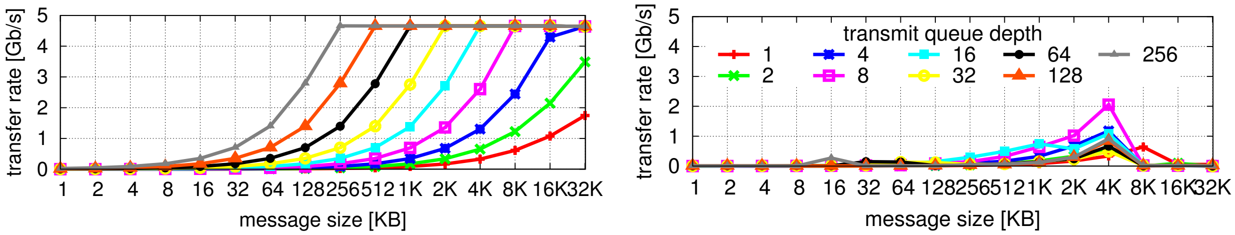


Figure 9: VANFC performance evaluation results



(a)

(b)

Figure 10: Performance of a single RoCE flow in the system with two competing RoCE flows. Graph (a) shows performance with enabled flow control; graph (b) shows performance with disabled flow control.

A few recent studies that evaluate performance of different data transfer protocols over high speed links have been published [48, 49, 67, 72]. Kissel et al. [49] compare TCP and RoCE transfers over 40GbE links using the same application they developed for benchmarking. Using TCP, they managed to reach a speed of 22Gbps while the sender's CPU load was 100% and the receiver's CPU load was 91%. With OS-level optimizations, they managed to reach a speed of 39.5 Gbps and reduce the sender's CPU load to 43%. Using the RoCE protocol, they managed to reach 39.2 Gbps while the CPU load of the receiver and sender was less than 2%! These results clearly show that RoCE significantly reduces CPU utilization and thus the overall cost of carrying out computations. It is especially important when a large amount of data is being moved between computational nodes in HPC or data center environments, where virtualization is becoming prevalent and increasing in popularity [24, 37, 54].

Studies show that RoCE cannot function properly without flow control [48, 49, 67, 72]. Figure 10, taken from Kissel et al. [49], with the authors' explicit permission, shows the performance effect of flow control on two competing data transfers using the RoCE protocol. Figure 10a shows the performance of a single RoCE data transfer while another RoCE data transfer is competing with it for bandwidth and flow control is enabled. Both transfers effectively share link bandwidth. Figure 10b shows the performance of the same RoCE data transfer when flow control is disabled. As can be seen in the figure, without flow control the RoCE data transfer suffers, achieving a fraction of the performance shown in Figure 10a. We have also independently reproduced and verified these results.

Kissel et al. also show [49] that the same problem is relevant not only to RoCE but can be generalized to TCP as well. Thus we conclude that disabling flow control would cause less effective resource utilization and lead to higher cost for cloud customers and for any organization deploying SRIOV. Conversely, securing SRIOV against flow control attacks would make it possible for SRIOV and flow control to coexist, providing the performance benefits of both without relinquishing security.

9 Discussion

Notes on Implementation VANFC can be implemented as part of an SRIOV device already equipped with an embedded Ethernet switch or it can be implemented in the edge Ethernet switch, by programming the edge switch to filter flow control frames from VFs' MAC addresses. Adding VANFC functionality to the NIC requires less manufacturing effort; it is also more convenient and cheaper to replace a single NIC on a host than to replace an edge switch. Nevertheless, in large-scale virtualization deployments, such as those of cloud providers or corporate virtual server farms, a single 10GbE Ethernet switch with high port density (for example, the 48 port HP 5900AF 10Gb Ethernet switch in our testbed) serves many host servers with SRIOV capable devices. In such scenarios, upgrading 48 SRIOV devices connected to the 48 port switch requires considerably more resources than single switch upgrade.

Having said that, we argue that proper implementation of the solution to the described problem is in the SRIOV NIC and not in the edge Ethernet switch. The problem we discuss is strictly related to the virtualization platform and caused by a design flaw in the SRIOV NIC's internal switching implementation. Mitigating the problem in the edge switch, an external device whose purpose is not handle virtualization problems of the host, would force the edge switch to learn about each VF's MAC address and to distinguish PFs from VFs, coupling the edge switch too closely with the NICs.

VEB and VEPA Another important security aspect of SRIOV is VM-to-VM traffic. In SRIOV devices with an embedded VEB switch, VM-to-VM traffic does not leave the host network device and is not visible to the external edge switch, which enforces the security policy on the edge of the network. To make all VM traffic visible to the external switch, the VEB switch should act as a VEPA and send all VM traffic to the adjacent switch.

A properly configured Ethernet switch and the use of a VEPA device can enforce a security policy (ACL, port security) on malicious VM traffic and prevent most L2 attacks. However, while VEPA solves many manageability and security issues that pertain to switching in virtualized environments [29], it does not address the flow control attack we presented earlier. This is because VEPA still shares the same single link between multiple untrusted guests and the host and does not manage flow control per VF. Besides not solving the flow control attack, it uses, again, the edge Ethernet switch, which is external to the source of the problem–SRIOV NIC. Thus, a VEPA extension should not be considered for the solution and the problem should be solved in the SRIOV NIC.

10 Related Work

Several recent works discussed the security of self-virtualizing devices. Pék et al. [61] described a wide range of attacks on host and tenant VMs using directly assigned devices. They performed successful attacks on PCI/PCIe configuration space, on memory mapped I/O, and by injecting interrupts. They also described an NMI injection attack. Most of the attacks they discussed can be blocked by a fix in the hypervisor or by proper hardware configuration.

Richter et al. [68] showed how a malicious VM with a directly attached VF can perform DoS attacks on other VMs that share the same PCIe link by overloading its own Memory Mapped I/O (MMIO) resources and flooding the PCIe link with write request packets. As the authors mention, this attack can be mitigated by using the QoS mechanisms defined by the PCIe standard [59].

All of the attacks discussed in the aforementioned papers are based on weak security implementations of software (e.g., a hypervisor) or hardware (a chipset system error reporting mechanism) that are internal to the host. Our attack exploits different design aspects of SRIOV devices: it targets the interoperability of SRIOV devices with software and hardware external to the host.

There are ongoing efforts of the Data Center Bridging Task Group, which is a part of the IEEE 802.1 Working Group, to standardize configuration, management and communication of virtual stations connected to the adjacent bridge. The working group proposed the 802.1Qbg Edge Virtual Bridging [10] and 802.1BR Bridge Port Extension [11] standards. Both standards concentrate on configuration and management of the bridge services for virtual stations, leaving the flow control of virtual stations out of their scope. To the best of our knowledge, our work is the first to present the problem of self-virtualizing devices in converged enhanced Ethernet environments with flow control, and the first to suggest a solution for it.

11 Conclusions and Future Work

Self-virtualizing devices with SRIOV lie at the foundation of modern enterprise data centers, cloud computing, and high-performance computing setups. We have shown that SRIOV, as currently deployed on current Ethernet networks, is incompatible with required functionality such as flow control. This is because flow control relies on the assumption that each endpoint is trusted, whereas with SRIOV, each network endpoint is comprised of multiple, possibly untrusted, virtual machines. We show how to overcome this flaw by teaching the NIC about virtual functions. We present the prototype of such a system, VANFC, and its evaluation. Our prototype is 100% effective in securing SRIOV against this flaw while imposing no overhead on throughput or latency-oriented workloads.

Future work includes continuing to investigate the security of SRIOV devices; extending our work from Ethernet to other networking technologies such as InfiniBand and Fiber Channel; looking at the security of direct-assigned self-virtualizing devices other than NICs, such as high-end NVMe SSDs and GPGPUs; developing VF co-residency detection techniques; and using the hypervisor to solve the problem of VM ring buffer exhaustion. Handling this with software without losing performance will be challenging. On VANFC specifically, we plan to continue our evaluation and to explore what an eventual hardware-based implementation would look like.

Acknowledgments

We would like to thank the anonymous reviewers, our shepherd, Srdjan Capkun, Shachar Raindel from Mellanox, David H. Lorenz and Ilya Lesokhin from Technion, and Sharon Kessler for insightful comments. This research was supported, in part, by the Ministry of Science and Technology, Israel, grant #3-9609. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

[1] Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.2/programs/ab.html. [Accessed Jul, 2014].

[2] High Performance Computing (HPC) on Amazon Elastic Compute Cloud (EC2) . Online : https://aws.amazon.com/hpc/. [Accessed Jun, 2014].

[3] Iperf - The TCP/UDP Bandwidth Measurement Tool. http://iperf.sourceforge.net. [Accessed Jul, 2014].

[4] Linux Ethernet Bridge. http://www.linuxfoundation.org/collaborate/workgroups/networking/bridge. [Accessed Jul, 2014].

[5] Tcpreplay: Pcap editing and replay tools for Unix systems. http://tcpreplay.synfin.net/. [Accessed Jul, 2014].

[6] IEEE Standards for Local and Metropolitan Area Networks: Supplements to Carrier Sense Multiple Access With Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications - Specification for 802.3 Full Duplex Operation and Physical Layer Specification for 100 Mb/s Operation on Two Pairs of Category 3 Or Better Balanced Twisted Pair Cable (100BASE-T2). *IEEE Std 802.3x-1997 and IEEE Std 802.3y-1997 (Supplement to ISO/IEC 8802-3: 1996; ANSI/IEEE Std 802.3, 1996 Edition)* (1997), 1–324.

[7] IEEE Standard for Local and metropolitan area networks–Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks. *IEEE Std 802.1Q-2011 (Revision of IEEE Std 802.1Q-2005)* (Aug 2011), 1–1365.

[8] IEEE Standard for Local and metropolitan area networks–Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks–Amendment 17: Priority-based Flow Control. *IEEE Std 802.1Qbb-2011 (Amendment to IEEE Std 802.1Q-2011 as amended by IEEE Std 802.1Qbe-2011 and IEEE Std 802.1Qbc-2011)* (Sept 2011), 1–40.

[9] IEEE Standard for Ethernet - Section 2. *IEEE Std 802.3-2012 (Revision to IEEE Std 802.3-2008)* (Dec 2012), 752–762.

[10] IEEE Standard for Local and metropolitan area networks–Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks–Amendment 21: Edge Virtual Bridging. *IEEE Std 802.1Qbg-2012 (Amendment to IEEE Std 802.1Q-2011 as amended by IEEE Std 802.1Qbe-2011, IEEE Std 802.1Qbc-2011, IEEE Std 802.1Qbb-2011, IEEE Std 802.1Qaz-2011, IEEE Std 802.1Qbf-2011, and IEEE Std 802.aq-2012)* (July 2012), 1–191.

[11] IEEE Standard for Local and metropolitan area networks–Virtual Bridged Local Area Networks–Bridge Port Extension. *IEEE Std 802.1BR-2012* (July 2012), 1–135.

[12] AGMON BEN-YEHUDA, O., BEN-YEHUDA, M., SCHUSTER, A., AND TSAFRIR, D. The Rise of RaaS: The Resource-as-a-Service Cloud. *Communications of the ACM (CACM)* (2014).

[13] ALIZADEH, M., ATIKOGLU, B., KABBANI, A., LAKSHMIKANTHA, A., PAN, R., PRABHAKAR, B., AND SEAMAN, M. Data Center Transport Mechanisms: Congestion Control Theory and IEEE Standardization. In *46th Annual Allerton Conference on Communication, Control, and Computing* (2008), IEEE.

[14] ALTUNBASAK, H., KRASSER, S., OWEN, H., GRIMMINGER, J., HUTH, H.-P., AND SOKOL, J. Securing Layer 2 in Local Area Networks. In *4th International Conference on Networking*, Lecture Notes in Computer Science. Springer, 2005.

[15] AMIT, N., BEN-YEHUDA, M., TSAFRIR, D., AND SCHUSTER, A. vIOMMU: efficient IOMMU emulation. In *USENIX Annual Technical Conference (ATC)* (2011).

[16] AMIT, N., BEN-YEHUDA, M., AND YASSOUR, B.-A. IOMMU: Strategies for Mitigating the IOTLB Bottleneck. In *Workshop on Interaction between Operating Systems & Computer Architecture (WIOSCA)* (2010).

[17] ARTEMJEV, O. K., AND MYASNYANKIN, V. V. Fun with the Spanning Tree Protocol. *Phrack 11* (2003), 61.

[18] ASSOCIATION, I. T. InfiniBand Architecture Specification Release 1.2.1, Volume 1. *InfiniBand Trade Association* (2007).

[19] ASSOCIATION, I. T. InfiniBand Architecture Specification Release 1.2.1, Volume 1, Annex A16: RoCE. *InfiniBand Trade Association* (2010).

[20] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the Art of Virtualization. *ACM SIGOPS Operating Systems Review* (2003).

[21] Ben-Yehuda, M., Borovik, E., Factor, M., Rom, E., Traeger, A., and Yassour, B.-A. Adding Advanced Storage Controller Functionality via Low-Overhead Virtualization. In *USENIX Conference on File & Storage Technologies (FAST)* (2012).

[22] Ben-Yehuda, M., Mason, J., Krieger, O., Xenidis, J., van Doorn, L., Mallick, A., Nakajima, J., and Wahlig, E. Utilizing IOMMUs for Virtualization in Linux and Xen. In *Ottawa Linux Symposium (OLS)* (2006).

[23] Ben-Yehuda, M., Xenidis, J., Ostrowski, M., Rister, K., Bruemmer, A., and van Doorn, L. The Price of Safety: Evaluating IOMMU Performance. In *Ottawa Linux Symposium (OLS)* (2007).

[24] Birkenheuer, G., Brinkmann, A., Kaiser, J., Keller, A., Keller, M., Kleineweber, C., Konersmann, C., Niehrster, O., Schfer, T., Simon, J., and Wilhelm, M. Virtualized HPC: a contradiction in terms? *Software: Practice and Experience* (2012).

[25] Bradner, S., and McQuaid, J. Benchmarking methodology for network interconnect devices. RFC 2544, Internet Engineering Task Force, Mar. 1999.

[26] Broadcom Corporation. *Broadcom BCM57810S NetXtreme II Converged Controller*, 2010. [Accessed February 2015].

[27] Chen, Y., Griffith, R., Liu, J., Katz, R. H., and Joseph, A. D. Understanding TCP Incast Throughput Collapse in Datacenter Networks. In *1st ACM workshop on Research on Enterprise Networking* (2009), ACM.

[28] Clark, D. D., Jacobson, V., Romkey, J., and Salwen, H. An analysis of TCP processing overhead. *Communications Magazine, IEEE*, 6 (1989).

[29] Congdon, P. Enabling Truly Converged Infrastructure. http://sysrun.haifa.il.ibm.com/hrl/wiov2010/talks/100313-WIOV-Congdon-dist.pdf, 2010.

[30] Congdon, P., Fischer, A., and Mohapatra, P. A Case for VEPA: Virtual Ethernet Port Aggregator. In *2nd Workshop on Data CenterConverged and Virtual Ethernet Switching* (2010).

[31] Congdon, P., and Hudson, C. Modularization of Edge Virtual Bridging–proposal to move forward. http://www.ieee802.org/1/files/public/docs2009/new-evb-congdon-vepa-modular-0709-v01.pdf, 2009.

[32] de Schuymer, B., and Fedchik, N. Ebtables/Iptables Interaction On A Linux-Based Bridge. http://ebtables.sourceforge.net, 2003. [Accessed Jul, 2014].

[33] Dong, Y., Yang, X., Li, X., Li, J., Tian, K., and Guan, H. High performance network virtualization with SR-IOV. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2010).

[34] Fielding, R. T., and Kaiser, G. The Apache HTTP Server Project. *IEEE Internet Computing*, 4 (1997).

[35] Fitzpatrick, B. Distributed Caching with Memcached. *Linux Journal*, 124 (2004).

[36] Foong, A. P., Huff, T. R., Hum, H. H., Patwardhan, J. P., and Regnier, G. J. TCP Performance Re-visited. In *International Symposium on Performance Analysis of Systems and Software* (2003), IEEE.

[37] Gavrilovska, A., Kumar, S., Raj, H., Schwan, K., Gupta, V., Nathuji, R., Niranjan, R., Ranadive, A., and Saraiya, P. High-Performance Hypervisor Architectures: Virtualization in HPC Systems. In *Workshop on System-level Virtualization for HPC (HPCVirt)* (2007).

[38] Gordon, A., Amit, N., Har'El, N., Ben-Yehuda, M., Landau, A., Schuster, A., and Tsafrir, D. ELI: bare-metal performance for I/O virtualization. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)* (2012), ACM.

[39] Har'El, N., Gordon, A., Landau, A., Ben-Yehuda, M., Traeger, A., and Ladelsky, R. Efficient and Scalable Paravirtual I/O System. In *USENIX Annual Technical Conference (ATC)* (2013).

[40] Hawley, A., and Eilat, Y. Oracle Exalogic Elastic Cloud: Advanced I/O Virtualization Architecture for Consolidating High-Performance Workloads. *An Oracle White Paper* (2012).

[41] Huang, S., and Baldine, I. Performance Evaluation of 10GE NICs with SR-IOV Support: I/O Virtualization and Network Stack Optimizations. In *16th International Conference on Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance* (2012), Springer-Verlag.

[42] Intel Corporation. *Intel 82599 10 GbE Controller Datasheet*, 2014. Revision 2.9. [Accessed August 2014].

[43] Intel Corporation. *Intel I350 10 GbE Controller Datasheet*, 2014. Revision 2.2. [Accessed February 2015].

[44] Jacobson, V., Leres, C., and McCanne, S. Tcpdump: a powerful command-line packet analyzer. http://www.tcpdump.org. [Accessed Jul, 2014].

[45] Jones, R. The Netperf Benchmark. http://www.netperf.org. [Accessed Jul, 2014].

[46] Kay, J., and Pasquale, J. The importance of non-data touching processing overheads in TCP/IP. *ACM SIGCOMM Computer Communication Review* (1993).

[47] Kiravuo, T., Sarela, M., and Manner, J. A Survey of Ethernet LAN Security. *Communications Surveys Tutorials, IEEE* (2013).

[48] Kissel, E., and Swany, M. Evaluating High Performance Data Transfer with RDMA-based Protocols in Wide-Area Networks. In *14th International Conference on High Performance Computing and Communication & 9th International Conference on Embedded Software and Systems (HPCC-ICESS)* (2012), IEEE.

[49] Kissel, E., Swany, M., Tierney, B., and Pouyoul, E. Efficient Wide Area Data Transfer Protocols for 100 Gbps Networks and Beyond. In *3rd International Workshop on Network-Aware Data Management* (2013), ACM.

[50] Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A. KVM: the Linux Virtual Machine Monitor. In *Ottawa Linux Symposium (OLS)* (2007). http://www.kernel.org/doc/ols/2007/ols2007v1-pages-225-230.pdf. [Accessed Apr, 2011].

[51] Ko, M., and Recio, R. Virtual Ethernet Bridging. http://www.ieee802.org/1/files/public/docs2008/new-dcb-ko-VEB-0708.pdf, 2008.

[52] LEVASSEUR, J., UHLIG, V., STOESS, J., AND GÖTZ, S. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Symposium on Operating Systems Design & Implementation (OSDI)* (2004).

[53] LIU, J. Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)* (2010).

[54] LOCKWOOD, G. SR-IOV: The Key to Fully Virtualized HPC Clusters. Online : http://insidehpc.com/2013/12/30/sr-iov-key-enabling-technology-fully-virtualized-hpc-clusters/. Presented on SC13: International Conference for High Performance Computing, Networking, Storage and Analysis. [Accessed Jun, 2014].

[55] MARKATOS, E. P. Speeding up TCP/IP: faster processors are not enough. In *21st International Conference on Performance, Computing, and Communications* (2002), IEEE.

[56] MARRO, G. M. Attacks at the Data Link Layer. Master's thesis, University of California, Davis, 2003.

[57] MELLANOX TECHNOLOGIES. *Mellanox OFED for Linux User Manual*, 2014. Revision 2.2-1.0.1. [Accessed July 2014].

[58] NAGLE, D., SERENYI, D., AND MATTHEWS, A. The Panasas Activescale Storage Cluster: Delivering Scalable High Bandwidth Storage. In *ACM/IEEE conference on Supercomputing* (2004), IEEE.

[59] PCI SIG. PCI Express Base Specification, Revision 3.0, 2010.

[60] PCI SIG. Single Root I/O Virtualization and Sharing 1.1 Specification, 2010.

[61] PÉK, G., LANZI, A., SRIVASTAVA, A., BALZAROTTI, D., FRANCILLON, A., AND NEUMANN, C. On the Feasibility of Software Attacks on Commodity Virtual Machine Monitors via Direct Device Assignment. In *9th ACM Symposium on Information, Computer and Communications Security* (2014), ACM.

[62] PHANISHAYEE, A., KREVAT, E., VASUDEVAN, V., ANDERSEN, D. G., GANGER, G. R., GIBSON, G. A., AND SESHAN, S. Measurement and Analysis of TCP Throughput Collapse in Cluster-based Storage Systems. In *USENIX Conference on File & Storage Technologies (FAST)* (2008).

[63] POSTEL, J. B. Transmission control protocol. RFC 793, Internet Engineering Task Force, Sept. 1981.

[64] RAJ, H., AND SCHWAN, K. High Performance and Scalable I/O Virtualization via Self-Virtualized Devices. In *International Symposium on High Performance Distributed Computer (HPDC)* (2007).

[65] RAM, K. K., SANTOS, J. R., TURNER, Y., COX, A. L., AND RIXNER, S. Achieving 10Gbps using Safe and Transparent Network Interface Virtualization. In *ACM/USENIX International Conference on Virtual Execution Environments (VEE)* (2009).

[66] REGNIER, G., MAKINENI, S., ILLIKKAL, R., IYER, R., MINTURN, D., HUGGAHALLI, R., NEWELL, D., CLINE, L., AND FOONG, A. TCP Onloading for Data Center Servers. *Computer Magazine, IEEE* (2004).

[67] REN, Y., LI, T., YU, D., JIN, S., ROBERTAZZI, T., TIERNEY, B., AND POUYOUL, E. Protocols for Wide-Area Data-Intensive Applications: Design and Performance Issues. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)* (2012).

[68] RICHTER, A., HERBER, C., RAUCHFUSS, H., WILD, T., AND HERKERSDORF, A. Performance Isolation Exposure in Virtualized Platforms with PCI Passthrough I/O Sharing. In *Architecture of Computing Systems (ARCS)*. Springer International Publishing, 2014.

[69] RUSSELL, R. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review (OSR)* (2008).

[70] STEPHENS, B., COX, A. L., SINGLA, A., CARTER, J., DIXON, C., AND FELTER, W. Practical DCB for Improved Data Center Networks. In *International Conference on Computer Communications (INFOCOM)* (2014), IEEE.

[71] SUGERMAN, J., VENKITACHALAM, G., AND LIM, B.-H. Virtualizing I/O Devices on Vmware Workstation's Hosted Virtual Machine Monitor. In *USENIX Annual Technical Conference (ATC)* (2001).

[72] TIERNEY, B., KISSEL, E., SWANY, M., AND POUYOUL, E. Efficient Data Transfer Protocols for Big Data. In *8th International Conference on E-Science* (2012), IEEE Computer Society.

[73] TREJO, L. A., MONROY, R., AND MONSALVO, R. L. Spanning Tree Protocol and Ethernet PAUSE Frames DDoS Attacks: Their Efficient Mitigation. Tech. rep., Instituto Tecnolgico de Estudios Superiores de Monterrey, ITESM-CEM, 2006.

[74] WILLMANN, P., SHAFER, J., CARR, D., MENON, A., RIXNER, S., COX, A. L., AND ZWAENEPOEL, W. Concurrent Direct Network Access for Virtual Machine Monitors. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2007).

[75] WONG, A., AND YEUNG, A. Network Infrastructure Security. In *Network Infrastructure Security*. Springer, 2009.

[76] YASSOUR, B.-A., BEN-YEHUDA, M., AND WASSERMAN, O. Direct Device Assignment for Untrusted Fully-Virtualized Virtual Machines. Tech. Rep. H-0263, IBM Research, 2008.

[77] YASSOUR, B.-A., BEN-YEHUDA, M., AND WASSERMAN, O. On the DMA Mapping Problem in Direct Device Assignment. In *Haifa Experimental Systems Conference (SYSTOR)* (2010), ACM.

[78] ZHUANG, M., AND AKER, B. Memslap: Load Testing and Benchmarking Tool for memcached. http://docs.libmemcached.org/bin/bin/memslap.html. [Accessed Jul, 2014].

EASEAndroid: Automatic Policy Analysis and Refinement for Security Enhanced Android via Large-Scale Semi-Supervised Learning

Ruowen Wang[1,2] William Enck[2] Douglas Reeves[2] Xinwen Zhang[1]
Peng Ning[1,2] Dingbang Xu[1] Wu Zhou[1] Ahmed M. Azab[1]

[1]Samsung KNOX R&D, Samsung Research America
{peng.ning, xinwen.z1, dingbang.xu, wu1.zhou, a.azab}@samsung.com
[2]Department of Computer Science, NC State University
{rwang9, whenck, reeves, pning}@ncsu.edu

Abstract

Mandatory protection systems such as SELinux and SEAndroid harden operating system integrity. Unfortunately, policy development is error prone and requires lengthy refinement using audit logs from deployed systems. While prior work has studied SELinux policy in detail, SEAndroid is relatively new and has received little attention. SEAndroid policy engineering differs significantly from SELinux: Android fundamentally differs from traditional Linux; the same policy is used on millions of devices for which new audit logs are continually available; and audit logs contain a mix of benign and malicious accesses. In this paper, we propose *EASEAndroid*, the first SEAndroid analytic platform for automatic policy analysis and refinement. Our key insight is that the policy refinement process can be modeled and automated using semi-supervised learning. Given an existing policy and a small set of known access patterns, EASEAndroid continually expands the knowledge base as new audit logs become available, producing suggestions for policy refinement. We evaluate EASEAndroid on 1.3 million audit logs from real-world devices. EASEAndroid successfully learns 2,518 new access patterns and generates 331 new policy rules. During this process, EASEAndroid discovers eight categories of attack access patterns in real devices, two of which are new attacks directly against the SEAndroid MAC mechanism.

1 Introduction

Operating system integrity relies on the correctness of 1) trusted computing base (TCB) code and 2) access control policy protecting the TCB code and OS resources. It is generally impractical to verify the correctness of OS code in commodity systems. Therefore, mandatory access control (MAC) policy is often used as a fallback when the security of the software inevitably fails [29].

SELinux [6] is the most notable MAC policy framework widely used in practice. Security Enhanced Android [38] (known simply as SEAndroid) is a recent port of SELinux to the Android platform. However, while Android is based on a Linux kernel, the runtime environment is vastly different than existing Linux distributions for commodity PCs. This difference resulted in a complete redesign of the MAC policy rules, with several new object classes (e.g., for Android's binder IPC).

As with SELinux, SEAndroid policy development is a challenging task, requiring many iterations of refinement to be ready for commercial deployment. For example, Google introduced a very permissive SEAndroid policy into Android version 4.3 and did not enable enforcement. Version 4.4 enabled enforcement, but the policy was still very permissive, containing only a few system daemons. Finally, Android version 5.0 provides a much more robust (but not perfect) version of the policy. Additionally, major smartphone vendors need to customize Google's base SEAndroid policy for their devices to add additional protections against known attacks.

SEAndroid policy refinement is currently a very manual process that typically involves analyzing audit logs to identify proposed changes. There are two general approaches to SEAndroid policy refinement. The first approach is to develop a least privilege [35] policy (also known as a "strict" policy in SELinux terminology) and monitor audit logs for access patterns that should be allowed. The second approach is to begin with a more permissive policy, refine the policy to prevent (or contain) privilege escalation attacks and use audit logs to verify the refinement. Each approach has disadvantages. If the policy is too strict, it will hurt the usability of deployed real-world devices. If the policy is too permissive, it will allow attacks. As a result, smartphone vendors use a combination of these two approaches.

The goal of our research is to significantly reduce the manual effort required to refine SEAndroid policy using audit logs. Audit log analysis is challenging for several reasons. First, audit logs are collected from millions of real-world devices, and purely manual analysis is impractical. Second, the audit logs contain both benign access patterns and malicious access patterns. Existing SELinux tools such as `audit2allow` that blindly create rules to allow all access patterns in audit logs are error prone. Third, the functionality of both benign applications and malicious exploits is continually changing/upgrading, requiring frequent reassessment of the deny/allow boundary to refine the policy.

In this paper, we present Elastic Analytics for SEAndroid (EASEAndroid) as the first large-scale audit log and policy analytic platform for automatic policy analysis and refinement of SEAndroid-style MAC policy. Our key insight is that the policy refinement process can be modeled and automated using semi-supervised learning [18], a popular knowledge-base construction technique [16, 21]. We apply EASEAndroid to a database of 1.3 million audit logs from real-world Samsung devices running Android 4.3 over the entire year of 2014.[1] EASEAndroid correctly discovers 336 new benign access patterns and 2,182 new malicious access patterns, and automatically translates them into 331 policy rules. The generated rules are consistent with rules manually added by policy analysts. Among the malicious access patterns, EASEAndroid further discovers two new types of attacks in the wild directly targeting SEAndroid MAC mechanism itself.

This paper makes the following contributions:

- *We propose EASEAndroid, a semi-supervised learning approach for refining MAC policy at large scale.* Our approach scales to millions of audit logs that contain a mix of benign and malicious access patterns. While we focus on SEAndroid, the approach is more broadly applicable to type enforcement (TE) MAC policy.

- *We implement a prototype of EASEAndroid to help policy analysts analyze SEAndroid audit logs.* The implementation generates policy refinements, and discovers new Android attacks, providing new knowledge of both benign and malicious access patterns learned from audit logs for policy analysts.

- *We evaluate EASEAndroid on 1.3 million audit logs from real-world Samsung devices.* Using this dataset, EASEAndroid successfully learns 2,518 benign and malicious access patterns and generates 331 policy rules as a refinement. EASEAndroid also discovers two new types of attacks directly targeting SEAndroid. With the help of EASEAndroid, this is the first large-scale study on real-world malicious access patterns in Android devices.

The remainder of this paper proceeds as follows. Section 2 provides background on SEAndroid and semi-supervised learning. Section 3 defines the problem addressed in this paper. Section 4 describes the EASEAndroid design. Section 5 evaluates EASEAndroid against a large database of real-world audit logs. Section 6 discusses limitations. Section 7 overviews related work. Section 8 concludes.

2 Background

2.1 SELinux and SEAndroid

SEAndroid is a port of SELinux's type enforcement (TE) MAC policy to the Android platform [4]. As such, SEAndroid enforces mandatory policy on system-level operations between subjects and objects (e.g., system calls) [6]. In general, processes are regarded as subjects, whereas files, sockets, etc. are objects in different classes. A security context label is assigned to subjects (or objects) that share the same semantics. Traditionally, the subject label is called a *domain*, and the object label is called a *type* (nomenclature from DTE [12]). A policy rule defines which domain of subjects can operate which class and type of objects with a set of permissions, such as open, read, write [28]. For example,

```
allow app app_data_file:file {open read}
```

allows processes with the `app` domain to `open` and `read` `file` class objects assigned the `app_data_file` type. In addition to `allow` rules, SELinux provides `neverallow` rules to define policy invariants for malicious accesses that should never be allowed. These rules are enforced at policy compile-time and are necessary due to the complexity of the SELinux policy language.

SEAndroid extends SELinux's policy semantics to support Android-specific functionality, including mediation of Binder IPC and assigning security contexts based on application digital signatures. The goal of SEAndroid is to reduce the attack surface and limit the damage if any flaw or vulnerability is exploited causing privilege escalation [38]. This goal is accomplished by confining the capabilities of different privileged Android applications and system daemons.

The Android platform is vastly different than traditional Linux distributions, therefore the SEAndroid policy rules were created from scratch. While the regularity of Android's UNIX-level interactions results in a policy that is less complex than the example SELinux policy for PCs, the SEAndroid policy is still nontrivial and error prone. It requires careful understanding of subtle interactions between different privileged processes. In practice, policy development requires continual manual refinement based on audit logs.

[1] See Appendix A for more details about audit log collection.

```
type=1400 msg=audit(1399587808.122:14):
avc: denied { entrypoint } pid=285 comm="init"
scontext=u:r:init:s0
tcontext=u:object_r:system_file:s0 tclass=file

type=1300 msg=audit(1399587808.122:14):
syscall=11(execve) success=no exit=-13
items=1 ppid=1 pid=285 uid=0 gid=0
comm="init" exe="/init" subj=u:r:init:s0

type=1302 msg=audit(1399587808.122:14):
item=0 name="/system/etc/install-recovery.sh"
inode=3799 dev=b3:10 mode=0100755
ouid=0 ogid=0 obj=u:object_r:system_file:s0
```

Listing 1: A denied access event example recorded at the epoch time 1399587808.122 in an audit log. It consists of three entries: labels & permission (1400), syscall & process info (1300), object info (1302).

An audit log captures security labels and system calls of the operations that are not explicitly allowed by a rule. As shown in Listing 1, a denied operation generally has three entries with epoch timestamps. Log entries with `type=1400` record the denied permission (e.g., `entrypoint`), the security labels of the subject (source), called `scontext`, and the object (target), called `tcontext`, as well as the object's class, called `tclass` (e.g., `file`). Log entries with `type=1300` record the system call and the subject's process information, including the executable file path. Log entries with `type=1302` record the object information (e.g., the file name).

Traditionally, policy analysts develop and refine a policy by manually analyzing audit logs. Existing SELinux tools such as Tresys's setools [8] are used to analyze SEAndroid policies based on interactive user interface. Analysts also develop simple shell and Python scripts to parse audit logs. Unfortunately, such tools are not scalable to a large number of audit logs, and cannot distinguish benign or malicious access patterns in real-world audit logs. In addition, analysts often use a tool called `audit2allow` [9] that can create new `allow` rules by directly using the security labels captured in `type=1400` entries in audit logs. However, blindly using this tool may increase attack surface, because in some cases, existing labels are too coarse-grained or semantically inappropriate. Therefore, policy refinement usually consists of the creation and modification of both security labels and policy rules. Once the policy is refined by analysts, it is pushed to users' devices through a secure over-the-air (OTA) channel, similar to antivirus signature updates.

2.2 Semi-Supervised Learning

Semi-supervised learning is a type of machine learning that trains on both labeled[2] data (used by supervised learning) and unlabeled data (used by unsupervised learning) [18]. It is typically used when labeled data is insufficient and expensive to collect, and a large set of unlabeled data is available. By correlating the features in unlabeled data with labeled data, a semi-supervised learner infers the labels of the unlabeled instances with strong correlation. This labeling increases the size of labeled data set, which can be used to further re-train and improve the learning accuracy [44]. This iterative training process is commonly referred to as bootstrapping. Semi-supervised learning is popular for information extraction and knowledge base construction. Examples include NELL [16, 17], Google Knowledge Vault [21].

We hypothesize that the process of developing and refining SEAndroid policy is analogous to semi-supervised learning. Human analysts encode their knowledge about various access patterns into a policy. When analyzing audit logs, analysts find semantic correlations between known and unknown access patterns to infer whether the unknown ones are benign or malicious, such as a known malicious subject performing an unseen behavior (likely malicious), or a system daemon performing a new but similar functional operation (likely benign). These new patterns expand analysts' knowledge and help them refine the policy. However, when more and more logs are collected containing access patterns about new Android systems and new attacks, manual learning is time-consuming and likely to miss important knowledge. Our insight is that semi-supervised learning can automate this process to achieve scalability in policy refinement.

3 Problem

Refining SEAndroid policy is more challenging than refining SELinux policy. Existing SELinux tools such as `audit2allow` are severely limited in their ability to help policy analysts. This task has the following challenges.

C-1: *Consumer devices produce millions of audit logs.* Policy analysts cannot practically analyze audit log entries manually. A solution must automate or semi-automate the audit log analysis.

C-2: *Real-world audit logs contain a mixture of benign and malicious accesses.* Classifying log entries as benign or malicious is a central design challenge. It is often difficult to classify an access in isolation. Instead, the analysis must look at the broad context of the access, as well as the contexts of related known accesses.

C-3: *Target functionality is not static.* The set of benign and malicious applications continues to evolve as new software and malware is developed, requiring continuous audit log analysis and policy refinement.

[2] Here, labeled and unlabeled are machine learning terms, not related to security labels.

For example, benign software may access new resources, while malware may exploit new vulnerabilities to achieve privilege escalation.

We now define two terms to clarify the discussion in the remainder of this paper.

Definition 1 (Access Event)**.** *An access event is the access control event that causes the three audit log entries described in Section 2. These log entries may result from a policy denial, or an* `auditallow` *policy rule, which allows but logs the access.*

Note that this definition does not include allowed accesses that are not contained in the audit log.

Since audit logs are collected for millions of devices, the logs contain many duplicate access events. For the purposes of audit log analysis, it is useful to abstract the salient details of access events into an access pattern.

Definition 2 (Access Pattern)**.** *An access pattern is a 6-tuple (sbj, sbj_label, perm, tclass, obj, obj_label). Many access events may map to the same access pattern.*

Here *sbj* refers to a concrete subject such as an Android application or system binary. Binaries carried inside an application are generalized as the application. *obj* refers to concrete objects such as file paths and socket names. In some cases, we group over-specific files that share the same filesystem semantics as one *obj* (e.g., `/sdcard`). The values of *sbj* and *obj* are derived from the `comm`, `exe`, `pid`, and `name` values in the `type=1300,1302` log entries. *perm* and *tclass* are the same as the permission and the object's class in the `type=1400` log entries and policy rules. *sbj_label* and *obj_label* are derived from the `scontext` and `tcontext` values in `type=1400` log entries. For example, the access pattern for the access event in Listing 1 is ("`/init`", "`init`", "`entrypoint`", "`file`", "`/system/etc/install-recovery.sh`", "`system_file`").

Problem Statement: Given 1) a large dataset of new access patterns from audit logs, 2) a small set of known access patterns (e.g., known attacks), and 3) an SEAndroid policy, we seek to a) separate new benign access patterns from new malicious access patterns in the dataset, and b) suggest new rules and refined labels for the policy.

Threat Model and Assumptions: We assume that an audit log is collected from an Android device with a policy loaded in either enforcing or permissive mode. We assume the integrity of audit log contents. We therefore assume that the Linux kernel and its audit subsystem are not compromised. However, even if the SEAndroid policy properly confines Android applications and system daemons, they may be compromised by the adversary.

4 EASEAndroid

Elastic Analytics for SEAndroid (EASEAndroid) is a large-scale audit log and policy analytic platform for automated policy refinement. The novelty of EASEAndroid is that it models the policy refining process as a semi-supervised learning of new access patterns. At a high level, EASEAndroid starts with an initial knowledge base containing existing policy rules and a small set of (potentially manually) identified access patterns. It expands the knowledge base by correlating, classifying, and incorporating new access patterns captured by audit logs. Based on the new knowledge, EASEAndroid suggests policy changes (new rules and new domain and type labels in the context of SEAndroid). As more audit logs become available, EASEAndroid continuously expands the knowledge base and refines the policy.

Figure 1 shows the architecture of EASEAndroid. The architecture uses three machine learning algorithms that consider different perspectives of the knowledge base and audit logs. The output of these algorithms is fed into a combiner that combines and appends the new knowledge into the knowledge base. This learning process is iterated multiple times until no more new knowledge can be learned from the current audit log input. Finally, the policy generator suggests refinements.

Each machine learning algorithm analyzes a different perspective of the data. The goal of each algorithm is to find semantic correlations between unknown new access patterns and existing knowledge base, in order to classify each new access pattern as benign or malicious.

1. The *nearest-neighbors-based (NN) classifier* classifies new access patterns based on their relations to known access patterns in the knowledge base. It finds new access patterns that are related to known subjects/objects (e.g., known subjects are updated and perform new access patterns). By treating these known subjects/objects as neighbors of the new access patterns, it classifies the new access patterns based on the majority of their known neighbors.
2. The *pattern-to-rule distance measurer* calculates the distance between new access patterns and existing policy rules. If a new access pattern is closest to an `allow` rule, it is classified as benign. If it is closest to a `neverallow` rule, it is classified as malicious. If the access pattern is not close to either type of rule, it remains unclassified. The pattern-to-rule distance measurer also exposes potentially incomplete rules in existing policy for refinement.
3. The *co-occurrence learner* considers correlations across access patterns using statistical relations between new and known access patterns that frequently occur together in audit logs. Our intuition is

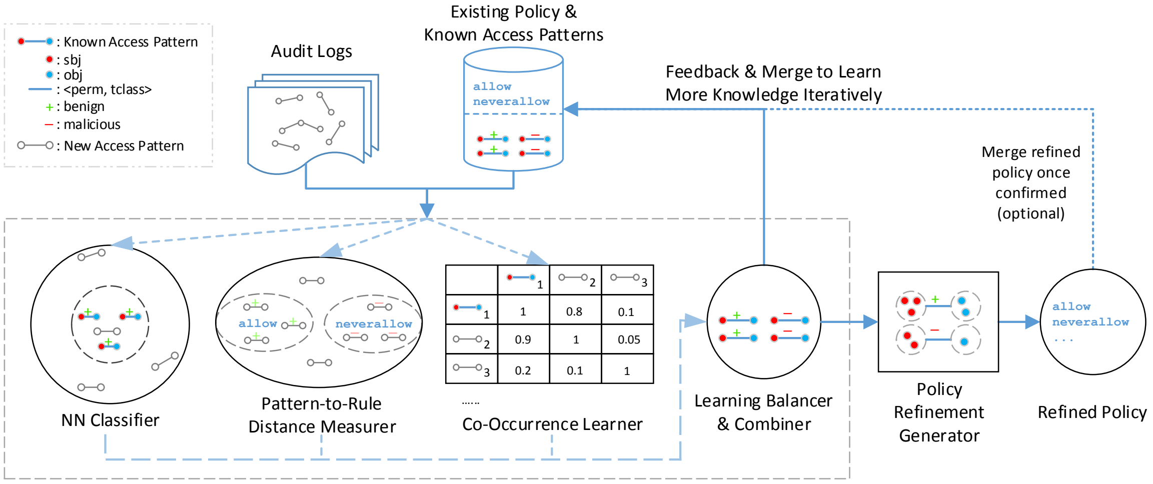


Figure 1: EASEAndroid consists of four learning components and a policy generator to iteratively learn new patterns and refine policy

that a benign functionality or a malicious attack often involves a series of access patterns that are captured together in an audit log. If known and new access patterns occur together, we can use the known ones to infer the classification of the new ones.

Each learner is configured with its own threshold to classify new access patterns independently. However, it is non-trivial to define proper thresholds because too relaxed thresholds could cause false classification, lowering the learning precision, while too strict thresholds could leave potential access pattern candidates unclassified, lowering the learning coverage.

The *learning balancer & combiner* manages the threshold of each learner, balances the precision and coverage, and combines the classification results from the three learners. It has two modes: (1) an *automated mode* that uses strict threshold in each learner to achieve high precision with the cost of less coverage; and (2) a *semi-automated mode* that relaxes each learner's threshold to achieve high coverage and relies on a majority vote from the three learners to increase the precision. In practice, the result of semi-automated mode requires policy analysts' verification to control error rate.

Finally, the *policy generator* takes newly classified access patterns as input from the combiner, to suggest policy refinements in the form of new rules and new security labels. It uses a clustering algorithm to group similar subjects and objects together. The clustering algorithm follows the principle of least privilege by inferring fine-grained labels that can cover and only cover the clustered concrete subjects and objects. In practice, the resulting refined policy can be confirmed by policy analysts and merge into the knowledge base to analyze new audit logs after the refined policy is deployed.

The remainder of this section describes each stage of the EASEAndroid architecture in detail.

4.1 Nearest-Neighbors-based Classifier

Nearest-neighbors-based (NN) learning is a common technique for classifying an unlabeled instance based on its nearest labeled neighbors within a defined distance [41]. Our intuition of using NN for access pattern classification is two-fold. First, known subjects often perform previously unseen access patterns in audit logs. This scenario often occurs when Android applications and system binaries are updated with new capabilities. Second, some known access patterns are also performed by new subjects. This scenario occurs when certain operations become popular and are copied by other new applications. In practice, some exploit kits and repackaged applications [43] have been found to share the same set of known malicious access patterns.

These two scenarios cause known subjects and patterns to be semantically connected with new subjects and patterns. EASEAndroid leverages this connectivity as the distance metric to design the NN classifier. When multiple known subjects (or patterns) connect to the same new pattern (or subject), the NN classifier can infer whether the new pattern (or subject) is benign or malicious, based on the majority of the connected known neighbors. Note that here the observation is with respect to concrete subjects and objects in access patterns. Hence, only a 4-tuple $(sbj, perm, tclass, obj)$ out of the original 6-tuple is required. For completeness, our implementation still includes sbj_label and obj_label in the dataset, but they are not used in this learner.

Algorithm 1 shows the procedure of the NN classifier. AP_k collects known 4-tuple access patterns, either benign or malicious. In practice, our AP_k is a small set contain-

Algorithm 1 NN-based Classification of Access Patterns

```
$AP_k \leftarrow \{(s_k, p_k, t_k, o_k) | s_k \in S_k, (p_k, t_k, o_k) \in P_k\}$
$AP_u \leftarrow \{(s_u, p_u, t_u, o_u) | s_u \in S_u, (p_u, t_u, o_u) \in P_u\}$
$AP_c \leftarrow \emptyset$
procedure NN_CLASSIFIER($AP_k$, $AP_u$, $AP_c$)
    for each $(s, p, t, o) \in AP_u$ do
        if $s \in S_k \cap S_u$ and $(p, t, o) \in P_u - P_k$ then
            $S_{tmp} \leftarrow findAllSbjs((p, t, o), AP_u)$
            if $IsMajorityKnown(S_{tmp}, S_k)$ then
                $AP_c \leftarrow AP_c \cup Classify((s, p, t, o))$
            end if
        else if $s \in S_u - S_k$ and $(p, t, o) \in P_k \cap P_u$ then
            $P_{tmp} \leftarrow findAllPatterns(s, AP_u)$
            if $IsMajorityKnown(P_{tmp}, P_k)$ then
                $AP_c \leftarrow AP_c \cup Classify((s, p, t, o))$
            end if
        end if
    end for
end procedure
return $AP_c$
```

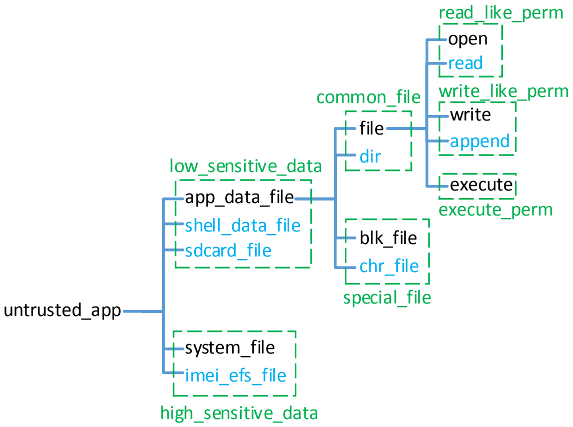


Figure 2: A decision tree example based on rules related to subject domain as `untrusted_app`. The black nodes are from existing rules and the blue nodes are semantic siblings.

ing a few well-confirmed subjects and patterns, used as the initial seed. AP_u collects all unknown new access patterns from audit logs. To clearly describe the above two cases, we further divide the 4-tuple into S for all subjects, and P for the triples *(perm, tclass, obj)* as partial patterns shared by multiple subjects. AP_c is the result set of newly classified access patterns.

For each 4-tuple in AP_u, we check if it is a known subject with a new triple (partial pattern), or a new subject with a known triple. In the first case, besides the subject in this 4-tuple, $findAllSbjs$ collects all subjects S_{tmp} that perform (connect) the same new triple in AP_u, including both known and new subjects. Then $IsMajorityKnown$ checks if the majority of S_{tmp} is a set of known subjects from S_k with the same benign or malicious flag. If so, the new access pattern is classified as benign or malicious accordingly. The second case is done in the same way but using known triples to classify new subjects.

The function $IsMajorityKnown$ uses two empirically defined thresholds (m, σ). m determines the minimum required neighbors and σ is a percentage for how many known neighbors in S_{tmp} or P_{tmp} are required as a majority. Table 1 in the evaluation studies the effects of different threshold values.

From the perspective of machine learning, our NN-based classifier is a type of *radius-based near neighbors learning* [13], a variant of the common k-nearest-neighbors (kNN). The difference is that kNN is based on the top k neighbors while we find all neighbors within a radius as nearest neighbors (connectivity is the radius in our case).

Note that, it is possible that some access patterns are rarely connected with known ones. Besides, an access pattern could be evenly connected to both known benign and malicious ones. Both cases cause $IsMajorityKnown$ to return false. In this case, the NN classifier leaves the access patterns as unclassified and relies on the following learners to complement the learning process.

4.2 Pattern-to-Rule Distance Measurer

EASEAndroid's second data perspective is the closeness of access patterns to policy rules. Since audit logs record denied accesses that cannot match with an `allow` rule, it is useful to know how far/close the denied access pattern is from an existing rule. In particular, because policy rules are developed incrementally, they may only cover a subset of permissions or access patterns and miss similar access patterns belonging to the same operation.

A common case of this is an imprecise list of permissions in an `allow` rule. For example, writing a file not only requires `write` permission, but also `append` and sometimes `create` (in case the file does not exist). Some malicious operations can also be performed using semantically equivalent, but different access patterns.

The pattern-to-rule distance measurer quantifies the difference between access patterns and existing rules. The purpose of this measurer is two-fold. First, pattern-to-rule distance indicates how likely a new access pattern is to be benign or malicious. Second, if an access pattern is very close to a policy rule, the policy refinement generator (Section 4.5) can update the rule rather than creating a new rule from scratch.

The distance measurer uses a metric based on the 1) subject label, i.e., *domain*, 2) object label, i.e., *type*, 3) tclass, e.g., `file`, and 4) permission, e.g., `write`. Note that all four of these elements are in both the SE-

Android policy allow rules, as well as the 6-tuple representing an access pattern in the audit log. Intuitively, an access pattern is very close to a rule if it shares the same labels and tclasses only with slightly different permissions (e.g., write vs append). The distance increases a little, but is still close, if a pattern and a rule operate on different but similar tclasses (e.g., file vs dir).

EASEAndroid systematically measures distance using decision trees based on existing policy rules. The distance is defined by the matching depth for a specific access pattern. Decision trees are built as follows.

Step 1: For every subject label, find all related rules and follow their semantic order to build a tree skeleton starting from the subject as the root, followed by object labels, tclasses and permissions as nodes in each layer.

Step 2: Extend each node with its semantically similar siblings.

Figure 2 shows an example decision tree. The black nodes indicate the tree skeleton, which uses rules such as allow untrusted_app app_data_file:file {open}, where app_data_file is the object node in the second layer and file is the tclass node in the third layer and so on. Then each node is extended with its semantic siblings, such as sdcard_file in the same group of low sensitive data as app data.

Given an access pattern and a decision tree, the distance measurer walks the decision tree and tries to match the access pattern's subject label, object label, tclass, and permission with each layer. The matching depth indicates how close a pattern is to existing rules. For the example in Figure 2, access pattern $ap_i =$ (untrusted_app, sdcard_file, dir, read) matches the fourth layer. The distance is computed as follows:

$$Dist(ap_i) = TotalLayerDepth - MatchedDepth(ap_i)$$

If we define $TotalLayerDepth = 4$, then $Dist(ap_i) = 0$, indicating the access pattern is very close to the rule. We create trees for both allow and neverallow rules to compute the distances from both sides.

In practice, the effectiveness of the distance metric depends on the correctness of semantic siblings. Fortunately, the SEAndroid policy development frequently uses semantic groups. A list of permissions, tclasses, and object types are already grouped together in policy source code using macros and attribute [7]. These groups form a ground truth for semantic siblings.

Additionally, recall that some existing subject and object labels are coarse-grained (e.g., labels assigned to various objects using wildcard in policy source code). If a pattern matches with a rule with a coarse-grained label, the distance measurer marks the distance as low confidence and relies on the learning balancer & combiner for additional verification (Section 4.4).

Finally, note that this technique can be further extended to measure access pattern to access pattern distance. Since the pattern-to-rule distance helps to infer both new patterns as well as identify incomplete rules for refinement, our design considers policy rules and leaves the distance between access patterns for future work.

4.3 Co-Occurrence Learner

When analyzing a large number of audit logs, some access patterns frequently occur together in many logs. This is because some high-level benign functionality or some popular multi-step attacks consist of a series of access patterns within a time period (typically minutes). The statistics of co-occurrence is a valuable means of correlating access patterns that have different subjects or objects, but share the same group semantics. When a group contains both known and new access patterns, the known access patterns can be used to infer the semantics of the new access patterns. In fact, co-occurrence is popular in natural language processing and knowledge extraction. For example, it is used for finding words that are frequently used together in a specialized domain [15].

The co-occurrence of access pattern can be represented using a $n \times n$ matrix for all n unique access patterns from the audit logs, as shown below. Each row stores one access pattern ap_i's co-occurrence percentage with every other access pattern, denoted in each column. The value c_{ij} is the percentage of the number of times that ap_i co-occurs with ap_j out of the total number of ap_i's occurrences throughout the logs.

$$CO_{AP} = \begin{array}{c} \\ ap_i \\ ap_j \\ \ldots \end{array} \begin{array}{c} \begin{array}{ccc} ap_i & ap_j & \ldots \end{array} \\ \begin{bmatrix} 1 & c_{ij} & \ldots \\ c_{ji} & 1 & \ldots \\ \ldots & \ldots & 1 \end{bmatrix} \end{array},$$

$$\text{where } c_{ij} = \frac{CoOccurNum(ap_i, ap_j)}{TotalOccurNum(ap_i)}$$

When counting the number of co-occurrences, it is important to avoid noise and duplicates. In practice, we use a time frame of 10 minutes to determine whether two access patterns are part of a co-occurrence set. Recall from Section 2 that each access pattern has an epoch timestamp. Additionally, when counting the occurrence at the granularity of logs, repeated pairs of co-occurred access patterns in one log are counted only once.

To use this co-occurrence matrix, the learner focuses on the rows with new access patterns. For each ap_i row, the learner sorts columns and selects the set of known access patterns in columns whose percentage is above a threshold. A majority vote of this known access pattern set determines the classification of the new ap_i (benign or malicious). On the other hand, the known access pattern rows may also have some highly co-occurred new access pattern columns. However, one known access pattern is usually not enough to classify a new access pattern.

Note that the matrix is not symmetric. c_{ij} can be different from c_{ji} due to different total occurrence counts. For instance, some popular known malicious access patterns (e.g., `remount /system`) can co-occur with multiple less popular new access patterns, because multi-step attacks often use different steppingstones to achieve the final privilege escalation goal.

4.4 Learning Balancer & Combiner

Each learner is configured with its own threshold to classify new access patterns independently. However, it is non-trivial to define proper thresholds due to two reasons. On the one hand, if a threshold is too relaxed, it could cause false classification, lowering the learning precision and might further propagate the error to the next iteration of semi-supervised learning. On the other hand, if a threshold is too strict, it could miss potential access pattern candidates and leaves them as unclassified, lowering the learning coverage.

We design the learning balancer & combiner to manage the threshold setting of each learner, balance the precision and coverage, and combine the classification results from the three learners (also called ensemble or multi-view learning [17]). The final combined classification result is added to the knowledge base and sent to the policy refinement generator. Specifically, we propose two quantifiable methods to achieve the balancing:

Automated Mode: Since each learner specializes in one dimension, each learner with a strict threshold can directly contribute its classified access patterns with high precision. For example, we can set a minimum of 10 required known neighbors with a 90% bar for $IsMajorityKnown$ in NN classifier; $Dist(ap_i) = 0$ with fine-grained rules in pattern-to-rule distance measurer; and $c_{ij} > 0.9$ with known access pattern set ≥ 10 in co-occurrence learner. The high precision of strict thresholds enables EASEAndroid to be used in an automated mode over multiple iterations of semi-supervised learning. However, with such strict thresholds, some access pattern candidates can be left as unclassified.

Semi-Automated Mode: This mode relaxes the thresholds to get more access pattern candidates. It uses a majority vote to choose the candidates shared by at least two learners with the same classification result, and the third learner must not have conflicting result.

Note that relaxed thresholds can increase the possibility of error propagation. However, if the analysis can tolerate a semi-automated configuration, relaxed thresholds can be used. Here a human analyst can investigate low-confident candidates and input external knowledge into EASEAndroid for better learning in future.

4.5 Policy Refinement Generator

Finally, the policy refinement generator translates newly classified access patterns[3] into the final policy form. A key part of the generator is to assign the concrete subjects (*sbjs*) and objects (*objs*) in the access pattern with appropriate security labels before generating policy rules.

According to the Android Open Source Project, Google provides a baseline definition of security labels for common subjects (e.g., system apps and binaries) and basic objects (e.g., basic files/dirs in Android file system structure). However, Google recommends that manufacturers replace the generic default labels with fine-grained labels to decrease the attack surface [4].

Recall that both the access pattern 6-tuple and the incomplete rules identified by the distance measurer include subject labels and object labels from the existing policy. While some of the labels are coarse-grained, they serve as a baseline to derive fine-grained labels. Specifically, the policy refinement generator takes all access patterns as input and clusters them into groups where each group shares the same 4-tuple *(sbj_label, perm, tclass, obj_label)*. Each group is further clustered by *sbjs* and *objs* to create subgroups that share the detailed semantics to derive fine-grained labels.

Our current generator prototype groups *sbjs* and derives fine-grained subject labels for built-in, vendor, and untrusted applications and binaries separately. The generator also groups file-like objects (e.g., `file`, `dir`, `blk_file`), which comprise the majority of tclasses. Group is performed using a *longest common prefix* search on file paths. This optimization helps to derive more fine-grained labels than provided by the general Android filesystem structure. Finally, the generator produces rules in the form of *(new_sbj_label, perm, tclass, new_obj_label)* as a policy refinement. If access patterns are matched with incomplete rules by the distance measurer, new rules merge with existing rules' permissions.

The generator handles benign and malicious patterns separately and generates `allow` and `neverallow` rules, respectively. Note that, it is possible that newly generated rules may conflict with existing rules due to incomplete or tightened access control. In such cases, policy analysts manually resolve conflicts (e.g., using `auditallow` to verify). Nevertheless, EASEAndroid exposes these conflicts with evidence collected through learning, therefore easing the policy refining process.

[3] In practice, the learning process can iterate multiple times with current audit logs. The generator caches all classified access patterns.

5 Evaluation

We implement a prototype of EASEAndroid and evaluate the learning capability and the security effectiveness of EASEAndroid from three perspectives:

1. We evaluate the coverage and precision of the classification result of EASEAndroid, and how they are affected by different threshold settings (Section 5.3).
2. We conduct a case study of the policy refinement generated by EASEAndroid, also comparing the generated rules with human-written rules (Section 5.4).
3. We further conduct a study on the new malicious access patterns classified by EASEAndroid and discuss several interesting new findings of attacks in the wild (Section 5.5).

5.1 Environment Setup

We build a prototype of EASEAndroid on an 8-node Hadoop cluster with each node having 8-core Xeon 2GHz, 32 GB memory. We use open source Cloudera Impala as the distributed SQL layer, with 10K SLOC Java as the learning layer. Parallelism is heavily employed for fast analytics. A data set of 1.3 million audit logs used in the following experiments are analyzed by EASEAndroid within 3 hours in a cold start.

5.2 Audit Log & Existing Knowledge

Audit Logs & Existing Policy We make use of 1.3 million audit logs over the entire 2014 from real-world devices running Android 4.3 (See Appendix A about audit log collection). All devices are loaded with an early version of Samsung SEAndroid policy (the policy remained unchanged) in enforcing mode[4]. The policy contains 5,094 `allow` rules and 59 `neverallow` rules developed by policy analysts. This policy is loaded as existing knowledge into EASEAndroid's knowledge base, used by the pattern-to-rule distance measurer.

The audit logs contain a total of over 14 million denied access events. After eliminating duplicate entries, we identify approximately 145K unique access events and further generalize them into 3,530 access patterns. For example, third-party app process ids under `/proc/` are generalized as `/proc/app_pid` in access patterns.

The subjects in the audit logs consist of 113 system (built-in) binaries, 1,182 external binaries (e.g., installed by `adb`), and 626 Android apps, which are captured because they perform system-level operations that do not go through Android framework/Dalvik VM (normal app operations are already allowed by the policy).

Initial Known Malicious Access Patterns In the initial knowledge base, we prepare a small set of known malicious access patterns as the initial seed to kick off learning. The set contains 9 confirmed exploit kits with their 17 malicious access patterns (e.g., psneuter CVE-2011-1149, Motochopper CVE-2013-2596, vroot CVE-2013-6282 and several exploit apps). Note that we do not have known benign access patterns initially as we rely on the `allow` rules in the existing policy.

Ground Truth To analyze the classification result of benign and malicious access patterns, we use a later version of human-written policy (6,337 `allow` rules, 94 `neverallow` rules) as the ground truth. We also consult with experienced policy analysts about the result.

5.3 Coverage & Precision of the Classification by EASEAndroid

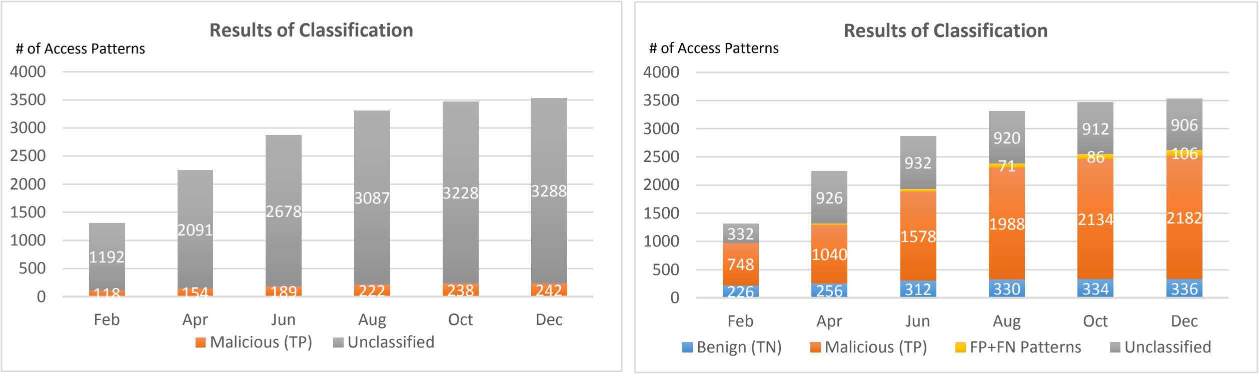
5.3.1 Coverage compared with naive matching

To illustrate the effect of EASEAndroid's learning coverage, we design a *naive matching* tool as a baseline to compare with EASEAndroid learning when both analyzing the same set of new access patterns from the audit logs, as shown in Figure 3. The naive matching tool is a dumb access pattern matching tool with no learning capability. It only uses the known subjects and access patterns in the initial knowledge base and can only match new access patterns directly related to them, based on the subjects and objects in the syscall entries in audit logs. In contrast, EASEAndroid starts from the initial knowledge base and keeps expanding the knowledge base.

As the audit logs are continuously collected over the year, we setup 6 analyses at a rate of every two months. Each analysis takes as input the accumulated audit logs from Jan 2014 to the current month (e.g., "Feb" is 2-month logs, "Apr" is 4-month logs, "Dec" is the entire year's logs). It is a typical scenario of semi-supervised learning with incremental input data. It also follows the nature that new benign/malicious patterns are gradually accumulated in audit logs over time.

As shown in Figure 3, EASEAndroid dramatically outperforms the naive matching in each analysis. As the total number of access pattern keeps increasing, EASEAndroid's coverage reaches about 74% in the final December analysis. EASEAndroid also discovers that the majority of denied access patterns in real world are malicious and they keep emerging while benign access patterns gradually stabilize. In contrast, the coverage of naive matching remains around 7%, because it can only match access patterns related to the initial known ones, the 9 exploit kits, which are updated with a small set of

[4] Some devices are found being rooted and may switch to permissive mode. See Section 5.5



(a) Access patterns matched by naive matching with no learning capability as the baseline. A small set of new malicious patterns are matched related to the 9 exploits since they are updated and keep trying some new malicious patterns over time. But the majority is still unclassified.

(b) Access patterns classified by EASEAndroid starting from the initial knowledge. New patterns classified in each analysis become the incremental knowledge to classify more patterns in the next analysis. Benign patterns get stable over time, while new malicious patterns keep emerging.

Figure 3: The comparison between naive matching and EASEAndroid on analyzing the same set of access patterns

Threshold Setting	Classified Malicious (TP+FP)	Classified Benign (TN+FN)	Remain Unclassified	True Malicious (TP)	False Malicious (FP)	True Benign (TN)	False Benign (FN)
$\sigma = 55\%, Dist \leq 2, c_{ij} > 0.55$	77.2%	14.0%	8.8%	62.96%	37.04%	58.65%	41.35%
$\sigma = 65\%, Dist \leq 1, c_{ij} > 0.65$	70.0%	11.8%	18.2%	88.73%	11.27%	71.35%	28.65%
$\sigma = 75\%, Dist \leq 1, c_{ij} > 0.75$	65.7%	10.9%	23.4%	91.35%	8.65%	88.92%	11.08%
$\sigma = 85\%, Dist = 0, c_{ij} > 0.85$	63.9%	10.5%	25.7%	96.81%	3.19%	90.81%	9.19%
$\sigma = 95\%, Dist = 0, c_{ij} > 0.95$	53.1%	9.2%	37.7%	97.27%	2.73%	100.00%	0.00%

Table 1: The coverage and precision of EASEAndroid with different threshold settings after comparison with ground truth. The first three columns summarize the overall classification coverage over the 3,530 patterns. The following four columns give more details about the percentages of each set of true/false-classified benign and malicious patterns. Row 4 is the threshold setting used in Figure 3(b).

new access patterns over time. But it still leaves the majority unclassified.

Specifically, all three learners of EASEAndroid contribute to the high classification coverage. In the first February analysis, EASEAndroid first matches 118 malicious access patterns, same as naive matching. Then it performs multiple learning iterations with the current audit logs in both automated and semi-automated mode. In summary, in automated mode, the NN classifier finds 282 patterns using threshold $(m,\sigma) = (10,85\%)$ in $IsMajorityKnown$. The pattern-to-rule distance measurer finds 95 patterns using $Dist(ap_i) = 0$ with existing `neverallow` rules. The co-occurrence learner finds 110 patterns with $c_{ij} > 0.85$. In semi-automated mode, we relax the thresholds to (10,75%), $Dist(ap_i) \leq 1$, $c_{ij} > 0.75$, respectively and further find 143 patterns based on the majority vote of the three learners.

As for benign access patterns in the February analysis, since the initial knowledge lacks benign patterns, the pattern-to-rule distance measurer classifies the first 23 benign patterns using $Dist(ap_i) = 0$ with existing `allow` rules and adds them to the knowledge base. Then the three learners contribute the remaining 203 using the same threshold settings.

Due to the strict thresholds, the automated mode classifies access patterns with no false positives or negatives. However, in the semi-automated mode, we do find 34 false-benign (False-Negative[5]) access patterns and the 72 false-malicious (False-Positive) ones in the final December analysis, mainly due to two reasons. First, a small set of access patterns are shared by both privileged benign system binaries and malicious apps with similar occurrences (e.g., both access `/proc/stat`), which make EASEAndroid hard to distinguish with relaxed thresholds. Second, some mis-classified patterns in early analyses affect the learning precision in later ones. In fact, there are only 4 false-malicious patterns in the February analysis. But then the NN classifier uses them to mistakenly find more false-malicious ones in the following analyses. Nevertheless, this limitation of the semi-automated mode is expected. Therefore in practice, the semi-automated mode requires policy analysts to verify the result to avoid error propagation. Analysts can also input extra constraints and knowledge about the access patterns of privileged system binaries to help EASEAndroid increase the precision.

There are still 906 access patterns unclassified in the

[5] We treat malicious as positive, benign as negative.

final analysis due to their low occurrence (less than five days throughout the year). After manual analysis, we find that some access patterns are likely malicious and might be isolated attack attempts in the wild. However, the statistics is too low to reach the threshold. In such cases, we have to wait for more similar access patterns coming in future audit logs. In Section 6, we discuss the limitation that isolated/targeted attacks may evade EASEAndroid's detection if they are not widely spread.

5.3.2 Coverage & precision with different threshold settings

The thresholds for the three learners play an important role on the coverage and the precision of EASEAndroid. In practice, it is important to find a balance between the coverage and the precision. In this section, we further investigate the detailed coverage and precision difference by choosing 5 different threshold settings from very relaxed to very strict as shown in Table 1. The listed threshold settings are for the automated mode. The semi-automated mode is relaxed by reducing 10% on both *IsMajorityKnown* (minimum neighbors unchanged) and c_{ij}, and increasing 1 in $Dist(ap_i)$. The first three columns show the overall percentage (adds to 100%) summarizing the classified malicious and benign and unclassified over the total 3,530 access patterns. The following four columns provide the more detailed TP/FP and TN/FN percentage of classified malicious and benign access patterns.

We can see that the thresholds in Row 1 is largely relaxed. Although it has the highest coverage, both FP (37.04%) and FN (41.35%) are too high, making it practically useless. In contrast, Row 5's thresholds achieve 100% correctness on classifying benign patterns. But it also leaves 37.7% patterns unclassified. The middle 3 rows are more balanced. Row 3 and 4 are candidates for practical use. In practice, analysts can also use multiple thresholds respectively, such as with Row 4 and 5 together, and only need to investigate the diff of their learning results since we have high confidence with the result of Row 5.

Admittedly, each individual threshold for each learner may have a different effect on the final classification result. To analyze more detailed threshold difference, or find an optimal vector of thresholds, a cross-validation [27] can be performed with multiple real-world audit log sets.

5.4 Case Study of Refinement Generation & Comparison with Human Policy

In the last December analysis in Figure 3, the policy refinement generator finally generates 51 new `allow` rules from the 336 benign access patterns, and 280 new `neverallow` rules from the 2,182 malicious access patterns, by extending identified incomplete rules and creating new fine-grained security labels to replace existing coarse-grained ones. In this section, we use the following example as a case study to illustrate the generated refinement.

EASEAndroid classified as benign 9 access patterns that read some time-zone data files under `/data/misc/zoneinfo`. These access patterns are found in multiple Android framework-related binaries (subjects) in `/system/bin`, including `surfaceflinger`,`dhcpcd`,`pppd` and a vendor-specific daemon. In the 6-tuples, the time-zone data files carry `system_data_file`, which is the default label for all files under `/data`. Naively generating a rule with this label (using `audit2allow`) would over-grant the subjects with permissions to access all files under `/data`.

EASEAndroid instead finds that these files all share the same `/data/misc/zoneinfo` file path prefix, and thus derives a new label `zoneinfo_file` specifically for them, adding to the labeling definition `file_contexts`:

```
/data/misc/zoneinfo/.* \
u:object_r:zoneinfo_file:s0
```

In practice, the full path prefix can be transformed into an underscore-joined label to keep the semantics and prevent conflict (though abbreviation may be required). EASEAndroid also creates a new attribute `access_zoneinfo_domain` to group the above subject domains, as the following:

```
attribute access_zoneinfo_domain;
typeattribute surfaceflinger \
access_zoneinfo_domain;...
```

Finally, EASEAndroid generates a new rule based on the new labels defined above:

```
allow access_zoneinfo_domain
zoneinfo_file:file {open read}
```

This rule only covers the 9 patterns observed by EASEAndroid, thus preventing unnecessary accesses being granted, following the least privilege principle.

The rules generated by EASEAndroid for the 336 benign access patterns are compared with human-written rules in the later policy version. Semantically, all access patterns allowed by EASEAndroid rules are also permitted by human-written rules. However, syntactically, EASEAndroid in general creates a larger set of more-specific rules, while human-written rules are more concise with the frequent use of policy `macros`, which ease the policy writing and are expanded during compile time [7]. It may be desirable to aggregate EASEAndroid's more specific rules for better human-readability; this remains for future work.

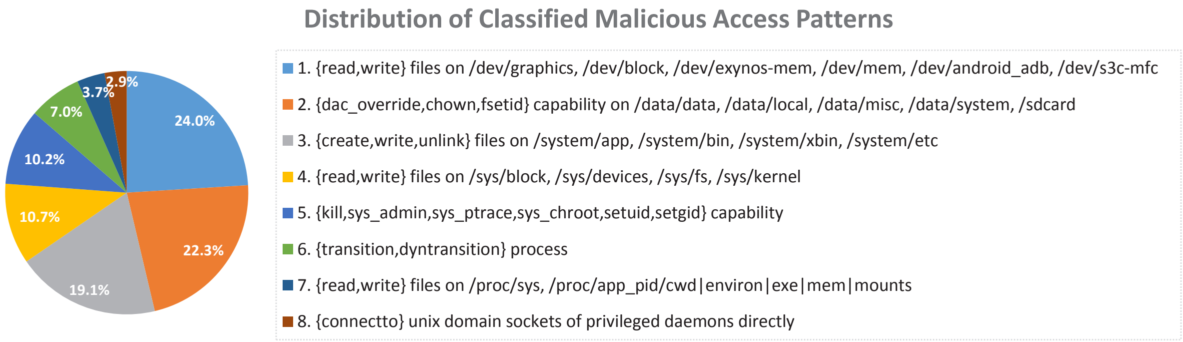


Figure 4: The distribution of malicious access patterns classified by EASEAndroid.

5.5 Case Study of Classified Malicious Access Patterns

For the one-year dataset of audit logs processed by EASEAndroid, 2,182 access patterns are classified as malicious. The reader is reminded that the starting point of analysis is 17 access patterns derived (manually) from 9 confirmed exploit kits. The access patterns newly classified as malicious by EASEAndroid capture malicious behavior much more precisely than has previously been possible. To the best of our knowledge, this is the first large-scale study of system-level malicious access patterns from real-world Android devices.

The subjects in these malicious access patterns are mostly untrusted third-party shell binaries and apps. For the purpose of understanding and discussion, they are categorized based on the permissions [5] (shown in braces) and the objects they accessed, which were mainly privileged files. Figure 4 shows the resulting 8 categories of malicious access patterns, each discussed below. Two of them (modify `/sys/fs/selinux` and `transition` to privileged domains) are new attacks in Android which directly target the SEAndroid MAC mechanism itself.

1. Exploit `/dev` nodes
The most common malicious access patterns are the ones that exploit various vulnerabilities in device nodes under `/dev`. For instance, EASEAndroid found 62 different shell binaries and exploit apps trying to directly read and write `/dev/graphics/*` (exploiting a previously-known framebuffer vulnerability). After identifying these subjects, EASEAndroid further discovered that they bundled various exploits targeting several other device nodes as well (including vendor-specific nodes). Some of these subjects were found to successfully gain root privileges. However, note that a good SEAndroid policy is still able to provide protection even on a rooted device (e.g., even `init` has limited permissions), as long as the Linux kernel is not compromised.[6]

2. Request file-related privileged capabilities
The second most frequent category of malicious access patterns are that subjects try to use privileged capabilities to modify the file mode bits and ownership of various files. This is a classic privilege escalation attack step; external binary files pushed to the device (e.g., in `/data/local/tmp`) may be given unintended capabilities, and important data files (e.g., in `/data/system`) can be made writable for attacks to proceed.

3. Modify `/system` partition
This is also a common step in exploits that the `/system` partition is modified with new binaries added such as `su,busybox`. Normally, the `/system` partition is mounted as read-only. But some subjects were able to remount the partition as writable. However, they were still captured by the audit logs because their domain labels were not allowed to write `system_file` under `/system`.

4. Access `/sys` filesystem
`/sys` is a virtual filesystem that exports kernel-level information to userspace, normally used by privileged system daemons. EASEAndroid found that untrusted subjects also try to directly access `/sys`, particularly `/sys/fs/selinux`, which contains the policy content and runtime state. Untrusted subjects may try to modify the policy content, either to switch to the permissive mode, or to get more permissions. We believe this is a new type of attack directly against SEAndroid MAC mechanism. Although this new attack is expected to emerge, it is still surprising to discover that the new attack has already become popular in the wild.

5. Request process-related privileged capabilities
EASEAndroid also found that some untrusted subjects ask for privileged process capabilities, such as `kill`'ing other processes, or `sys_admin` managing a list of functionalities [5]. The most common example is to use `sys_ptrace` to ptrace another process. This capability is attempted by several third-party management/moni-

[6] Since certain subjects gain root, they may be able to rollback to the permissive mode. The audit logs might just record the malicious access patterns but not actually block them.

tor apps, and game hacking apps (to modify other game apps' score/rewards).

6. Transition to privileged domains
Another new type of attack directly targeting SEAndroid is that untrusted apps try (some succeed) to gain more privileges by transforming their subject domains from `untrusted_app` to domains with higher privileges, including `init,init_shell,system_app`, and vendor daemon domains.[7] Interestingly, this case is found due to the conflict reported by the majority-vote in the semi-automated mode. An access pattern classified as malicious by both the NN classifier and the co-occurrence learner, is classified as benign by the pattern-to-rule distance measurer, because it is close to an `allow` rule. The conflict indicates that the subject carries a wrong domain.

7. Access `/proc` filesystem
Like `/sys`, `/proc` is also frequently accessed, especially by third-party management/monitor apps. Although reading `/proc/app_pid/*` might not be directly damaging, the information can be leveraged as a side-channel to compose attacks [20]. Besides, EASEAndroid also showed that certain apps try to write `/proc/sys/kernel/kptr_restrict` to gain access to the kernel symbol table, a common step in kernel exploits.

8. Connect to Unix sockets of privileged daemons
Unix domain socket is a more complicated case in SEAndroid. By design, some Unix sockets in system daemons such as `adbd,debuggerd` can be connected by apps, while others are reserved only for privileged daemons. EASEAndroid is able to distinguish these two cases, mainly by the co-occurrence learner. It found one new benign access pattern between two vendor daemons and several malicious ones that untrusted apps try to directly connect to Unix sockets of highly privileged daemons, such as `init`.

In summary, with EASEAndroid's learning, we find a group of interesting malicious access patterns and new attacks in Android. EASEAndroid also generates 280 fine-grained `neverallow` rules. 52 rules are found in the later policy. But others still require deeper investigation, since the knowledge learned by current EASEAndroid prototype may not be sufficient to understand the attack mechanisms behind these malicious access patterns.

6 Discussion

Blurred line between benign and malicious In practice, the line between benign and malicious might be blurred and subjective. It depends on specific security requirements and use cases to determine whether an access pattern is really benign or malicious. For example, individual users may like rooting their own devices and using the game hacking apps mentioned above, while game developers treat them as malicious because they bypass the in-app purchase. Nevertheless, EASEAndroid's learning provides more detailed evidence of access patterns' semantics for policy analysts to make the final decision.

Information missed by audit logs EASEAndroid relies on audit logs to learn new access patterns and derive policy refinements. However, two types of information could be missed or not available in audit logs, which can cause EASEAndroid to miss important knowledge. First, by default, audit logs only capture system-level operations that are denied by the policy currently loaded in a device. If the policy is too permissive or has too coarse-grained allow rules, malicious access patterns could be mistakenly allowed and missed by audit logs. To mitigate this issue, policy analysts should use `auditallow` to mark coarse-grained or uncertain rules so that audit logs can capture the operations allowed by these rules for EASEAndroid's analysis.

Second, framework-level operations are not available in audit logs, because they are controlled by Android permission model. But these upper-level operations contain valuable semantics (e.g., attack mechanisms). Without them, it is difficult to explain and distinguish certain benign/malicious access patterns in audit logs. Since Android 5.0, `logcat` is involved in SEAndroid auditing. In future, EASEAndroid can integrate logs from `logcat` to have more semantics in the knowledge base.

Countermeasure against EASEAndroid Similar to tampering virus sampling in AntiVirus programs, attackers can disable or compromise the audit log mechanism (logging and uploading) to avoid malicious access patterns being learned. Currently, we rely on Linux kernel protection [11] to ensure the integrity of audit log mechanism. And we argue that enabling audit log with policy refinement updates is a recommended security service for the majority users to have the latest security protection (or mandatory for enterprise users).

By design, if a malicious access pattern is widely spread and affects a large number of normal users, audit logs can catch the pattern for EASEAndroid to analyze. However, it is possible that isolated or targeted attacks may evade the detection of EASEAndroid if they are not popular enough to reach the thresholds, or deliberately avoid having semantic correlation with known malicious access patterns. In such cases, although EASEAndroid may leave them as unclassified, it still helps narrow down the scope for policy analysts to investigate. And policy analysts can input extra knowledge into EASEAndroid to help increase the coverage and precision.

It is also potentially possible that attackers manipulate the co-occurrence rate by intentionally forcing the be-

[7] Policy analysts suggest that it is also possible that some daemons may have zero-day vulnerabilities that are exploited to run attacks.

nign and malicious patterns to co-occur in one log, such as triggering the benign pattern first and then launching the attack. Such data poisoning attack may fool EASEAndroid's learning, which requires extra constraints or more logs from different devices to dilute the poisoned logs [14].

7 Related Work

Though SEAndroid is fairly new, SELinux has been developed and researched for years, including SELinux policy analysis and verification [10, 23, 36, 42], policy visualization [40], policy conflict resolving [26], policy simplifying [33, 34], policy comparison [19], policy information-flow integrity measurement [22, 24, 25, 37], etc. Also, the above research work usually assumes a relatively complete SELinux policy that has already been well developed. And the analysis usually focuses on stable desktop Linux system or only a few specific application programs (e.g., `sshd,httpd`). Due to the architecture difference, SEAndroid faces different challenges from SELinux, because current SEAndroid policy is still incomplete and under active development and continuous refinement.

In terms of SELinux policy generation, Polgen proposed by MITRE is a tool that guides policy analysts to develop policies based on system call traces [39]. However, it does not have machine learning capability and only focuses on system call traces from a single application program, which is not scalable. Madison proposed by Redhat is an extension of `audit2allow` that can generate policy similar to the reference policy style, such as using `macros` [30]. However, like `audit2allow`, it cannot create new security labels to cover new access patterns.

There is very little SELinux research related to machine learning. Marouf et al. proposed a similar approach to Polgen that analyzes system call traces to simplify SELinux policy [32]. Markowsky et al. proposed an IDS system that uses SELinux denials as input to an SVM classifier to detect attacks [31]. But there is no policy analysis or refinement.

Android SafetyNet [1] is a new security service provided by Google, which includes analyzing SELinux logs collected from Android devices, though no specific technical details about the SELinux log analysis have been disclosed.

8 Conclusion

Developing SEAndroid policies is a non-trivial task. In this paper, we have proposed EASEAndroid, the first SEAndroid audit log analytic platform for automatic policy analysis and refinement. EASEAndroid innovatively applies semi-supervised learning to MAC policy development. It has been evaluated with 1.3 million audit logs from real-world devices. It successfully discovered over 2,500 new benign and malicious access patterns, generated 331 policy rules, and found 2 new attacks in the wild directly targeting SEAndroid MAC mechanism.

Acknowledgement
We would like to thank Michael Grace, Kunal Patel and Xiaoyong Zhou from Samsung Research America for their valuable input for this paper. We also like to thank the paper shepherd and anonymous reviewers for their support to publish this paper.

This work is done in Samsung Research America. All data used to conduct the experiments was handled according to Samsung strict policies as explained in Appendix A. William Enck's work in this paper is supported by NSF grant CNS-1253346. Douglas Reeves's work in this paper is supported by ARO under MURI grant W911NF-09-1-0525. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Samsung or the funding agencies.

References

[1] Android SafetyNet, Google. `https://developer.android.com/training/safetynet`.

[2] End User License Agreement, Samsung Software. `http://www.samsung.com/us/Legal/PH_Warranty_2_EULA_INDEVICE_SINGLE_EULA.pdf`.

[3] Samsung Privacy Policy. `http://www.samsung.com/us/common/privacy.html`.

[4] Security-Enhanced Linux in Android. `https://source.android.com/devices/tech/security/selinux`.

[5] SELinux ObjectClassesPerms. `http://selinuxproject.org/page/ObjectClassesPerms`.

[6] SELinux Project. `http://selinuxproject.org`.

[7] SELinux Type Statements. `http://selinuxproject.org/page/TypeStatements`.

[8] setools, Tresys Technology. `https://github.com/TresysTechnology/setools`.

[9] Validating SELinux. `https://source.android.com/devices/tech/security/selinux/validate.html`.

[10] ALAM, M., SEIFERT, J.-P., LI, Q., AND ZHANG, X. Usage Control Platformization via Trustworthy SELinux. In *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security* (2008), ASIACCS '08, ACM, pp. 245–248.

[11] AZAB, A. M., NING, P., SHAH, J., CHEN, Q., BHUTKAR, R., GANESH, G., MA, J., AND SHEN, W. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), CCS '14, ACM, pp. 90–102.

[12] Badger, L., Sterne, D., Sherman, D., Walker, K., and Haghighat, S. A Domain and Type Enforcement UNIX Prototype. In *Proceedings of the Fifth USENIX UNIX Security Symposium* (June 1995).

[13] Bentley, J. L. A Survey of Techniques for Fixed Radius Near Neighbor Searching. Tech. rep., Stanford University, Stanford, CA, USA, 1975.

[14] Biggio, B., Corona, I., Fumera, G., Giacinto, G., and Roli, F. Bagging Classifiers for Fighting Poisoning Attacks in Adversarial Classification Tasks. In *Proceedings of the 10th International Conference on Multiple Classifier Systems* (Berlin, Heidelberg, 2011), MCS'11, Springer-Verlag, pp. 350–359.

[15] Bullinaria, J. A., and Levy, J. P. Extracting semantic representations from word co-occurrence statistics: A computational study. *Behavior Research Methods 39*, 3 (2007), 510–526.

[16] Carlson, A., Betteridge, J., Kisiel, B., Settles, B., Jr, E. R. H., and Mitchell, T. M. Toward an Architecture for Never-Ending Language Learning. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence* (2010), AAAI '10.

[17] Carlson, A., Betteridge, J., Wang, R. C., Mitchell, T. M., Carlos, S., and Brazil, S. P. Coupled Semi-Supervised Learning for Information Extraction. In *Proceedings of the third ACM international conference on Web search and data mining* (2010), WSDM '10, pp. 101–110.

[18] Chapelle, O., Scholkopf, B., and Zien, A. *Semi-Supervised Learning*. The MIT Press, Sept. 2006.

[19] Chen, H., Li, N., and Mao, Z. Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems. In *NDSS '09* (2009).

[20] Chen, Q. A., Qian, Z., and Mao, Z. M. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *USENIX Security '14* (2014), no. August, pp. 1037–1052.

[21] Dong, X. L., Gabrilovich, E., Heitz, G., Horn, W., Lao, N., Murphy, K., Strohmann, T., Sun, S., and Zhang, W. Knowledge Vault : A Web-Scale Approach to Probabilistic Knowledge Fusion. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining* (2014), KDD '14, pp. 601–610.

[22] Ganapathy, V., Jaeger, T., and Jha, S. Retrofitting Legacy Code for Authorization Policy Enforcement. In *2006 IEEE Symposium on Security and Privacy* (2006), S&P '06, Ieee, pp. 15 pp.–229.

[23] Hicks, B., Rueda, S., and Clair, L. S. A Logical Specification and Analysis for SELinux MLS Policy. *ACM Transactions on Information and System Security (TISSEC) 13*, 3 (2010), 1–31.

[24] Jaeger, T., Sailer, R., and Shankar, U. PRIMA: Policy-reduced Integrity Measurement Architecture. In *Proceedings of the eleventh ACM symposium on Access control models and technologies* (2006), SACMAT '06, pp. 19–28.

[25] Jaeger, T., Sailer, R., and Zhang, X. Analyzing Integrity Protection in the SELinux Example Policy. In *USENIX Security '03* (2003).

[26] Jaeger, T., Sailer, R., and Zhang, X. Resolving Constraint Conflicts. In *Proceedings of the ninth ACM symposium on Access control models and technologies* (New York, New York, USA, 2004), SACMAT '04, ACM Press, pp. 105–114.

[27] Kohavi, R. A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2* (San Francisco, CA, USA, 1995), IJCAI'95, pp. 1137–1143.

[28] Loscocco, P., and Smalley, S. Integrating Flexible Support for Security Policies into the Linux Operating System. In *USENIX Annual Technical Conference '01* (2001), no. February, pp. 29–42.

[29] Loscocco, P. A., Smalley, S. D., Muckelbauer, P. A., Taylor, R. C., Turner, S. J., and Farrell, J. F. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. In *Proceedings of the 21st National Information Systems Security Conference* (Oct. 1998).

[30] MacMillan, K. Madison : A New Approach to Policy Generation. In *SELnux Symposium '07* (2007).

[31] Markowsky, L. Towards Making SELinux Smart Leveraging SELinux to Protect End Nodes in a Federated Environment. Tech. rep., University of Maine, Orono, 2012.

[32] Marouf, S., Phuong, D. M., and Shehab, M. A Learning-Based Approach for SELinux Policy Optimization with Type Mining. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research* (New York, New York, USA, 2010), CSIIRW '10, ACM Press, p. 70.

[33] Nakamura, Y. Simplifying Policy Management with SELinux Policy Editor. In *SELinux Symposium '05* (2005), pp. 1–34.

[34] Nakamura, Y., Sameshima, Y., and Tabata, T. SEEdit: SELinux Security Policy Configuration System with Higher Level Language. In *Proceedings of the 23rd conference on Large installation system administration* (2009), LISA '09.

[35] Saltzer, J., and Schroeder, M. The Protection of Information in Computer Systems. *Proceedings of the IEEE 63*, 9 (Sept. 1975).

[36] Sasturkar, A., Stoller, S. D., Ramakrishnan, C. R., Science, C., and Brook, S. Policy Analysis for Administrative Role Based Access Control. In *19th IEEE Computer Security Foundations Workshop* (2006), CSFW '06.

[37] Shankar, U., Jaeger, T., and Sailer, R. Toward Automated Information-Flow Integrity Verification for Security-Critical Applications. In *NDSS '06* (2006).

[38] Smalley, S., and Craig, R. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *NDSS '13* (2013).

[39] Sniffen, B. T., Harris, D. R., and Ramsdell, J. D. Guided Policy Generation for Application Authors. In *SELnux Symposium '06* (2006).

[40] Xu, W., Shehab, M., and Ahn, G.-J. J. Visualization Based Policy Analysis: Case Study in SELinux. In *Proceedings of the 13th ACM Symposium on Access control models and technologies* (2008), SACMAT '08, pp. 165–174.

[41] Yianilos, P. N. Data structures and algorithms for nearest neighbor search in general metric spaces. In *ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms* (1993), vol. 93 of *SODA*, pp. 311–321.

[42] Zanin, G., La, R., Informatica, D., and Salaria, V. Towards a Formal Model for Security Policies Specification and Validation in the SELinux System. In *Proceedings of the ninth ACM symposium on Access control models and technologies* (2004), SACMAT '04, pp. 136–145.

[43] Zhou, Y., and Jiang, X. Dissecting android malware: Characterization and evolution. In *2012 IEEE Symposium on Security and Privacy* (2012), S&P '12, IEEE, pp. 95–109.

[44] Zhu, X. Semi-Supervised Learning Literature Survey. Tech. rep., University of Wisconsin - Madison, 2008.

A Data Collection Policies

Collection of audit logs used in this research strictly followed the Privacy Policy of Samsung [3], and conformed to the conditions described in Samsung's End User License Agreement [2]. Audit logs were collected anonymously from users who consented to provide diagnostic and usage data to help Samsung improve the quality and the performance of its products and services. Only Samsung authorized employees, using Samsung's internal computer systems, had access to the audit logs. No individual audit log information was released outside of Samsung while conducting the experiments described in this paper.

Marionette: A Programmable Network-Traffic Obfuscation System

Kevin P. Dyer
Portland State University
kdyer@cs.pdx.edu

Scott E. Coull
RedJack, LLC.
scott.coull@redjack.com

Thomas Shrimpton
Portland State University
teshrim@cs.pdx.edu

Abstract

Recently, a number of obfuscation systems have been developed to aid in censorship circumvention scenarios where encrypted network traffic is filtered. In this paper, we present *Marionette*, the first programmable network traffic obfuscation system capable of simultaneously controlling encrypted traffic features at a variety of levels, including ciphertext formats, stateful protocol semantics, and statistical properties. The behavior of the system is directed by a powerful type of probabilistic automata and specified in a user-friendly domain-specific language, which allows the user to easily adjust their obfuscation strategy to meet the unique needs of their network environment. In fact, the Marionette system is capable of emulating many existing obfuscation systems, and enables developers to explore a breadth of protocols and depth of traffic features that have, so far, been unattainable. We evaluate Marionette through a series of case studies inspired by censor capabilities demonstrated in the real-world and research literature, including passive network monitors, stateful proxies, and active probing. The results of our experiments not only show that Marionette provides outstanding flexibility and control over traffic features, but it is also capable of achieving throughput of up to 6.7Mbps when generating RFC-compliant cover traffic.

1 Introduction

Many countries have begun to view encrypted network services as a threat to the enforcement of information control and security policies. China [41] and Iran [7] are well-known for their efforts to block encrypted services like Tor [14], while other countries, such as the United Kingdom [18], have begun to express interest in blocking VPNs and anonymity systems. These discriminatory routing policies are empowered by analyzing traffic at both the network layer (e.g. TCP/IP headers) and, more recently, the application layer. The latter looks for specific features of packet payloads that act as a signature for the application-layer protocol being transported.

To combat application-layer filtering, several systems have been proposed to obfuscate packet payloads, and generally hide the true protocol being transported. Broadly speaking, these methods fall into one of three categories: those that use encryption to fully randomize the messages sent (e.g., obfs4 [34], ScrambleSuit [42], Dust [40]); those that tunnel traffic using existing software artifacts (e.g., FreeWave [21], Facet [24]); and those that use encryption in combination with some lightweight ciphertext formatting to make the traffic mimic an allowed protocol (e.g., FTE [15], StegoTorus [38]). A few of these systems have been deployed and are currently used by more comprehensive circumvention systems, such as Lantern [1], uProxy [5], and Tor [14].

Despite the progress these obfuscation systems represent, each of them suffers from one or more shortcomings that severely limit their ability to adapt to new network environments or censorship strategies. Lightweight obfuscation methods based on randomization fail in situations where protocol whitelisting is applied, as in the recent Iranian elections [13]. Tunneling systems are intimately tied to a specific protocol that may not always be permitted within the restrictive network environment, such as the case of Skype in Ethiopia [27]. Protocol-mimicry systems really only aim to mimic *individual* protocol messages, and therefore fail to traverse proxies that enforce stateful protocol semantics (e.g., Squid [39]) . Moreover, these systems can be quite brittle in the face of proxies that alter protocol messages in transit (e.g., altering message headers can render FTE [15] inoperable). In any case, all of the systems are incapable of changing their target protocol or traffic features without heavy system re-engineering and redeployment of code. This is a huge undertaking in censored networks.

	Message	**Stateful**	**Multi-layer**	**Traffic**	**Active**		**Goodput**
Case Study	**Content**	**Behavior**	**Control**	**Statistics**	**Probing**	**Protocol(s)**	**(Down/Up)**
Regex-Based DPI	✓	-	-	-	-	HTTP, SSH, SMB	68.2 / 68.2 Mbps
Proxy Traversal	✓	✓	-	-	-	HTTP	5.8 / 0.41 Mbps
Protocol Compliance	✓	✓	✓	-	-	FTP, POP3	6.6 / 6.7 Mbps
Traffic Analysis	✓	✓	✓	✓	-	HTTP	0.45 / 0.32 Mbps
Active Probing	✓	✓	✓	-	✓	HTTP, FTP, SSH	6.6 / 6.7 Mbps

Figure 1: Summary of Marionette case studies illustrating breadth of protocols and depth of feature control.

The Marionette System. To address these shortcomings, we develop the Marionette system. Marionette is a network-traffic obfuscation system that empowers users to rapidly explore a rich design space, without the need to deploy new code or re-design the underlying system.

The conceptual foundation of Marionette is a powerful kind of probabilistic automaton, loosely inspired by probabilistic input/output automata [45]. We use these to enforce (probabilistic) sequencing of individual ciphertext message types. Each transition between automata states has an associated block of actions to perform, such as encrypting and formatting a message, sampling from a distribution, or spawning other automata to support hierarchical composition. By composing automata, we achieve even more comprehensive control over mimicked protocol behaviors (e.g., multiple dependent channels) and statistical features of traffic. In addition, the automata admit distinguished error states, thereby providing an explicit mechanism for handling active attacks, such as censor-initiated "probing attacks."

At the level of individual ciphertext formats, we introduce another novel abstraction that supports fine-grained control. These *template grammars* are probabilistic context-free grammars (CFG) that compactly describes a language of templates for ciphertexts. Templates are strings that contain placeholder tokens, marking the positions where information (e.g., encrypted data bytes, dates, content-length values) may be embedded by user-specified routines. Adopting a CFG to describe templates has several benefits, including ease of deployment due to their compact representation, ability to directly translate grammars from available RFCs, and use of the grammar in receiver-side parsing tasks.

Everything is specified in a user-friendly domain-specific language (DSL), which enables rapid development and testing of new obfuscation strategies that are robust and responsive to future network monitoring tools. To encourage adoption and use of Marionette it has been made available as free and open source software[1].

Case studies. To display what is possible with Marionette, we provide several case studies that are inspired by recent research literature and real-world censor capabilities. These are summarized in Figure 1. For one example, we show that Marionette can implement passive-mode FTP by spawning multiple models that control interdependent TCP connections. For another, we use Marionette to mimic HTTP with enforced protocol semantics and resilience to message alteration, thereby successfully traversing HTTP proxies.

Our studies show that Marionette is capable of implementing a range of application-layer protocols, from HTTP to POP3, while also providing great depth in the range of traffic features it controls. Most importantly, it maintains this flexibility without unduly sacrificing performance – achieving up to 6.7Mbps while still maintaining fully RFC-compliant protocol semantics. We also show that the system performance is network-bound, and directly related to the constraints of the Marionette format being used.

Security Considerations. While our case studies are motivated by well-known types of adversaries, we avoid a formal security analysis of our framework for two reasons. First, the security of the system is intimately tied to the automata and template grammars specified by the user, as well as how the chosen protocols and features interact with the adversary. Second, any principled security analysis requires a generally accepted adversarial model. At the moment, the capabilities of adversaries in this space are poorly understood, and there are no formalized security goals to target. With that said, we believe our case studies represent a diverse sample of adversaries known to exist in practice, and hope that the flexibility of our system allows it to adapt to new adversaries faced in deployment. More fully understanding the limits of our system, and the adversaries it may face, is left for future work.

2 Related Work

In this section, we discuss previous work in the area of obfuscation and mimicry of application-layer protocols, as well as their common ancestry with network traffic generation research. The majority of systems aiming to avoid application-layer filtering are *non-programmable*, in the sense that they adopt one strategy at design-time

[1] `https://github.com/kpdyer/marionette/`

System		Blacklist DPI	Whitelist DPI	Statistical-test DPI	Protocol-enforcing Proxy	Multi-layer Control	High Throughput
Randomization	obfs2/3 [34]	✓	-	-	-	-	✓
	ScrambleSuit [42]	✓	-	✓	-	-	✓
	obfs4 [34]	✓	-	✓	-	-	✓
	Dust [40]	✓	-	✓	-	-	✓
Mimicry	SkypeMorph [26]	✓	✓	✓	-	-	-
	StegoTorus [38]	✓	✓	-	-	-	-
Tunneling	Freewave [21]	✓	✓	-	-	-	-
	Facet [24]	✓	✓	✓	-	-	-
	SWEET [47]	✓	✓	-	-	-	-
	JumpBox [25]	✓	✓	-	-	-	✓
	CensorSpoofer [36]	✓	✓	-	-	-	-
	CloudTransport [8]	✓	✓	-	✓	-	✓
Programmable	FTE [15]	✓	✓	-	-	-	✓
	Marionette	✓	✓	✓	✓	✓	✓

Figure 2: A comparison of features across randomization, mimicry, tunneling, and programmable obfuscation systems. A “✓" in the first four columns mean the system is appropriate for the indicated type of monitoring device; in the last two, it means that the system has the listed property. Multi-layer control is the ability to control features beyond single, independent connections. High-throughput systems are defined as any system capable of > 1Mbps throughput. Both FTE and Marionette can trade throughput for control over ciphertext traffic features.

and it cannot be changed without a major overhaul of the system and subsequent re-deployment. The non-programmable systems can be further subdivided into three categories based on their strategy: randomization, mimicry, or tunneling. A *programmable* system, however, allows for a variety of dynamically applied strategies, both randomization and mimicry-based, without the need for changes to the underlying software. Figure 2 presents a comparison of the available systems in each category, and we discuss each of them below. For those interested in a broader survey of circumvention and obfuscation technologies, we suggest recent work by Khattak et al. that discusses the space in greater detail [23].

Network Traffic Generation. Before beginning our discussion of obfuscation systems, it is important to point out the connection that they share with the broader area of network traffic generation. Most traffic generation systems focus on simple replay of captured network sessions [33, 19], replay with limited levels of message content synthesis [12, 31], generation of traffic mixes with specific statistical properties and static content [10, 37], or heavyweight emulation of user behavior with applications in virtualized environments [43]. As we will see, many mimicry and tunneling systems share similar strategies with the the key difference that they must also transport useful information to circumvent filtering.

Randomization. For systems implementing the randomization approach, the primary goal is to remove all static fingerprints in the content and statistical characteristics of the connection, effectively making the traffic look like “nothing.” The obfs2 and obfs3 [34] protocols were the first to implement this approach by re-encrypting standard Tor traffic with a stream cipher, thereby removing all indications of the underlying protocol from the content. Recently, improvements on this approach were proposed in the ScrambleSuit system [42] and obfs4 protocol [34], which implement similar content randomization, but also randomize the distribution of packet sizes and inter-arrival times to bypass both DPI and traffic analysis strategies implemented by the censor. The Dust system [40] also offers both content and statistical randomization, but does so on a per-packet, rather than per-connection basis. While these approaches provide fast and efficient obfuscation of the traffic, they only work in environments that block specific types of known-bad traffic (i.e., blacklists). In cases where a whitelist strategy is used to allow known-good protocols, these randomization approaches fail to bypass filtering, as was demonstrated during recent elections in Iran [13].

Mimicry. Another popular approach is to mimic certain characteristics of popular protocols, such as HTTP or Skype, so that blocking traffic with those characteristics would result in significant collateral damage. Mimicry-based systems typically perform shallow mimicry of only a protocol’s messages or the statistical properties of a single connection. As an example, StegoTorus [38] embeds data into the headers and payloads of a fixed set of previously collected HTTP messages, using various steganographic techniques. However, this provides no mechanism to control statistical properties, beyond what replaying of the filled-in message templates achieves. SkypeMorph [26], on the other hand, relies on the fact that Skype traffic is encrypted and focuses primarily on replicating the statistical features of packet sizes and timing. Ideally, these mimicked protocols would easily blend into the background traffic of the network, however research has shown that mimicked protocols can be distinguished from real versions of the same protocol using protocol semantics, dependencies among connections, and error conditions [20, 17]. In addition, they incur sometimes significant amounts of overhead due to the constraints of the content or statistical mimicry, which makes them much slower than randomization approaches.

Tunneling. Like mimicry-based systems, tunneling approaches rely on potential collateral damage caused by blocking popular protocols to avoid filtering. However, these systems tunnel their data in the payload of real instances of the target protocols. The Freewave [21] system, for example, uses Skype's voice channel to encode data, while Facet [24] uses the Skype video channel, SWEET [47] uses the body of email messages, and JumpBox [25] uses web browsers and live web servers. CensorSpoofer [36] also tunnels data over existing protocols, but uses a low-capacity email channel for upstream messages and a high-capacity VoIP channel for downstream. CloudTransport [8] uses a slightly different approach by tunneling data over critical (and consequently unblockable) cloud storage services, like Amazon S3, rather than a particular protocol. The tunneling-based systems have the advantage of using real implementations of their target protocols that naturally replicate all protocol semantics and other distinctive behaviors, and so they are much harder to distinguish. Even with this advantage, however, there are still cases where the tunneled data causes tell-tale changes to the protocol's behavior [17] or to the overall traffic mix through skewed bandwidth consumption. In general, tunneling approaches incur even more overhead than shallow mimicry systems since they are limited by the (low) capacity of the tunneling protocols.

Programmable Systems. Finally, programmable obfuscation systems combine the benefits of both randomization and mimicry-based systems by allowing the system to be configured to accommodate either strategy. Currently, the only system to implement programmable obfuscation is Format-Transforming Encryption (FTE) [15], which transforms encrypted data into a format dictated by a regular expression provided by the user. The approach has been demonstrated to have both high throughput and the ability to mimic a broad range of application-layer protocols, including randomized content. Unfortunately, FTE only focuses on altering the content of the application-layer messages, and not statistical properties, protocol semantics, or other potentially distinguishing traffic features.

Comparison with Marionette. Overall, each of these systems suffers from a common set of problems that we address with Marionette. For one, these systems, with the exception of FTE, force the user to choose a single target protocol to mimic without regard to the user's throughput needs, network restrictions, and background traffic mix. Moreover, many of the systems focus on only a fixed set of traffic features to control, usually only content and statical features of a single connection. In those cases where tunneling is used, the overhead and latency incurred often renders the channel virtually unusable for many common use cases, such as video streaming. The primary goal of Marionette, therefore, is not to develop a system that implements a single obfuscation method to defeat all possible censor strategies, but instead to provide the user with the ability to choose the obfuscation method that best fits their use case in terms of breadth of target protocols, depth of controlled traffic features, and overall network throughput.

3 Models and Actions

We aim for a system that enables broad control over several traffic properties, not just those of individual application-layer protocol messages. These properties may require that the system maintain some level of state about the interaction to enforce protocols semantics, or allow for non-deterministic behavior to match distributions of message size and timing. A natural approach to efficiently model this sort of stateful and nondeterministic system is a special type of probabilistic state machine, which we find to be well-suited to our needs and flexible enough to support a wide range of design approaches.

Marionette models. A *Marionette model* (or just model, for short) is a tuple $M = (Q, Q_{\rm nrm}, Q_{\rm err}, C, \Delta)$. The *state* set $Q = Q_{\rm nrm} \cup Q_{\rm err}$, where $Q_{\rm nrm}$ is the set of *normal states*, $Q_{\rm err}$ is the set of *error states*, and $Q_{\rm nrm} \cap Q_{\rm err} = \emptyset$. We assume that $Q_{\rm nrm}$ contains a distinguished start state, and that at least one of $Q_{\rm nrm}, Q_{\rm err}$ contains a distinguished finish state. The set C is the set of *actions*, which are (potentially) randomized algorithms. A string $\mathcal{B} = f_1 f_2 \cdots f_n \in C^*$ is called an *action-block*, and it defines a sequence of actions. Finally, Δ is a transition relation $\Delta \subseteq Q \times C^* \times (\mathsf{dist}(Q_{\rm nrm}) \cup \emptyset) \times \mathbb{P}(Q_{\rm err})$ where $\mathsf{dist}(X)$ the set of distributions over a set X, and $\mathbb{P}(X)$ is the powerset of X. The roles of $Q_{\rm nrm}$ and $Q_{\rm err}$ will be made clear shortly.

A tuple $(s, \mathcal{B}, (\mu_{\rm nrm}, S)) \in \Delta$ is interpreted as follows. When M is in state s, the action-block $\mathcal{B}$ may be executed and, upon completion, one samples a state $s'_{\rm nrm} \in Q_{\rm nrm}$ (according to distribution $\mu_{\rm nrm} \in$

$\mathsf{dist}(Q_{\mathrm{nrm}}))$. If the action-block fails, then an error state is chosen non-deterministically from S. Therefore, $\{s'_{\mathrm{nrm}}\} \cup S$ is the set of valid next states, and in this way our models have both proper probabilistic and non-deterministic choice, as in probabilistic input/output automata [45]. When $(s, \mathcal{B}, (\mu_{\mathrm{nrm}}, \emptyset)) \in \Delta$, then only transitions to states in Q_{nrm} are possible, and similarly for $(s, \mathcal{B}, (\emptyset, S))$ with transitions to states in Q_{err}.

In practice, normal states will be states of the model that are reached under normal, correct operation of the system. Error states are reached with the system detects an operational error, which may or may not be caused by an active adversary. For us, it will typically be the case that the results of the action-block $\mathcal{B}$ determine whether or not the system is operating normally or is in error, thus which of the possible next states is correct.

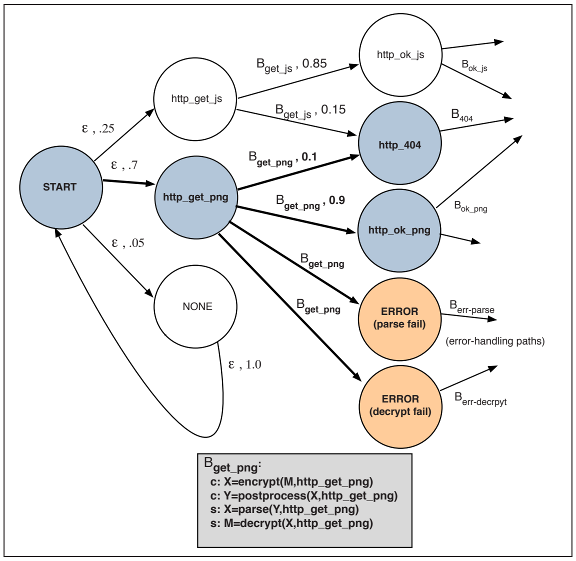


Figure 3: A partial graphical representation of a Marionette model for an HTTP exchange. (Transitions between http_get_js and error states dropped to avoid clutter.) The text discusses paths marked with bold arrows; normal states on these are blue, error states are orange.

Discussion. Marionette models support a broad variety of uses. One is to capture the intended state of a channel between two communicating parties (i.e., what message the channel should be holding at a given point in time). Such a model serves at least two related purposes. First, it serves to drive the implementation of procedures for either side of the channel. Second, it describes what a passive adversary would see (given implementations that realize the model), and gives the communicating parties some defense against active adversaries. The model tells a receiving party exactly what types of messages may be received next; receiving any other type of message (i.e., observing an invalid next channel state) provides a signal to commence error handling, or defensive measures.

Consider the partial model in Figure 3 for an exchange of ciphertexts that mimic various types of HTTP messages. The states of this model represent effective states of the shared channel (i.e., what message type is to appear next on the channel). Let us refer to the first-sender as the client, and the first-receiver as the server. In the beginning, both client and server are in the start state. The client moves to state http_get_js with probability 0.25, state http_get_png with probability 0.7, and state NONE with probability 0.05. In transitioning to any of these states, the empty action-block is executed (denoted by ε), meaning there are no actions on the transition. Note that, at this point, the server knows only the set $\{$http_get_js, http_get_png, NONE$\}$ of valid states and the probabilities with which they are selected.

Say that the client moves to state http_get_png, thus the message that should be placed on the channel is to be of the http_get_png type. The action-block $\mathcal{B}_{\textsf{get\_png}}$ gives the set of actions to be carried out in order to affect this. We have annotated the actions with "c:" and "s:" to make it clear which meant to be executed by the client and which are meant to be executed by the server, respectively. The client is to encrypt a message M using the parameters associated to the handle http_get_png, and then apply any necessary post-processing in order to produce the (ciphertext) message Y for sending. The server, is meant to parse the received Y (e.g. to undo whatever was done by the post-processing), and then to decrypt the result.

If parsing and decrypting succeed at the server, then it knows that the state selected by the client was http_get_png and, hence, that it should enter http_404 with probability 0.1, or http_ok_png with probability 0.9. If parsing fails at the server (i.e. the server action parse(Y,http_get_png) in action block $\mathcal{B}_{\textsf{get\_png}}$ fails) then the server must enter state ERROR (parse fail). If parsing succeeds but decryption fails (i.e., the server action decrypt(X,http_get_png) in action block $\mathcal{B}_{\textsf{get\_png}}$ fails) then the server must enter state ERROR (decrypt fail). At this point, it is the client who must keep alive a front of potential next states, namely the four just mentioned (error states are shaded orange in the figure). Whichever state the server chooses, the associated action-block is executed and progress through the model continues until it reaches the specified finish state.

Models provide a useful design abstraction for specifying allowable sequencings of ciphertext messages, as well as the particular actions that the communicating parties should realize in moving from message to message (e.g., encrypt or decrypt according to a particular ciphertext format). In practice, we do not expect sender and receiver instantiations of a given model will be identical. For example, probabilistic or nondeterministic choices made by the sender-side instantiation of a model (i.e., which transition was just followed) will need to be "determinized" by the receiver-side instantiation. This determinization process may need mechanisms to handle ambiguity. In Section 7 we will consider concrete specifications of models.

4 Templates and Template Grammars

In an effort to allow fined-grained control over the format of individual ciphertexts on the wire, we introduce the ideas of ciphertext-format templates, and grammars for creating them. *Templates* are, essentially, partially specified ciphertext strings. The unspecified portions are marked by special placeholders, and each placeholder will ultimately be replaced by an appropriate string, (e.g., a string representing a date, a hexadecimal value representing a color, a URL of a certain depth). To compactly represent a large set of these templates, we will use a probabilistic context-free grammar. Typically, a grammar will create templates sharing a common motif, such as HTTP request messages or CSS files.

Template Grammars. A template grammar $G = (V, \Sigma, R, S, p)$ is a probabilisitic CFG, and we refer to strings $T \in L(G)$ as templates. The set V is the set of non-terminals, and $S \in V$ is the starting non-terminal. The set $\Sigma = \overline{\Sigma} \cup P$ consists of two disjoint sets of symbols: $\overline{\Sigma}$ are the *base terminals*, and P is a set of *placeholder terminals* (or just placeholders). Collectively, we refer to Σ as template terminals. The set of rules R consists of pairs $(v, \beta) \in V \times (V \cup \Sigma)^*$, and we will sometimes adopt the standard notation $v \to \beta$ for these. Finally, the mapping $p\colon R \to (0, 1]$ associates to each rule a probability. We require that the sum of values $p(v, \cdot)$ for a fixed $v \in V$ and any second component is equal to one. For simplicity, we have assumed all probabilities are non-zero. The mapping p supports a method for sampling templates from $L(G)$. Namely, beginning with S, carry out a leftmost derivation and sample among the possible productions for a given rule according to the specified distribution.

Template grammars produce templates, but it is not templates that we place on the wire. Instead, a template T serves to define a set of strings in Σ^*, all of which share the same template-enforced structure. To produce these strings, each placeholder $\gamma \in P$ has associated to it a *handler*. Formally, a handler is a algorithm that takes as inputs a template $T \in \Sigma^*$ and (optionally) a bit string $c \in \{0, 1\}^*$, and outputs a string in Σ^* or the distinguished symbol $\perp$, which denotes error. A handler for γ scans T and, upon reading γ, computes a string in $s \in \Sigma^*$ and replaces γ with s. The handler halts upon reaching the end of T, and returns the new string T' that is T but will all occurrences of γ replaced. If a placeholder γ is to be replaced with a string from a particular set (say a dictionary of fixed strings, or an element of a regular language described by some regular expression), we assume the restrictions are built into the handler.

As an example, consider the following (overly simple) production rules that could be a subset of a context-free grammar for HTTP requests/responses.

$$\begin{array}{rcl} \langle\text{header}\rangle & \to & \langle\text{date\_prop}\rangle\text{: }\langle\text{date\_val}\rangle\texttt{\textbackslash r\textbackslash n} \\ & & \mid \langle\text{cookie\_prop}\rangle\text{: }\langle\text{cookie\_val}\rangle\texttt{\textbackslash r\textbackslash n} \\ \langle\text{date\_prop}\rangle & \to & \texttt{Date} \\ \langle\text{cookie\_prop}\rangle & \to & \texttt{Cookie} \\ \langle\text{date\_val}\rangle & \to & \gamma_{\text{date}} \\ \langle\text{cookie\_val}\rangle & \to & \gamma_{\text{cookie}} \end{array}$$

To handle our placeholders γ_{date} and γ_{cookie}, we might replace the former with the result of FTE["(Jan|Feb|...")], and the latter with the result of running FTE["([a-zA-Z...)"]. In this example our FTE-based handlers are responsible for replacing the placeholder with a ciphertext that is in the language of its input regular expression. To recover the data we parse the string according to the the template grammar rules, processing terminals in the resultant parse tree that correspond to placeholders.

5 System Architecture

In Section 3 we described how a Marionette model can be used to capture stateful and probabilistic communications between two parties. The notion of abstract actions (and action-blocks) gives us a way to use models generatively, too. In this section, we give a high-level description of an architecture that supports this use, so that we may transport arbitrary datastreams via ciphertexts that adhere to our models. We will discuss certain aspects of our design in detail in subsequent sections. Figure **??** provides a diagram of this client-server proxy architecture. In addition to models, this architecture consists of the following components:

- The client-side *driver* runs the main event loop, instantiates models (from a model specification file, see Section 6.3), and destructs them when they have reached the end of their execution. The complimentary receiver-side *broker* is responsible for listening to incoming connections and constructing and destructing models.

- *Plugins* are the mechanism that allow user-specified actions to be invoked in action-blocks. We discuss plugins in greater detail in Section 6.2.

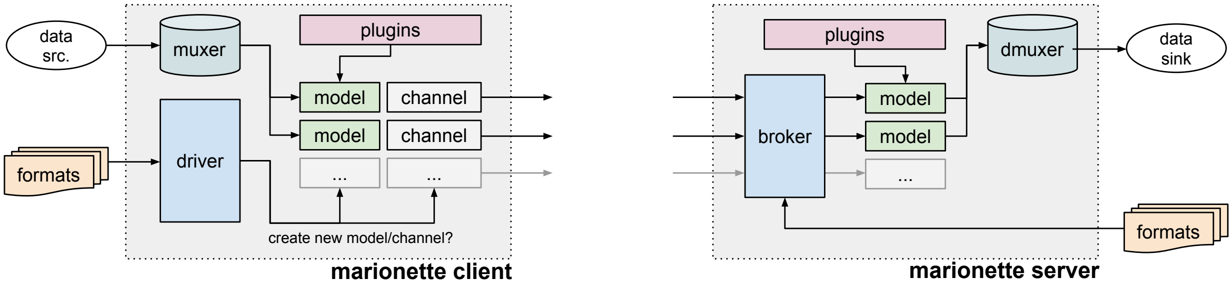


Figure 4: A high-level diagram of the Marionette client-server architecture and its major components for the client-server stream of communications in the Marionette system.

- The client-side *multiplexer* is an interface that allows plugins to serialize incoming datastreams into bitstrings of precise lengths, to be encoded into messages via plugins. The receiver-side *demultiplexer* parses and deserializes streams of cells to recover the underlying datastream. We discuss the implementation details of our (de)multiplexer in Section 6.1.

- A *channel* is a logical construct that connects Marionette models to real-world (e.g., TCP) data connections, and represents the communications between a specific pair of Marionette models. We note that, over the course of a channel's lifetime, it may be associated with multiple real-world connections.

Let's start by discussing how data traverses the components of a Marionette system. A datastream's first point of contact with the system is the incoming multiplexer, where it enters a FIFO buffer. Then a driver invokes a model that, in turn, invokes a plugin that wants to encode n bits of data into a message. Note that if the FIFO buffer is empty, the multiplexer returns a string that contains no payload data and is padded to n bits. The resultant message produced by the plugin is then relayed to the server. Server-side, the broker attempts to dispatch the received message to a model. There are three possible outcomes when the broker dispatches the message: (1) an active model is able to process it, (2) a new model needs to be spawned, or (3) an error has occurred and the message cannot be processed. In case 1 or 2, the cell is forwarded to the demultiplexer, and onward to its ultimate destination. In case 3, the server enters an error state for that message, where it can respond to a non-Marionette connection. We also note that the Marionette system can, in fact, operate with some of its components disabled. As an example, by disabling the multiplexer/demultiplexer we have a traffic generation system that doesn't carry actual data payloads, but generates traffic that abides by our model(s). This shows that there's a clear decoupling of our two main system features: control over cover traffic and relaying datastreams.

6 Implementation

Our implementation of Marionette consists of two command line applications, a client and server, which share a common codebase, and differ only in how they interpret a model. (e.g., initiate connection vs. receive connection) Given a model and its current state, each party determines the set of valid transitions and selects one according to the model's transition probabilities. In cases where normal transitions and error transitions are both valid, the normal transitions are preferred.

Our prototype of Marionette is written in roughly three thousand lines of Python code. All source code and engineering details are available as free and open-source software[2]. In this section, we will provide an overview of some of the major engineering obstacles we overcame to realize Marionette.

6.1 Record Layer

First, we will briefly describe the Marionette record layer and its objectives and design. Our record layer aims to achieve three goals: (1) enable multiplexing and reliability of multiple, simultaneous datastreams, (2) aid Marionette in negotiating and initializing models, and (3) provide privacy and authenticity of payload data. We implement the record layer using variable-length *cells*, as depicted in Figure 5, that are relayed between the client and server. In this section, we will walk through each of our goals and discuss how our record layer achieves them.

Multiplexing of datastreams. Our goal is to enable reliability and in-order delivery of datastreams that we tunnel through the Marionette system. If multiple streams are multiplexed over a single marionette channel, it must be capable of segmenting these streams. We achieve this by including a *datastream ID* and *datastream sequence number* in each cell, as depicted in Figure 5. Sender side, these values are populated at the time of the cell

[2] `https://github.com/kpdyer/marionette`

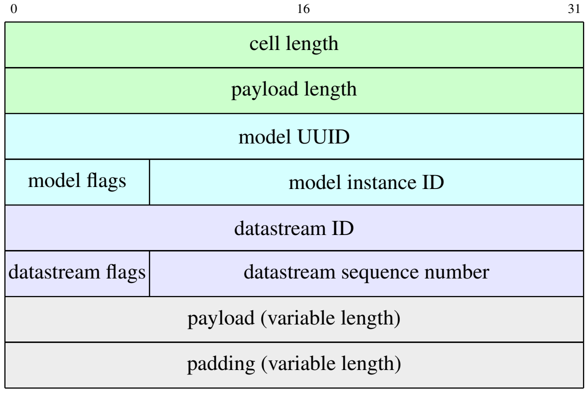


Figure 5: Format of the plaintext Marionette record layer cell.

creation. Receiver side, these values used to reassemble streams and delegate them to the appropriate data sink. The *datastream flags* field may have the value of OPEN, RELAY or CLOSE, to indicate the state of the datastream.

Negotiation and initialization of Marionette models. Upon accepting an incoming message, a Marionette receiver iterates through all transitions from the given model's start state. If one of the action blocks for a transition is successful, the underlying record layer (Figure 5) is recovered and then processed. The *model flags* field, in Figure 5, may have three values: START, RUNNING, or END. A START value is set when this is the first cell transmitted by this model, otherwise the value is set to RELAY until the final transmission of the model where an END is sent. The *model UUID* field is a global identifier that uniquely identifies the model that transmitted the message. The *model instance ID* is used to uniquely identify the instance of the model that relayed the cell from amongst all currently running instances of the model.

For practical purposes, in our proof of concept, we assume that a Marionette instance ID is created by either the client or server, but not both. By convention, the party that sends the first information-carrying message (i.e., first-sender) initiates the instance ID. Once established, the model instance ID has two potential uses. In settings where we have a proxy between the Marionette client and server, the instance ID can be used to determine the model that originated a message despite multiplexing performed by the proxy. In other settings, the instance ID can be used to enhance performance and seed a random number generator for shared randomness between the client and server.

Encryption of the cell. We encrypt each record-layer cell M using a slightly modified encrypt-then-MAC authenticated encryption scheme, namely $C = \mathsf{AES}_{K1}(\mathrm{IV}_1 \| \langle |M| \rangle) \| \mathsf{CTR}[\mathsf{AES}]^{\mathrm{IV}_2}_{K_1}(M) \| T$, where $\mathrm{IV}_1 = 0 \| R$ and $\mathrm{IV}_2 = 1 \| R$ for per-message random R. The first component of the encrypted record is a header. Here we use AES with key $K1$ to encrypt IV_1 along with an encoding of the length of M[3]. The second component is the record body, which is the counter-mode encryption of M under IV_2 and key $K1$, using AES as the underlying blockcipher[4]. Note that CTR can be length-preserving, not sending IV_2 as part of its output, because IV_2 is recoverable from IV_1. The third and component is an authentication tag T resulting from running $\mathsf{HMAC\text{-}SHA256}_{K2}$ over the entire record header and record body. One decrypts in the standard manner for encrypt-then-MAC.

6.2 Plugins

User-specified plugins are used to execute actions described in each model's action blocks. A plugin is called by the Marionette system with four parameters: the current channel, global variables shared across all active models, local variables scoped to our specific model, and the input parameters for this specific plugin (e.g., the FTE regex or the template grammar). It is the job of the plugin to attempt its action given the input parameters. By using global and local dictionaries, plugins can maintain long-term state and even enable message passing between models. We place few restrictions on plugins, however we do require that if a plugin fails (e.g., couldn't receive a message) it must return a failure flag and revert any changes it made when attempting to perform the action. Meanwhile, if it encounters a fatal error (e.g., channel is unexpectedly closed) then it must throw an exception.

To enable multi-level models, we provide a *spawn* plugin that can be used to spawn new model instances. In addition, we provide *puts* and *gets* for the purpose of transmitting static strings. As one example, this can be used to transmit a static, non-information carrying banner to emulate an FTP server. In addition, we implemented FTE and template grammars (Section 4) as our primary message-level plugins. Each plugin has a synchronous (i.e., blocking) and asynchronous (i.e., non-blocking) implementation. The FTE plugin is a wrapper around the FTE[5] and regex2dfa[6] libraries used by the Tor Project for FTE [15].

[3] One could also use the cell-length field in place of $\langle |M| \rangle$.

[4] Since $\mathrm{IV}_1 \neq \mathrm{IV}_2$ we enforce domain separation between the uses of AES_{K1}. Without this we would need an extra key.

[5] `https://github.com/kpdyer/libfte`

[6] `https://github.com/kpdyer/regex2dfa`

6.3 The Marionette DSL

Finally, we present a domain-specific language that can be used to compactly describe Marionette models. We refer to the formats that are created using this language as *Marionette model specifications* or *model specifications* for short. Figure 6 shows the Marionette modeling language syntax.

We have two primary, logical blocks in the model specification. The `connection` block is responsible for establishing model states, actions blocks that are executed upon a transition, and transition probabilities. An `error` transition may be specified for each state and is taken if all other potential transitions encounter a fatal error. The `action` block is responsible for defining a set of actions, which is a line for each party (client or server) and the plugin the party should execute. Let's illustrate the Marionette language by considering the following example.

```
connection([connection_type]):
    start [dst] [block_name] [prob | error]
    [src] [dst] [block_name] [prob | error]
    ...
    [src]  end  [block_name] [prob | error]

action [block_name]:
    [client | server] plugin(arg1, arg2, ...)
    ...
```

```
connection(tcp, 80):
    start         http_get_js   NULL    0.25
    start         http_get_png  NULL    0.7
    http_get_png  http_404      get_png 0.1
    http_get_png  http_ok_png   get_png 0.9
    http_ok_png   ...

action get_png:
    client fte.send("GET /\w+ HTTP/1\.1...")

action ok_png:
    server fte.send("HTTP/1\.1 200 OK...")

...
```

Figure 6: **Top:** The Marionette DSL. The connection block is responsible for establishing the Marionette model, its states and transitions probabilities. Optionally, the `connection_type` parameter specifies the type of channel that will be used for the model. **Bottom:** The partial model specification that implements the model from Figure 3.

Example: Simple HTTP model specification. Recall the model in Figure 3, which (partially) captures an HTTP connection where the first client-server message is an HTTP get for a JS or PNG file. Translating the diagram into our Marionette language is a straightforward process. First, we establish our connection block and specify `tcp` and port `80` — the server listens on this port and the client connects to it. For each transition we create an entry in our connection block. As an example, we added a transition between the `http_get_png` and `http_404` state with probability 0.1. For this transition we execute the `get_png` action block. We repeat this process for all transitions in the model ensuring that we have the appropriate action block for each transition.

For each action block we use synchronous FTE. One party is sending, one is receiving, and neither party can advance to the next state until the action successfully completes. Marionette transparently handles the opening and closing of the underlying TCP connection.

7 Case Studies

We evaluate the Marionette implementation described in Section 6 by building model specifications for a breadth of scenarios: protocol misidentification against regex-based DPI, protocol compliance for complex stateful protocols, traversal of proxy systems that actively manipulate Marionette messages, controlling statistical features of traffic, and responding to network scanners. We then conclude this section with a performance analysis of the formats considered.

For each case study, we analyze the performance of Marionette for the given model specification using our testbed. In our testbed, we deployed our Marionette client and server on Amazon Web Services m3.2xlarge instances, in the us-west (Oregon) and us-east (N. Virginia) zones, respectively. These instances include 8 virtual CPUs based on the Xeon E5-2670 v2 (Ivy Bridge) processor at 2.5GHz and 30GB of memory. The average round-trip latency between the client and server was 75ms. Downstream and upstream *goodput* was measured by transmitting a 1MB file, and averaged across 100 trials. Due to space constraints we omit the full model specifications used in our experiments, but note that each of these specifications is available with the Marionette source code[7].

7.1 Regex-Based DPI

As our first case study, we confirm that Marionette is able to generate traffic that is misclassified by regex-based DPI as a target protocol of our choosing. We are reproducing the tests from [15], using the regular expressions referred to as *manual-http*, *manual-ssh* and *manual-smb* in order to provide a baseline for the performance of the Marionette system under the simplest of specifications. Using these regular expressions, we engineered a Mari-

[7] `https://github.com/kpdyer/marionette`

	Misclassification	
Target Protocol	**bro [28]**	**YAF [22]**
HTTP (manual-http from [15])	100%	100%
SSH (manual-ssh from [15])	100%	100%
SMB (manual-smb from [15])	100%	100%

Figure 7: Summary of misclassification using existing FTE formats for HTTP, SSH, and SMB.

onette model that invokes the non-blocking implementation of our FTE plugins.

For each configuration we generated 100 datastreams in our testbed and classified this traffic using bro [28] (version 2.3.2) and YAF [22] (version 2.7.1.) We considered it a success if the classifier reported the *manual-http* datastreams as HTTP, the *manual-ssh* datastreams as SSH, and so on. In all six cases (two classifiers, three protocols) we achieved 100% success. These results are summarized in Figure 7. All three formats exhibited similar performance characteristics, which is consistent with the results from [15]. On average, we achieved 68.2Mbps goodput for both the upstream and downstream directions, which actually exceeds the goodput reported in [15].

7.2 Protocol-Compliance

As our next test, we aim to achieve protocol compliance for scenarios that require a greater degree of inter-message and inter-connection state. In our testing we created model specifications for HTTP, POP3, and FTP that generate protocol-compliant (i.e., correctly classified by bro) network traffic. The FTP format was the most challenging of the three, so we will use it as our illustrative example.

An FTP session in passive mode uses two data connections: a control channel and a data channel. To enter passive mode a client issues the `PASV` command, and the server responds with an address in the form `(a,b,c,d,x,y)`. As defined by the FTP protocol [30], the client then connects to TCP port `a.b.c.d:(256*x+y)` to retrieve the file requested in the `GET` command.

Building our FTP model specification. In building our FTP model we encounter three unique challenges, compared to other protocols, such as HTTP:

1. FTP has a range of message types, including usernames, passwords, and arbitrary files, that could be used to encode data. In order to maximize potential encoding capacity, we must utilize multiple encoding strategies (e.g., FTE, template grammars, etc.)
2. The FTP protocol is stateful (i.e., message order matters) and has many message types (e.g., `USER`, `PASV`, etc,) which do not have the capacity to encode information.
3. Performing either an active or passive FTP file transfer requires establishing a new connection and maintaining appropriate inter-connection state.

To address the first challenge, we utilize Marionette's plugin architecture, including FTE, template grammars, multi-layer models, and the ability to send/receive static strings. To resolve the second, we rely on Marionette's ability to model stateful transitions and block until, say, a specific static string (e.g., the FTP server banner) has been sent/received. For the third, we rely not only on Marionette's ability to spawn a new model, but we also rely on inter-model communications. In fact, we can generate the listening port server-side on the the fly and communicate it in-band to the client via the `227 Entering Passive Mode (a,b,c,d,x,y)` command, which is processed by a client-side template-grammar handler to populate a client-side global variable. This global variable value is then used to inform the spawned model as to which server-side TCP port it should connect.

Our FTP model specification relies upon the upstream password field, and upstream (PUT) and downstream (GET) file transfers to relay data. In our testbed the FTP model achieved 6.6Mbps downstream and 6.7Mbps upstream goodput.

7.3 Proxy Traversal

As our next case study, we evaluate Marionette in a setting where a protocol-enforcing proxy is positioned between the client and server. Given the prevalence of the HTTP protocol and breadth of proxy systems available, we focus our attention on engineering Marionette model specifications that are able to traverse HTTP proxies.

When considering the presence of an HTTP proxy, there are at least five ways it could interfere with our communications. A proxy could: (1) add HTTP headers, (2) remove HTTP headers, (3) modify header or payload contents, (4) re-order/multiplex messages, or (5) drop messages. Marionette is able to handle each of these cases with only slight enhancements to the plugins we have already described.

We first considered using FTE to generate ciphertexts that are valid HTTP messages. However, FTE is sensitive to modifications to its ciphertexts. As an example, changing the case of a single character of an FTE ciphertext would result in FTE decryption failure. Hence, we need a more robust solution.

Fortunately, template grammars (Section 4) give us fine-grained control over ciphertexts and allows us to tolerate ciphertext modification, and our record layer (Section 6.1) provides mechanisms to deal with stream multiplexing, message re-ordering and data loss. This covers all five types of interference mentioned above.

Building our HTTP template grammar. As a proof of concept we developed four HTTP template grammars. Two languages that are HTTP-GET requests, one with a header field of `Connection: keep-alive` and one with `Connection: close`. We then created analogous HTTP-OK languages that have keep-alive and close headers. Our model oscillates between the keep-alive GET and OK states with probability 0.9, until it transitions from the keep-alive OK state to the GET close state, with probability 0.1

In all upstream messages we encode data into the URL and cookie fields using the FTE template grammar handler. Downstream we encode data in the payload body using the FTE handler and follow this with a separate handler to correctly populate the content-length field.

We provide receiver-side HTTP parsers that validate incoming HTTP messages (e.g., ensure content length is correct) and then extract the URL, cookie and payload fields. Then, we take each of these components and reassemble them into a complete message, independent of the order they appeared. That is, the order of the incoming headers does not matter.

Coping with multiplexing and re-ordering. The template grammar plugin resolves the majority of issues that we could encounter. However, it does not allow us to cope with cases where the proxy might re-order or multiplex messages. By multiplex, we mean that a proxy may interleave two or more Marionette TCP channels into a single TCP stream between the proxy and server. In such a case, we can no longer assume that two messages from the same incoming datastream are, in fact, two sequential messages from the same client model. Therefore, in the non-proxy setting there is a one-to-one mapping between channels and server-side Marionette model instances. In the proxied setting, the channel to model instance mapping may be one-to-many.

We are able to cope with this scenario by relying upon the non-determinism of our Marionette models, and our record layer. The server-side broker attempts to execute all action blocks for available transitions across all active models. If no active model was able to successfully process the incoming message, then the broker (Section 5) attempts to instantiate a new model for that message. In our plugins we must rely upon our record layer to determine success for each of these operations. This allows us to deal with cases where a message may successfully decode and decrypt, but the model instance ID field doesn't match the current model.

Testing with Squid HTTP proxy. We validated our HTTP model specification and broker/plugin enhancements against the Squid [39] caching proxy (version 3.4.9). The Squid caching proxy adds headers, removes header, alters headers and payload contents, and reorders/multiplexes datastreams. We generated 10,000 streams through the Squid proxy and did not encounter any unexpected issues, such as message loss.

In our testbed, our HTTP model specification for use with Squid proxy achieved 5.8Mbps downstream and 0.41Mbps upstream goodput, with the upstream bandwidth limited by the capacity of the HTTP request format.

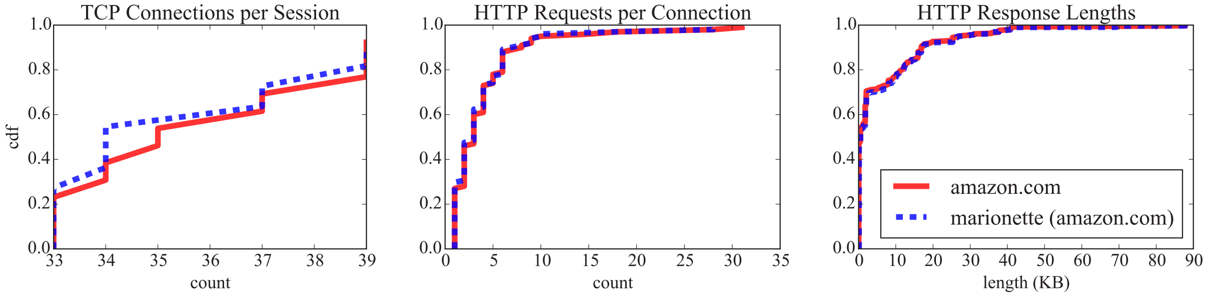
7.4 Traffic Analysis Resistance

In our next case study, we control statistical features of HTTP traffic. As our baseline, we visited `Amazon.com` with Firefox 35 ten times and captured all resultant network traffic[8]. We then post-processed the packet captures and recorded the following values: the lengths of HTTP response payloads, the number of HTTP request-response pairs per TCP connection, and the number of TCP connections generated as a result of each page visit. Our goal in this section is to utilize Marionette to model the traffic characteristics of these observed traffic patterns to make network sessions that "look like" a visit to `Amazon.com`. We will discuss each traffic characteristic individually, then combine them in a single model to mimic all characteristics simultaneously.

Message lengths. To model message lengths, we started with the HTTP response template grammar described in Section 7.3. We adapted the response body handler such that it takes an additional, integer value as input. This integer dictates the output length of the HTTP response body. On input n, the handler must return an HTTP response payload of exactly length n bytes.

From our packet captures of `Amazon.com` we recorded the message length for each observed HTTP response payload. Each time our new HTTP response template grammar was invoked by Marionette, we sampled from our recorded distribution of message lengths and used this value as input to the HTTP response template grammar. With this, we generate HTTP response payloads with lengths that match the distribution of those observed during our downloads of `Amazon.com`.

[8] Retrieval performed on February 21, 2015.


TCP Connections per Session

HTTP Requests per Connection

HTTP Response Lengths

Figure 8: A comparison of the aggregate traffic features for ten downloads of amazon.com using Firefox 35, compared to the traffic generated by ten executions of the Marionette model mimicking amazon.com.

Messages per TCP connection. We model the number of HTTP request-response pairs per TCP connection using the following strategy, which employs hierarchical modeling. Let's start with the case where we want to model a single TCP connection that has n HTTP request-response pairs. We start by creating a set of models which contain exactly n request-response pair with probability 1, for all n values of interest. We can achieve this by creating a model M_n with $n+1$ states, n transitions, and exactly one path. From the start state each transition results in an action block that performs one HTTP request-response. Therefore, M_n models a TCP connection with exactly n HTTP request-response pairs.

Then, we can employ Marionette's hierarchical model structure to have fine-grained control over the number of HTTP request-response pairs per connection. Let's say that we want to have n_1 request-response pairs with probability p_1, n_2 with probability p_2, and so on. For simplicity, we assume that all values n_i are unique, all values p_i are greater than 0, and $\Sigma_{i=0}^{m} p_i = 1$. For each possible value of n_i we create a model M_{n_i}, as described above. Then, we create a single parent model which has a start state with a transition that spawns M_{n_1} with probability p_1, M_{n_2} with probability p_2, and so on. This enables us to create a single, hierarchical model that that controls the number of request-response pairs for arbitrary distributions.

Simultaneously active connections. Finally, we aim to control the total number of connections generated by a model during an HTTP session. That is, we want our model to spawn n_i connections with probability p_i, according to some distribution dictated by our target. We achieve this by using the same hierarchical approach as the request-response pairs model, with the distinction that each child model now spawns n_i connections.

Building the model and its performance. For each statistical traffic feature, we analyzed the distribution of values in the packet captures from our `Amazon.com` visits. We then used the strategies in this section to construct a three-level hierarchical model that controls all of the traffic features simultaneously: message lengths, number of request-response pairs per connection, and the number of simultaneously active TCP connections. With this new model we deployed Marionette in our testbed and captured all network traffic it generated. In Figure 8 we have a comparison of the traffic features of the `Amazon.com` traffic, compared to the traffic generated by our Marionette model.

In our testbed, this model achieved 0.45Mbps downstream and 0.32Mbps upstream goodput. Compared to Section 7.3 this decrease in performance can be explained, in part, by the fact that `Amazon.com` has many connections with only a single HTTP request-response, and very short messages. As one example, the most common payload length in the distribution was 43 bytes. Consequently, the majority of the processing time was spent waiting for setup and teardown of TCP connections.

7.5 Resisting Application Fingerprinting

In our final case study, we evaluate Marionette's ability to resist adversaries that wish to identify Marionette servers using active probing or fingerprinting methods. We assume that an adversary is employing off-the-shelf tools to scan a target host and determine which services it is running. An adversary may have an initial goal to identify that a server is running Marionette and not an industry-standard service (e.g., Apache, etc.). Then, they may use this information to perform a secondary inspection or immediately block the server. This problem has been shown to be of great practical importance for services such as Tor [41] that wish to remain undetected in the presence of such active adversaries.

Our goal is to show that Marionette can coerce fingerprinting tools to incorrectly classify a Marionette server

```
connection(tcp, 8080):
  start    upstream       http_get    1.0
  upstream downstream     http_ok     1.0
  upstream downstream_err http_ok_err error
  ...

action http_ok_err:
  server io.puts("HTTP/1.1 200 OK\r\n" \
             + "Server: Apache/2.4.7\r\n..."
             ...
```

Figure 9: Example HTTP model specification including active probing resistance.

Protocol	Fingerprint Target	Scanner		
		nmap	Nessus	metasploit
HTTP	Apache 2.4.7	✓	✓	✓
FTP	Pure-FTPd 1.0.39	✓	✓	✓
SSH	OpenSSH 6.6.1	✓	✓	✓

Figure 10: A ✓ indicates that Marionette was able to successful coerce the fingerprinting tool into reporting that the Marionette server is the fingerprint target.

as a service of our choosing. As one example, we'll show that with slight embellishments to the formats we describe in Section 7.1 and Section 7.2, we can convince nmap [4] that Marionette is an instance of an Apache server.

7.5.1 Building Fingerprinting-Resistant Formats

In our exploration of fingerprinting attacks we consider three protocols: HTTP [16], SSH [46], and FTP [30]. For HTTP and SSH we started with the formats described in Section 7.1, and for FTP we started the format described in Section 7.2. We augmented these formats by adding an error transition (Section 3) that invokes an action that mimics the behavior of our target service. This error transition is traversed if all other potential transitions encounter fatal errors in their action blocks, which occur if an invalid message is received.

As an example, for our HTTP format we introduce an error transition to the `downstream_err` state. This transition is taken if the `http_ok` action block encounters a fatal error when attempting to invoke an FTE decryption. In this specific format, a fatal error in the `http_ok` action block is identified if an invalid message is detected when attempting to perform FTE decryption (i.e., doesn't match the regex or encounters a MAC failure). In the example found in Figure 9, upon encountering an error, we output the default response produced when requesting the index file from an Apache 2.4.7 server.

7.5.2 Fingerprinting Tools

For our evaluation we used nmap [4], Nessus [3], and metasploit [2], which are three commonly used tools for network reconnaissance and application fingerprinting. Our configuration was as follows.

nmap: We used nmap version 6.4.7 with version detection enabled and all fingerprinting probes enabled. We invoked nmap via the command line to scan our host. Nmap's service and version fields were used to identify its fingerprint of the target.

Nessus: For Nessus we used version 6.3.6 and performed a Basic Network Scan. We invoked Nessus via its REST API to start the scan and then asynchronously retrieved the scan with a second request. The reported fingerprint was determined by the `protocol` and `svc_name` for all plugins that were triggered.

metasploit: We used version 4.11.2 of metasploit. For fingerprinting SSH, FTP, and HTTP we used the `ssh_version` , `ftp_version` and `http_version` modules, respectively. For each module we set the RHOST and RPORT variable to our host and the reported fingerprint was the complete text string returned by the module.

7.5.3 Results

We refer to the *target* or *fingerprint target* as the application that we are attempting to mimic. To establish our fingerprint targets we installed Apache 2.4.7, Pure-FTPd 1.0.39 and OpenSSH 6.6.1 on a virtual machine. We then scanned each of these target applications with each of our three fingerprinting tools and stored the fingerprints.

To create our Marionette formats that mimic these targets, we added error states that respond identically to our target services. As an example, for our Apache 2.4.7, we respond with a success status code (200) if the client requests the `index.html` or `robots.txt` file. Otherwise we respond with a File Not Found (404) error code. Each server response includes a `Server: Apache 2.4.7` header. For our FTP and SSH formats we used a similar strategy. We observed the request initiated by each probe, and ensured that our error transitions triggered actions that are identical to our fingerprinting target.

We then invoked Marionette with our three new formats and scanned each of the listening instances with our fingerprinting tools. In order to claim success, we require two conditions. First, the three fingerprinting tools in our evaluation must report the exact same fingerprint as the target. Second, we require that a Marionette client must be able to connect to the server and relay data, as described in prior sections. We achieved this for all

Section	Protocol	Percent of Time Blocking on Network I/O	
		Client	Server
7.1	HTTP, SSH, etc.	56.9%	50.1%
7.2	FTP, POP3	90.1%	80.5%
7.3	HTTP	84.0%	96.8%
7.4	HTTP	65.5%	98.8%

Figure 11: Summary of case study formats and time spent blocking on network I/O for both client and server.

nine configurations (three protocols, three fingerprinting tools) and we summarize our results in Figure 10.

7.6 Performance

In our experiments, the performance of Marionette was dominated by two variables: (1) the structure of the model specification and (2) the client-server latency in our testbed. To illustrate the issue, consider our FTP format in Section 7.2 where we require nine back-and-forth messages in the FTP command channel before we can invoke a PASV FTP connection. This format requires a total of thirteen round trips (nine for our messages and four to establish the two TCP connections) before we can send our first downstream ciphertext. In our testbed, with a 75ms client-server latency, this means that (at least) 975ms elapse before we send any data. Therefore, a disproportionately large amount of time is spent blocking on network I/O.

In Figure 11 we give the percentage of time that our client and server were blocked due to network I/O, for each of the Marionette formats in our case studies. In the most extreme case, the Marionette server for the HTTP specification in Section 7.4 sits idle 98.8% of the time, waiting for network events. These results suggest that that certain Marionette formats (e.g., HTTP in Section 7.4) that target high-fidelity mimicry of protocol behaviors, network effects can dominate overall system performance. Appropriately balancing efficiency and realism is an important design consideration for Marionette formats.

8 Conclusion

The Marionette system is the first programmable obfuscation system to offer users the ability to control traffic features ranging from the format of individual application-layer messages to statistical features of connections to dependencies among multiple connections. In doing so, the user can choose the strategy that best suits their network environment and usage requirements. More importantly, Marionette achieves this flexibility without sacrificing performance beyond what is required to maintain the constraints of the model. This provides the user with an acceptable trade-off between depth of control over traffic features and network throughput. Our evaluation highlights the power of Marionette through a variety of case studies motivated by censorship techniques found in practice and the research literature. Here, we conclude by putting those experimental results into context by explicitly comparing them to the state of the art in application identification techniques, as well as highlighting the open questions that remain about the limitations of the Marionette system.

DPI. The most widely used method for application identification available to censors is DPI, which can search for content matching specified keywords or regular expressions. DPI technology is now available in a variety of networking products with support for traffic volumes reaching 30Gbps [11], and has been demonstrated in real-world censorship events by China [41] and Iran [7]. The Marionette system uses a novel template grammar system, along with a flexible plugin system, to control the format of the messages produced and how data is embedded into those messages. As such, the system can be programmed to produce messages that meet the requirements for a range of DPI signatures, as demonstrated in Sections 7.1 and 7.2.

Proxies and Application Firewalls. Many large enterprise networks implement more advanced proxy and application-layer firewall devices that are capable of deeper analysis of particular protocols, such as FTP, HTTP, and SMTP [39]. These devices can cache data to improve performance, apply protocol-specific content controls, and examine entire protocol sessions for indications of attacks targeted at the application. In many cases, the proxies and firewalls will rewrite headers to ensure compliance with protocol semantics, multiplex connections for improved efficiency, change protocol versions, and even alter content (e.g., HTTP chunking). Although these devices are not known to be used by nation-states, they are certainly capable of large traffic volumes (e.g., 400TB/day [6]) and could be used to block most current obfuscation and mimicry systems due to the changes they make to communications sessions. Marionette avoids these problems by using template grammars and a resilient record layer to combine several independent data-carrying fields into a message that is robust to reordering, changes to protocol headers, and connection multiplexing. The protocol compliance and proxy traversal capabilities of Marionette were demonstrated in Sections 7.2 and 7.3, respectively.

Advanced Techniques. Recent papers by Houmansadr et al. [20] and Geddes et al. [17] have presented a number of passive and active tests that a censor could use to identify mimicry systems. The passive tests include examination of dependent communication channels that are not present in many mimicry systems, such as a TCP control channel in the Skype protocol. Active tests include dropping packets or preemptively closing connections to elicit an expected action that the mimicked systems do not perform. Additionally, the networking community have been developing methods to tackle the problem of traffic identification for well over a decade [9], and specific methods have even been developed to target encrypted network traffic [44].

To this point, there has been no evidence that these more advanced methods have been applied in practice. This is likely due to two very difficult challenges. First, many of the traffic analysis techniques proposed in the literature require non-trivial amounts of state to be kept on every connection (e.g., packet size bi-gram distributions), as well as the use of machine learning algorithms that do not scale to the multi-gigabit traffic volumes of enterprise and backbone networks. As a point of comparison, the Bro IDS system [28], which uses DPI technology, has been known to have difficulties scaling to enterprise-level networks [35]. The second issue stems from the challenge of identifying rare events in large volumes of traffic, commonly referred to as the base-rate fallacy. That is, even a tiny false positive rate can generate an overwhelming amount of collateral damage when we consider traffic volumes in the 1 Gbps range. Sommer and Paxson [32] present an analysis of the issue in the context of network intrusion detection and Perry [29] for the case of website fingerprinting attacks.

Regardless of the current state of practice, there may be some cases where technological developments or a carefully controlled network environment enables the censor to apply these techniques. As we have shown in Section 7.4, however, the Marionette system is capable of controlling multiple statistical features not just within a single connection, but also across many simultaneous connections. We also demonstrate how our system can be programmed to spawn interdependent models across multiple connections in Section 7.2. Finally, in Section 7.5, we explored the use of error transitions in our models to respond to active probing and fingerprinting.

Future Work. While the case studies described in the previous section cover a range of potential adversaries, we note that there are still many open questions and potential limitations that have yet to be explored. For one, we do not have a complete understanding of the capabilities of the probabilistic I/O automata to model long-term state. These automata naturally exhibit the Markov property, but can also be spawned in a hierarchical manner with shared global and local variables, essentially providing much deeper conditional dependencies. Another area of exploration lies in the ability of template grammars to produce message content outside of simple message headers, potentially extending to context-sensitive languages found in practice. Similarly, there are many questions surrounding the development of the model specifications themselves since, as we saw in Section 7.6, these not only impact the unobservability of the traffic but also its efficiency and throughput.

References

[1] Lantern. `https://getlantern.org/`.

[2] metasploit. `http://www.metasploit.com/`.

[3] Nessus. `http://www.tenable.com/`.

[4] Nmap. `https://nmap.org/`.

[5] uproxy. `https://uproxy.org/`.

[6] Apache traffic server. `http://trafficserver.apache.org/`.

[7] Simurgh Aryan, Homa Aryan, and J. Alex Halderman. Internet censorship in iran: A first look. In *Presented as part of the 3rd USENIX Workshop on Free and Open Communications on the Internet*, Berkeley, CA, 2013. USENIX.

[8] Chad Brubaker, Amir Houmansadr, and Vitaly Shmatikov. Cloudtransport: Using cloud storage for censorship-resistant networking. In *Proceedings of the 14th Privacy Enhancing Technologies Symposium (PETS 2014)*, July 2014.

[9] A. Callado, C. Kamienski, G. Szabo, B. Gero, J. Kelner, S. Fernandes, and D. Sadok. A survey on internet traffic identification. *Communications Surveys Tutorials, IEEE*, 11(3):37–52, rd 2009.

[10] Jin Cao, William S. Cleveland, Yuan Gao, Kevin Jeffay, F. Donelson Smith, and Michele Weigle. Stochastic models for generating synthetic http source traffic. In *IN PROCEEDINGS OF IEEE INFOCOM*, 2004.

[11] Cisco sce 8000 service control engine. `http://www.cisco.com/c/en/us/products/collateral/service-exchange/sce-8000-series-service-control-engine/data_sheet_c78-492987.html`, June 2015.

[12] Weidong Cui, Vern Paxson, Nicholas Weaver, and Randy H. Katz. Protocol-independent adaptive replay of application dialog. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium (NDSS)*, February 2006.

[13] Holly Dagres. Iran induces internet 'coma' ahead of elections. `http://www.al-monitor.com/pulse/originals/2013/05/iran-internet-censorship-vpn.html`, May 2013.

[14] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *In Proceedings of the 13th USENIX Security Symposium*, 2004.

[15] Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. Protocol misidentification made easy with format-transforming encryption. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, November 2013.

[16] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999.

[17] John Geddes, Max Schuchard, and Nicholas Hopper. Cover your acks: Pitfalls of covert channel censorship circumvention. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, pages 361–372. ACM, 2013.

[18] Andrew Griffin. Whatsapp and imessage could be banned under new surveillance plans. The Independent, January 2015.

[19] Seung-Sun Hong and S. Felix Wu. On interactive internet traffic replay. In *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, RAID'05, pages 247–264, Berlin, Heidelberg, 2006. Springer-Verlag.

[20] Amir Houmansadr, Chad Brubaker, and Vitaly Shmatikov. The Parrot is Dead: Observing Unobservable Network Communications. In *The* 34^{th} *IEEE Symposium on Security and Privacy*, 2013.

[21] Amir Houmansadr, Thomas Riedl, Nikita Borisov, and Andrew Singer. I Want my Voice to be Heard: IP over Voice-over-IP for Unobservable Censorship Circumvention. In *Proceedings of the Network and Distributed System Security Symposium - NDSS'13*. Internet Society, February 2013.

[22] Christopher M. Inacio and Brian Trammell. Yaf: yet another flowmeter. In *Proceedings of the* 24^{th} *international conference on Large installation system administration*, LISA'10, 2010.

[23] Sheharbano Khattak, Mobin Javed, Philip D. Anderson, and Vern Paxson. Towards illuminating a censorship monitor's model to facilitate evasion. In *Presented as part of the 3rd USENIX Workshop on Free and Open Communications on the Internet*, Berkeley, CA, 2013. USENIX.

[24] Shuai Li, Mike Schliep, and Nick Hopper. Facet: Streaming over videoconferencing for censorship circumvention. In *Proceedings of the 12th Workshop on Privacy in the Electronic Society (WPES)*, November 2014.

[25] Jeroen Massar, Ian Mason, Linda Briesemeister, and Vinod Yegneswaran. Jumpbox–a seamless browser proxy for tor pluggable transports. *Security and Privacy in Communication Networks. Springer*, page 116, 2014.

[26] Hooman Mohajeri Moghaddam, Baiyu Li, Mohammad Derakhshani, and Ian Goldberg. Skypemorph: protocol obfuscation for tor bridges. In *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012.

[27] Katia Moskvitch. Ethiopia clamps down on skype and other internet use on tor. BBC News, June 2012.

[28] Vern Paxson. Bro: a system for detecting network intruders in real-time. In *Proceedings of the* 7^{th} *conference on USENIX Security Symposium - Volume 7*, SSYM'98, 1998.

[29] Mike Perry. A critique of website traffic fingerprinting attacks. `https://blog.torproject.org/`, November 2013.

[30] J. Postel and J. Reynolds. File Transfer Protocol. RFC 959 (Standard), October 1985. Updated by RFCs 2228, 2640, 2773, 3659.

[31] Sam Small, Joshua Mason, Fabian Monrose, Niels Provos, and Adam Stubblefield. To catch a predator: A natural language approach for eliciting malicious payloads. In *Proceedings of the 17th Conference on Security Symposium*, 2008.

[32] R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *Security and Privacy (SP), 2010 IEEE Symposium on*, 2010.

[33] Tcpreplay. `http://tcpreplay.synfin.net/`.

[34] Tor Project. Obfsproxy. `https://www.torproject.org/projects/obfsproxy.html.en`, 2015.

[35] Matthias Vallentin, Robin Sommer, Jason Lee, Craig Leres, Vern Paxson, and Brian Tierney. The nids cluster: Scalable, stateful network intrusion detection on commodity hardware. In *Recent Advances in Intrusion Detection*, pages 107–126. Springer, 2007.

[36] Qiyan Wang, Xun Gong, Giang Nguyen, Amir Houmansadr, and Nikita Borisov. CensorSpoofer: Asymmetric Communication using IP Spoofing for Censorship-Resistant Web Browsing. In *The* 19^{th} *ACM Conference on Computer and Communications Security*, 2012.

[37] Michele C. Weigle, Prashanth Adurthi, Félix Hernández-Campos, Kevin Jeffay, and F. Donelson Smith. Tmix: A tool for generating realistic tcp application workloads in ns-2. *SIGCOMM Comput. Commun. Rev.*, 36(3):65–76, July 2006.

[38] Zachary Weinberg, Jeffrey Wang, Vinod Yegneswaran, Linda Briesemeister, Steven Cheung, Frank Wang, and Dan Boneh. Stegotorus: a camouflage proxy for the tor anonymity system. In *ACM Conference on Computer and Communications Security*, 2012.

[39] D. Wessels and k. claffy. ICP and the Squid web cache. *IEEE Journal on Selected Areas in Communications*, 16(3):345–57, Mar 1998.

[40] Brandon Wiley. Dust: A blocking-resistant internet transport protocol. Technical report, School of Information, University of Texas at Austin, 2011.

[41] Philipp Winter and Stefan Lindskog. How the Great Firewall of China is Blocking Tor. In *Free and Open Communications on the Internet*, 2012.

[42] Philipp Winter, Tobias Pulls, and Juergen Fuss. Scramblesuit: a polymorphic network protocol to circumvent censorship. In *Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society*, pages 213–224. ACM, 2013.

[43] Charles V. Wright, Christopher Connelly, Timothy Braje, Jesse C. Rabek, Lee M. Rossey, and Robert K. Cunningham. Generating client workloads and high-fidelity network traffic for controllable, repeatable experiments in computer security. In Somesh Jha, Robin Sommer, and Christian Kreibich, editors, *Recent Advances in Intrusion Detection*, volume 6307 of *Lecture Notes in Computer Science*, pages 218–237. Springer Berlin Heidelberg, 2010.

[44] Charles V. Wright, Fabian Monrose, and Gerald M. Masson. On inferring application protocol behaviors in encrypted network traffic. *Journal on Machine Learning Research*, 7, December 2006.

[45] Sue-Hwey Wu, Scott A Smolka, and Eugene W Stark. Composition and behaviors of probabilistic i/o automata. *Theoretical Computer Science*, 176(1):1–38, 1997.

[46] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253 (Proposed Standard), January 2006.

[47] Wenxuan Zhou, Amir Houmansadr, Matthew Caesar, and Nikita Borisov. Sweet: Serving the web by exploiting email tunnels. *HotPETS. Springer*, 2013.

CONIKS: Bringing Key Transparency to End Users

Marcela S. Melara, Aaron Blankstein, Joseph Bonneau[†], Edward W. Felten, Michael J. Freedman

Princeton University, [†]Stanford University/Electronic Frontier Foundation

Abstract

We present CONIKS, an end-user key verification service capable of integration in end-to-end encrypted communication systems. CONIKS builds on transparency log proposals for web server certificates but solves several new challenges specific to key verification for end users. CONIKS obviates the need for global third-party monitors and enables users to efficiently monitor their own key bindings for consistency, downloading less than 20 kB per day to do so even for a provider with billions of users. CONIKS users and providers can collectively audit providers for non-equivocation, and this requires downloading a constant 2.5 kB per provider per day. Additionally, CONIKS preserves the level of privacy offered by today's major communication services, hiding the list of usernames present and even allowing providers to conceal the total number of users in the system.

1 Introduction

Billions of users now depend on online services for sensitive communication. While much of this traffic is transmitted encrypted via SSL/TLS, the vast majority is not end-to-end encrypted meaning service providers still have access to the plaintext in transit or storage. Not only are users exposed to the well-documented insecurity of certificate authorities managing TLS certificates [10, 11, 64], they also face data collection by communication providers for improved personalization and advertising [25] and government surveillance or censorship [24, 57].

Spurred by these security threats and users' desire for stronger security [43], several large services including Apple iMessage and WhatsApp have recently deployed end-to-end encryption [19, 62]. However, while these services have limited users' exposure to TLS failures and demonstrated that end-to-end encryption can be deployed with an excellent user experience, they still rely on a centralized directory of public keys maintained by the service provider. These key servers remain vulnerable to technical compromise [17, 48], and legal or extralegal pressure for access by surveillance agencies or others.

Despite its critical importance, secure key verification for end users remains an unsolved problem. Over two decades of experience with PGP email encryption [12, 55, 70] suggests that manual key verification is error-prone and irritating [22, 69]. The EFF's recent Secure Messaging Scorecard reported that none of 40 secure messaging apps which were evaluated have a practical and secure system for contact verification [50]. Similar conclusions were reached by a recent academic survey on key verification mechanisms [66].

To address this essential problem, we present CONIKS, a deployable and privacy-preserving system for end-user key verification.

Key directories with consistency. We retain the basic model of service providers issuing authoritative name-to-key bindings within their namespaces, but ensure that users can automatically verify *consistency* of their bindings. That is, given an *authenticated binding* issued by *foo.com* from the name *alice@foo.com* to one or more public keys, anybody can verify that this is the same binding for *alice@foo.com* that every other party observed.

Ensuring a stronger *correctness* property of bindings is impractical to automate as it would require users to verify that keys bound to the name *alice@foo.com* are genuinely controlled by an individual named Alice. Instead, with CONIKS, Bob can confidently use an authenticated binding for the name *alice@foo.com* because he knows Alice's software will monitor this binding and detect if it does not represent the key (or keys) Alice actually controls.

These bindings function somewhat like certificates in that users can present them to other users to set up a secure communication channel. However, unlike certificates, which present only an authoritative signature as a proof of validity, CONIKS bindings contain a cryptographic proof of consistency. To enable consistency checking, CONIKS servers periodically sign and publish an authenticated data structure encapsulating all bindings issued within their namespace, which all clients automatically verify is consistent with their expectations. If a CONIKS server ever tries to *equivocate* by issuing multiple bindings for a single username, this would require publishing distinct data structures which would provide irrefutable proof of the server's equivocation. CONIKS clients will detect the equivocation promptly with high probability.

Transparency solutions for web PKI. Several proposals seek to make the complete set of valid PKIX (SSL/TLS) certificates visible by use of public authenticated data structures often called *transparency logs* [4, 34, 38, 39, 53, 60]. The security model is similar to CONIKS in that publication does not ensure a certificate is correct, but users can accept it knowing the valid domain owner will promptly detect any certificate issued maliciously.

Follow-up proposals have incorporated more advanced features such as revocation [4, 34, 38, 60] and finer-grained limitations on certificate issuance [4, 34], but all have made several basic assumptions which make sense for web PKI but not for end-user key verification. Specifically, all of these systems make the set of names and keys/certificates completely public and rely to varying degrees on third-party monitors interested in ensuring the security of web PKI on the whole. End-user key verification has stricter requirements: there are hundreds of thousands of email providers and communication applications, most of which are too small to be monitored by independent parties and many of which would like to keep their users' names and public keys private.

CONIKS solves these two problems:

1. Efficient monitoring. All previous schemes include third-party monitors since monitoring the certificates/bindings issued for a single domain or user requires tracking the entire log. Webmasters might be willing to pay for this service or have their certificate authority provide it as an add-on benefit. For individual users, it is not clear who might provide this service free of charge or how users would choose such a monitoring service, which must be independent of their service provider itself.

CONIKS obviates this problem by using an efficient data structure, a *Merkle prefix tree*, which allows a single small proof (logarithmic in the total number of users) to guarantee the consistency of a user's entry in the directory. This allows users to monitor only their own entry without needing to rely on third parties to perform expensive monitoring of the entire tree. A user's device can automatically monitor the user's key binding and alert the user if unexpected keys are ever bound to their username.

2. Privacy-preserving key directories. In prior systems, third-party monitors must view the entire system log, which reveals the set of users who have been issued keys [34, 39, 53, 60]. CONIKS, on the contrary, is privacy-preserving. CONIKS clients may only query for individual usernames (which can be rate-limited and/or authenticated) and the response for any individual queries leaks no information about which other users exist or what key data is mapped to their username. CONIKS also naturally supports obfuscating the number of users and updates in a given directory.

CONIKS in Practice. We have built a prototype CONIKS system, which includes both the application-agnostic CONIKS server and an example CONIKS Chat application integrated into the OTR plug-in [8, 26, 65] for Pidgin [1]. Our CONIKS clients automatically monitor their directory entry by regularly downloading consistency proofs from the CONIKS server in the background, avoiding any explicit user action except in the case of notifications that a new key binding has been issued.

In addition to the strong security and privacy features, CONIKS is also efficient in terms of bandwidth, computation, and storage for clients and servers. Clients need to download about 17.6 kB per day from the CONIKS server and verifying key bindings can be done in milliseconds. Our prototype server implementation is able to easily support 10 million users (with 1% changing keys per day) on a commodity machine.

2 System Model and Design Goals

The goal of CONIKS is to provide a key verification system that facilitates practical, seamless, and secure communication for virtually all of today's users.

2.1 Participants and Assumptions

CONIKS's security model includes four main types of principals: identity providers, clients (specifically client software), auditors and users.

Identity Providers. Identity providers run CONIKS servers and manage disjoint namespaces, each of which has its own set of name-to-key bindings.[1] We assume a separate PKI exists for distributing providers' public keys, which they use to sign authenticated bindings and to transform users' names for privacy purposes.

While we assume that CONIKS providers may be malicious, we assume they have a reputation to protect and do not wish to attack their users in a public manner. Because CONIKS primarily provides transparency and enables reactive security in case of provider attacks, CONIKS cannot deter a service provider which is willing to attack its users openly (although it will expose the attacks).

Clients. Users run CONIKS client software on one or more trusted devices; CONIKS does not address the problem of compromised client endpoints. Clients *monitor* the consistency of their user's own bindings. To support monitoring, we assume that at least one of a user's clients has access to a reasonably accurate clock as well as access to secure local storage in which the client can save the results of prior checks.

We also assume clients have network access which cannot be reliably blocked by their communication provider. This is necessary for *whistleblowing* if a client detects

[1] Existing communication service providers can act as identity providers, although CONIKS also enables dedicated "stand-alone" identity providers to become part of the system.

misbehavior by an identity provider (more details in §4.2). CONIKS cannot ensure security if clients have no means of communication that is not under their communication provider's control.[2]

Auditors. To verify that identity providers are not equivocating, *auditors* track the chain of signed "snapshots" of the key directory. Auditors publish and gossip with other auditors to ensure global consistency. Indeed, CONIKS clients all serve as auditors for their own identity provider and providers audit each other. Third-party auditors are also able to participate if they desire.

Users. An important design strategy is to provide good baseline security which is accessible to nearly all users, necessarily requiring some security tradeoffs, with the opportunity for upgraded security for advanced users within the same system to avoid fragmenting the communication network. While there are many gradations possible, we draw a recurring distinction between *default users* and *strict users* to illustrate the differing security properties and usability challenges of the system.

We discuss the security tradeoffs between these two user security policies in §4.3.

2.2 Design Goals

The design goals of CONIKS are divided into security, privacy and deployability goals.

Security goals.

G1: Non-equivocation. An identity provider may attempt to equivocate by presenting diverging views of the name-to-key bindings in its namespace to different users. Because CONIKS providers issue signed, chained "snapshots" of each version of the key directory, any equivocation to two distinct parties must be maintained forever or else it will be detected by auditors who can then broadcast non-repudiable cryptographic evidence, ensuring that equivocation will be detected with high probability (see Appendix B for a detailed analysis).

G2: No spurious keys. If an identity provider inserts a malicious key binding for a given user, her client software will rapidly detect this and alert the user. For default users, this will not produce non-repudiable evidence as key changes are not necessarily cryptographically signed with a key controlled by the user. However, the user will still see evidence of the attack and can report it publicly. For strict users, all key changes must be signed by the user's previous key and therefore malicious bindings will not be accepted by other users.

Privacy goals.

G3: Privacy-preserving consistency proofs. CONIKS servers do not need to make any information about their bindings public in order to allow consistency verification. Specifically, an adversary who has obtained an arbitrary number of consistency proofs at a given time, even for adversarially chosen usernames, cannot learn any information about which other users exist in the namespace or what data is bound to their usernames.

G4: Concealed number of users. Identity providers may not wish to reveal their exact number of users. CONIKS allows providers to insert an arbitrary number of *dummy entries* into their key directory which are indistinguishable from real users (assuming goal G3 is met), exposing only an upper bound on the number of users.

Deployability goals.

G5: Strong security with human-readable names. With CONIKS, users of the system only need to learn their contacts' usernames in order to communicate with end-to-end encryption. They need not explicitly reason about keys. This enables seamless integration in end-to-end encrypted communication systems and requires no effort from users in normal operation.

G6: Efficiency. Computational and communication overhead should be minimized so that CONIKS is feasible to implement for identity providers using commodity servers and for clients on mobile devices. All overhead should scale at most logarithmically in the number of total users.

3 Core Data Structure Design

At a high level, CONIKS identity providers manage a directory of verifiable bindings of usernames to public keys. This directory is constructed as a Merkle prefix tree of all registered bindings in the provider's namespace.

At regular time intervals, or epochs, the identity provider generates a non-repudiable "snapshot" of the directory by digitally signing the root of the Merkle tree. We call this snapshot a *signed tree root* (STR) (see §3.3). Clients can use these STRs to check the consistency of key bindings in an efficient manner, obviating the need for clients to have access to the entire contents of the key directory. Each STR includes the hash of the previous STR, committing to a linear history of the directory.

To make the directory privacy-preserving, CONIKS employs two cryptographic primitives. First, a *private index* is computed for each username via a verifiable unpredictable function (described in §3.4). Each user's keys are stored at the associated private index rather than his or her username (or a hash of it). This prevents the data structure from leaking information about usernames. Second, to ensure that it is not possible to test if a users'

[2] Even given a communication provider who also controls all network access, it may be possible for users to whistleblow manually by reading information from their device and using a channel such as physical mail or sneakernet, but we will not model this in detail.

key data is equal to some known value even given this user's lookup index, a cryptographic *commitment*[3] to each user's key data is stored at the private index, rather than the public keys themselves.

3.1 Merkle Prefix Tree

CONIKS directories are constructed as Merkle binary prefix trees. Each node in the tree represents a unique prefix i. Each branch of the tree adds either a 0 or a 1 to the prefix of the parent node. There are three types of nodes, each of which is hashed slightly differently into a representative value using a collision-resistant hash $H()$:

Interior nodes exist for any prefix which is shared by more than one index present in the tree. An interior node is hashed as follows, committing to its two children:

$$h_{\text{interior}} = \mathrm{H}\left(h_{\text{child.0}}||h_{\text{child.1}}\right)$$

Empty nodes represent a prefix i of length ℓ (depth ℓ in the tree) which is not a prefix of any index included in the tree. Empty nodes are hashed as:

$$\mathrm{h}_{\text{empty}} = \mathrm{H}\left(k_{\text{empty}}||k_n||i||\ell\right)$$

Leaf nodes represent exactly one complete index i present in the tree at depth ℓ (meaning its first ℓ bits form a unique prefix). Leaf nodes are hashed as follows:

$$\mathrm{h}_{\text{leaf}} = \mathrm{H}\left(k_{\text{leaf}}||k_n||i||\ell||\mathrm{commit}(\mathrm{name}_i||\mathrm{keys}_i)\right)$$

where $\mathrm{commit}(\mathrm{name}_i||\mathrm{keys}_i)$ is a cryptographic commitment to the name and the associated key data. Committing to the name, rather than the index i, protects against collisions in the VUF used to generate i (see §3.4).

Collision attacks. While arbitrary collisions in the hash function are not useful, a malicious provider can mount a birthday attack to try to find two nodes with the same hash (for example by varying the randomness used in the key data commitment). Therefore, for t-bit security our hash function must produce at least $2t$ bits of output.

The inclusion of depths ℓ and prefixes i in leaf and empty nodes (as well as constants k_{empty} and k_{leaf} to distinguish the two) ensures that no node's pre-image can be valid at more than one location in the tree (including interior nodes, whose location is implicit given the embedded locations of all of its descendants). The use of a tree-wide nonce k_n ensures that no node's pre-image can be valid at the same location between two distinct trees which have chosen different nonces. Both are countermeasures for the *multi-instance setting* of an attacker attempting to find a collision at more than one location simultaneously.[4]. Uniquely encoding the location requires the attacker to target a specific epoch and location in the tree and ensures full t-bit security.

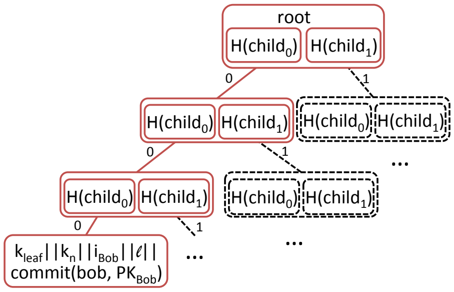


Figure 1: An authentication path from Bob's key entry to the root node of the Merkle prefix tree. Bob's index, i_{Bob}, has the prefix "000". Dotted nodes are not included in the proof's authentication path.

If the tree-wide nonce k_n is re-used between epochs, a parallel birthday attack is possible against each version of the tree. However, choosing a new k_n each epoch means that every node in the tree will change.

3.2 Proofs of Inclusion

Since clients no longer have a direct view on the contents of the key directory, CONIKS needs to be able to prove that a particular index exists in the tree. This is done by providing a proof of inclusion which consists of the complete *authentication path* between the corresponding leaf node and the root. This is a pruned tree containing the prefix path to the requested index, as shown in Figure 1. By itself, this path only reveals that an index exists in the directory, because the commitment hides the key data mapped to an index. To prove inclusion of the full binding, the server provides an opening of the commitment in addition to the authentication path.

Proofs of Absence. To prove that a given index j has no key data mapped to it, an authentication path is provided to the longest prefix match of j currently in the directory. That node will either be a leaf node at depth ℓ with an index $i \neq j$ which matches j in the first ℓ bits, or an empty node whose index i is a prefix of j.

3.3 Signed Tree Roots

At each epoch, the provider signs the root of the directory tree, as well as some metadata, using their directory-signing key SK_d. Specifically, an STR consists of

$$\mathrm{STR} = \mathrm{Sign}_{SK_d}\left(t||t_{prev}||\mathrm{root}_t||\mathrm{H}(\mathrm{STR}_{prev})||P\right)$$

[3] Commitments are a basic cryptographic primitive. A simple implementation computes a collision-resistant hash of the input data and a random nonce.

[4] This is inspired by Katz' analysis [33] of hash-based signature trees

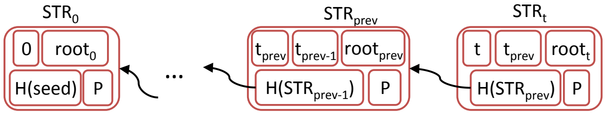


Figure 2: The directory's history is published as a linear hash chain of signed tree roots.

where t is the epoch number and P is a summary of this provider's current security policies. P may include, for example, the key K_{VUF} used to generate private indices, an expected time the next epoch will be published, as well as the cryptographic algorithms in use, protocol version numbers, and so forth. The previous epoch number t_{prev} must be included because epoch numbers need not be sequential (only increasing). In practice, our implementation uses UNIX timestamps.

By including the hash of the previous epoch's STR, the STRs form a *hash chain* committing to the entire history, as shown in Figure 2. This hash chain is used to ensure that if an identity provider ever equivocates by creating a fork in its history, the provider must maintain these forked hash chains for the rest of time (i.e. it must maintain *fork consistency* [41]). Otherwise, clients will immediately detect the equivocation when presented with an STR belonging to a different branch of the hash chain.

3.4 Private Index Calculation

A key design goal is to ensure that each authentication path reveals no information about whether any *other* names are present in the directory. If indices were computed using any publicly computable function of the username (such as a simple hash), each user's authentication path would reveal information about the presence of other users with prefixes "close" to that user.

For example, if a user *alice@foo.com*'s shortest unique prefix in the tree is i and her immediate neighbor in the tree is a non-empty node, this reveals that at least one users exists with the same prefix i. An attacker could hash a large number of potential usernames offline, searching for a potential username whose index shares this prefix i.

Private Indices. To prevent such leakage, we compute private indices using a *verifiable unpredictable function*, which is a function that requires a private key to compute but can then be publicly verified. VUFs are a simpler form of a stronger cryptographic construction called verifiable random functions (VRFs) [47]. In our application, we only need to ensure that a user's location in the tree is not predictable and do not need pseudorandomness (although statistical randomness helps to produce a balanced tree).

Given such a function VUF(), we generate the index i for a user u as:

$$i = \mathrm{H}\left(\mathrm{VUF}_{K_{\mathrm{VUF}}}(u)\right)$$

K_{VUF} is a public key belonging to the provider, and it is specified in the policy field of each STR. A hash function is used because indices are considered public and VUFs are not guaranteed to be one-way. A full proof of inclusion for user u therefore requires the value of VUF(u) in addition to the authentication path and an opening of the commitment to the user's key data.

We can implement a VUF using any deterministic, existentially unforgeable signature scheme [47]. The signature scheme must be deterministic or else the identity provider could insert multiple bindings for a user at different locations each with a valid authentication path. We discuss our choice for this primitive in §5.2.

Note that we might like our VUF to be collision-resistant to ensure that a malicious provider cannot produce two usernames u, u' which map to the same index. However, VUFs are not guaranteed to be collision-resistant given knowledge of the private key (and the ability to pick this key maliciously). To prevent any potential problems we commit to the username u in each leaf node. This ensures that only one of u or u' can be validly included in the tree even if the provider has crafted them to share an index.

4 CONIKS Operation

With the properties of key directories outlined in §3, CONIKS provides four efficient protocols that together allow end users to verify each other's keys to communicate securely: registration, lookup, monitoring and auditing. In these protocols, providers, clients and auditors collaborate to ensure that identity providers do not publish spurious keys, and maintain a single linear history of STRs.

4.1 Protocols

4.1.1 Registration and Temporary Bindings

CONIKS provides a registration protocol, which clients use to register a new name-to-key binding with an identity provider on behalf of its user, or to update the public key of the user's existing binding when revoking her key. An important deployability goal is for users to be able to communicate immediately after enrollment. This means users must be able to use new keys *before* they can be added to the key directory. An alternate approach would be to reduce the epoch time to a very short interval (on the order of seconds). However, we consider this undesirable both on the server end and in terms of client overhead.

CONIKS providers may issue *temporary bindings* without writing any data to the Merkle prefix tree. A temporary binding consists of:

$$\mathrm{TB} = \mathrm{Sign}_{K_d}\left(STR_t, i, k\right)$$

The binding includes the most recent signed tree root STR_t, the index i for the user's binding, and the user's new key information k. The binding is signed by the identity provider, creating a non-repudiable promise to add this data to the next version of the tree.

To register a user's key binding with a CONIKS identity provider, her client now participates in the following protocol. First, the client generates a key pair for the user and stores it in some secure storage on the device. Next, the client sends a registration request to the provider to the bind the public key to the user's online name, and if this name is not already taken in the provider's namespace, it returns a temporary binding for this key. The client then needs to wait for the next epoch and ensure that the provider has kept its promise of inserting Alice's binding into its key directory by the next epoch.

4.1.2 Key Lookups

Since CONIKS clients only regularly check directory roots for consistency, they need to ensure that public keys retrieved from the provider are contained in the most recently validated directory. Thus, whenever a CONIKS client looks up a user's public key to contact her client, the provider also returns a proof of inclusion showing that the retrieved binding is consistent with a specific STR. This way, if a malicious identity provider attempts to distribute a spurious key for a user, it is not able to do so without leaving evidence of the misbehavior. Any client that looks up this user's key and verifies that the binding is included in the presented STR will then promptly detect the attack.

In more detail, CONIKS's lookup protocol achieves this goal in three steps (summarized in Fig. 3). When a user wants to send a secure message to another user, her client first requests the recipient's public key at her provider. To allow the client to check whether the recipient's binding is included in the STR for the current epoch, the identity provider returns the full authentication path for the recipient's binding in the Merkle prefix tree along with the current STR. In the final step, the client recomputes the root of the tree using the authentication path and checks that this root is consistent with the presented STR. Note that, if the recipient has not registered a binding with the identity provider, it returns an authentication path as a proof of absence allowing the client to verify that the binding is indeed absent in the tree and consistent with the current STR.

4.1.3 Monitoring for Spurious Keys

CONIKS depends on the fact that each client monitors its own user's binding every epoch to ensure that her key binding has not changed unexpectedly. This prevents a malicious identity provider from inserting spurious keys

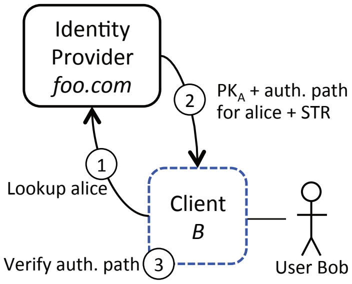


Figure 3: Steps taken when a client looks up a user's public key at her identity provider.

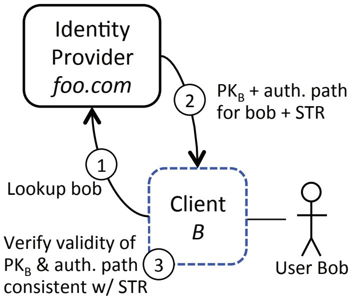


Figure 4: Steps taken when a client monitors its own user's binding for spurious keys every epoch.

that are properly included in the STR. Clients do not monitor other user's bindings as they may not have enough information to determine when another user's binding has changed unexpectedly.

Fig. 4 summarizes the steps taken during the monitoring protocol. The client begins monitoring by performing a key lookup for its own user's name to obtain a proof of inclusion for the user's binding. Next, the client checks the binding to ensure it represents the public key data the user believes is correct. In the simplest case, this is done by checking that a user's key is consistent between epochs. If the keys have not changed, or the client detects an authorized key change, the user need not be notified.

In the case of an unexpected key change, by default the user chooses what course of action to take as this change may reflect, for example, having recently enrolled a new device with a new key. Alternatively, security-conscious users may request a stricter key change policy which can be automatically enforced, and which we discuss further in §4.3. After checking the binding for spurious keys, the client verifies the authentication path as described in §3, including verifying the user's private index.

4.1.4 Auditing for Non-Equivocation

Even if a client monitors its own user's binding, it still needs to ensure that its user's identity provider is presenting consistent versions of its key directory to all participants in the system. Similarly, clients need to check

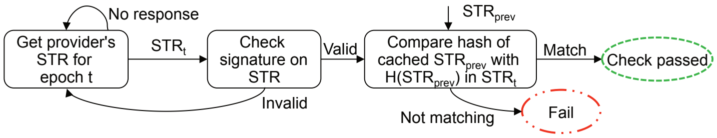

No response
Get provider's STR for epoch t
STR_t
Check signature on STR
Valid
Invalid
STR_{prev}
Compare hash of cached STR_{prev} with $H(STR_{prev})$ in STR_t
Match
Check passed
Not matching
Fail

Figure 5: Steps taken when verifying if a provider's STR history is linear in the auditing protocol.

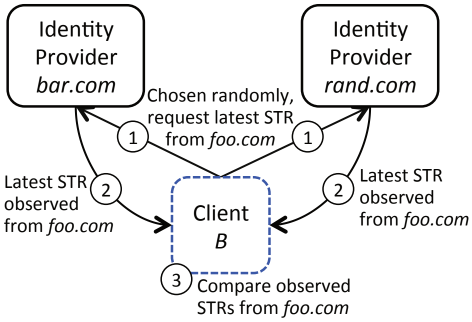

Identity Provider *bar.com*
Identity Provider *rand.com*
Chosen randomly, request latest STR from *foo.com*
1
1
Latest STR observed from *foo.com*
2
2
Latest STR observed from *foo.com*
Client *B*
3
Compare observed STRs from *foo.com*

Figure 6: Steps taken when comparing STRs in the auditing protocol.

that the identity provider of any user they contact is not equivocating about its directory. In other words, clients need to verify that any provider of interest is maintaining a linear STR history. Comparing each observed STR with every single other client with which a given client communicates would be a significant performance burden.

Therefore, CONIKS allows identity providers to facilitate auditing for their clients by acting as auditors of all CONIKS providers with which their users have been in communication (although it is also possible for any other entity to act as an auditor). Providers achieve this by distributing their most recent STR to other identity providers in the system at the beginning of every epoch.[5]

The auditing protocol in CONIKS checks whether an identity provider is maintaining a linear STR history. Identity providers perform the history verification whenever they observe a new STR from any other provider, while clients do so whenever they request the most recent STR from a specific identity provider directly. We summarize the steps required for an auditor to verify an STR history in Fig. 5. The auditor first ensures that the provider correctly signed the STR before checking whether the embedded hash of the previous epoch's STR matches what the auditor saw previously. If they do not match, the provider has generated a fork in its STR history.

Because each auditor has independently verified a provider's history, each has its own view of a provider's STR, so clients must perform an STR comparison to check for possible equivocation between these views (summarized in Fig. 6). Once a client has verified the provider's STR history is linear, the client queries one or more CONIKS identity providers at random.[6] The client asks the auditor for the most recent STR it observed from the provider in question. Because the auditor has already verified the provider's history, the client need not verify the STR received from the auditor. The client then compares the auditor's observed STR with the STR which the provider directly presented it. The client may repeat this process with different auditors as desired to increase confidence. For an analysis of the number of checks necessary to detect equivocation with high probability, see App. B.

CONIKS auditors store the current STRs of CONIKS providers; since the STRs are chained, maintaining the current STR commits to the entire history. Because this is a small, constant amount of data (less than 1 kB) it is efficient for a single machine to act as an auditor for thousands of CONIKS providers.

4.2 Secure Communication with CONIKS

When a user Bob wants to communicate with a user Alice via their CONIKS-backed secure messaging service *foo.com*, his client *client B* performs the following steps. We assume both Alice's and Bob's clients have registered their respective name-to-key bindings with *foo.com* as described in §4.1.1.

1. Periodically, *client B* checks the consistency of Bob's binding. To do so, the client first performs the monitoring protocol (per §4.1.3), and then it audits *foo.com* (per §4.1.4).
2. Before sending Bob's message to *client A*, *client B* looks up the public key for the username *alice* at *foo.com* (§4.1.2). It verifies the proof of inclusion for *alice* and performs the auditing protocol (§4.1.4) for *foo.com* if the STR received as part of the lookup is different or newer than the STR it observed for *foo.com* in its latest run of step 1.
3. If *client B* determines that Alice's binding is consistent, it encrypts Bob's message using *alice*'s public key and signs it using Bob's key. It then sends the message.

Performing checks after missed epochs. Because STRs are associated with each other across epochs, clients can "catch up" to the most recent epoch if they have not veri-

[5] CONIKS could support an auditing protocol in which clients directly exchange observed STRs, obviating the need of providers to act as auditors. The design of such a protocol is left as future work.

[6] We assume the client maintains a list of CONIKS providers acting as auditors from which it can choose any provider with equal probability. The larger this list, the harder it is for an adversary to guess which providers a client will query.

fied the consistency of a binding for several epochs. They do so by performing a series of the appropriate checks until they are sure that the proofs of inclusion and STRs they last verified are consistent with the more recent proofs. This is the only way a client can be sure that the security of its communication has not been compromised during the missed epochs.

Liveness. CONIKS servers may attempt to hide malicious behavior by ceasing to respond to queries. We provide flexible defense against this, as servers may also simply go down. Servers may publish an expected next epoch number with each STR in the policy section *P*. Clients must decide whether they will accept STRs published at a later time than previously indicated.

Whistleblowing. If a client ever discovers two inconsistent STRs (for example, two distinct versions signed for the same epoch time), they should notify the user and *whistleblow* by publishing them to all auditors they are able to contact. For example, clients could include them in messages sent to other clients, or they could explicitly send whistleblowing messages to other identity providers. We also envision out-of-band whistleblowing in which users publish inconsistent STRs via social media or other high-traffic sites. We leave the complete specification of a whistleblowing protocol for future work.

4.3 Multiple Security Options

CONIKS gives users the flexibility to choose the level of security they want to enforce with respect to key lookups and key change. For each functionality, we propose two security policies: a default policy and a strict policy, which have different tradeoffs of security and privacy against usability. All security policies are denoted by flags that are set as part of a user's directory entry, and the consistency checks allow users to verify that the flags do not change unexpectedly.

4.3.1 Visibility of Public Keys

Our goal is to enable the same level of privacy SMTP servers employ today,[7] in which usernames can be queried (subject to rate-limiting) but it is difficult to enumerate the entire list of names.

Users need to decide whether their public key(s) in the directory should be publicly visible. The difference between the default and the strict lookup policies is whether the user's public keys are encrypted with a secret symmetric key known only to the binding's owner and any other user of her choosing. For example, if the user Alice follows the default lookup policy, her public keys are not encrypted. Thus, anyone who knows Alice's name *alice@foo.com* can look up and obtain her keys from her *foo.com*'s directory. On the other hand, if Alice follows the strict lookup policy, her public keys are encrypted with a symmetric key only known to Alice and the users of her choosing.

Under both lookup policies, any user can verify the consistency of Alice's binding as described in §4, but if she enforces the strict policy, only those users with the symmetric key learn her public keys. The main advantage of the default policy is that it matches users' intuition about interacting with any user whose username they know without requiring explicit "permission". On the other hand, the strict lookup policy provides stronger privacy, but it requires additional action to distribute the symmetric key which protects her public keys.

4.3.2 Key Change

Dealing with key loss is a difficult quandary for any security system. Automatic key recovery is an indispensable option for the vast majority of users who cannot perpetually maintain a private key. Using password authentication or some other fallback method, users can request that identity providers change a user's public key in the event that the user's previous device was lost or destroyed. If Alice chooses the default key change policy, her identity provider *foo.com* accepts any key change statement in which the new key is signed by the previous key, as well as unsigned key change requests. Thus, *foo.com* should change the public key bound to *alice@foo.com* only upon her request, and it should reflect the update to Alice's binding by including a key change statement in her directory entry. The strict key change policy *requires* that Alice's client sign all of her key change statements with the key that is being changed. Thus, Alice's client only considers a new key to be valid if the key change statement has been authenticated by one of her public keys.

While the default key change policy makes it easy for users to recover from key loss and reclaim their username, it allows an identity provider to maliciously change a user's key and falsely claim that the user requested the operation. Only Alice can determine with certainty that she has not requested the new key (and password-based authentication means the server cannot prove Alice requested it). Still, her client will detect these updates and can notify Alice, making surreptitious key changes risky for identity providers to attempt. Requiring authenticated key changes, on the other hand, does sacrifice the ability for Alice to regain control of her username if her key is

[7] The SMTP protocol defines a *VRFY* command to query the existence of an email address at a given server. To protect user's privacy, however, it has long been recommended to ignore this command (reporting that any usernames exists if asked) [42].

ever lost. We discuss some implications for key loss for strict users in §6.

5 Implementation and Evaluation

CONIKS provides a framework for integrating key verification into communications services that support end-to-end encryption. To demonstrate the practicality of CONIKS and how it interacts with existing secure communications services, we implemented a prototype CONIKS Chat, a secure chat service based on the Off-the-Record Messaging [8] (OTR) plug-in for the Pidgin instant messaging client [1, 26]. We implemented a stand-alone CONIKS server in Java (∼2.5k sloc), and modified the OTR plug-in (∼2.2k sloc diff) to communicate with our server for key management. We have released a basic reference implementation of our prototype on Github.[8]

5.1 Implementation Details

CONIKS Chat consists of an enhanced OTR plug-in for the Pidgin chat client and a stand-alone CONIKS server which runs alongside an unmodified Tigase XMPP server. Clients and servers communicate using Google Protocol Buffers [2], allowing us to define specific message formats. We use our client and server implementations for our performance evaluation of CONIKS.

Our implementation of the CONIKS server provides the basic functionality of an identity provider. Every version of the directory (implemented as a Merkle prefix tree) as well as every generated STR are persisted in a MySQL database. The server supports key registration in the namespace of the XMPP service, and the directory efficiently generates the authentication path for proofs of inclusion and proofs of absence, both of which implicitly prove the proper construction of the directory. Our server implementation additionally supports STR exchanges between identity providers.

The CONIKS-OTR plug-in automatically registers a user's public key with the server upon the generation of a new key pair and automatically stores information about the user's binding locally on the client to facilitate future consistency checks. To facilitate CONIKS integration, we leave the DH-based key exchange protocol in OTR unchanged, but replace the socialist millionaires protocol used for key verification with a public key lookup at the CONIKS server. If two users, Alice and Bob, both having already registered their keys with the *coniks.org* identity provider, want to chat, Alice's client will automatically request a proof of inclusion for Bob's binding in *coniks.org*'s most recent version of the directory. Upon receipt of this proof, Alice's client automatically verifies the authentication path for Bob's name-to-key binding (as described in §4.1.2), and caches the newest information about Bob's binding if the consistency checks pass. If Bob has not registered his key with *coniks.org*, the client falls back to the original key verification mechanism. Additionally, Alice's client and Bob's clients automatically perform all monitoring and auditing checks for their respective bindings upon every login and cache the most recent proofs.

CONIKS Chat currently does not support key changes. Furthermore, our prototype only supports the default lookup policy for name-to-key bindings. Fully implementing these features is planned for the near future.

5.2 Choice of Cryptographic Primitives

To provide a 128-bit security level, we use SHA-256 as our hash function and EC-Schnorr signatures [21, 63].

Unfortunately Schnorr signatures (and related discrete-log based signature schemes like DSA [36]) are not immediately applicable as a VUF as they are not deterministic, requiring a random nonce which the server can choose arbitrarily.[9] In Appendix A we describe a discrete-log based scheme for producing a VUF (and indeed, a VRF) in the random-oracle model. Note that discrete-log based VUFs are longer than basic signatures: at a 128-bit security level using elliptic curves, we expect signatures of size 512 bits and VUF proofs of size 768 bits.

Alternately, we could employ a deterministic signature scheme like classic RSA signature [59] (using a deterministic padding scheme such as PKCS v. 1.5 [31]), although this is not particularly space-efficient at a 128-bit security level. Using RSA-2048 provides approximately 112 bits of security [3] with proofs of size 2048 bits. [10]

Using pairing-based crypto, BLS "short signatures" [7] are also deterministic and provide the best space efficiency with signature sizes of just 256 bits, making them an efficient choice both for signatures and VUF computations. BLS signatures also support *aggregation*, that is, multiple signatures with the same key can be compressed into a single signature, meaning the server can combine the signatures on n consecutive roots. However there is not widespread support for pairing calculations required for BLS, making it more difficult to standardize and deploy.

We evaluate performance in Table 1 in the next section for all three potential choices of signature/VUF scheme.

[8] `https://github.com/coniks-sys/coniks-ref-implementation`

[9] There are deterministic variants of Schnorr or DSA [5, 49] but these are not *verifiably* deterministic as they generate nonces pseudorandomly as a symmetric-key MAC of the data to be signed.

[10] We might tolerate slightly lower security in our VUF than our signature scheme, as this key only ensures privacy and not non-equivocation.

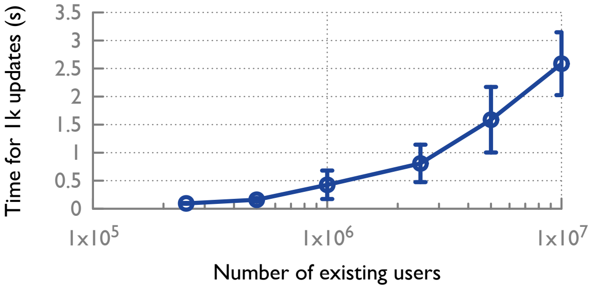


Figure 7: Mean time to re-compute the tree for a new epoch with 1K updated nodes. The x-axis is logarithmic and each data point is the mean of 10 executions. Error bars indicate standard deviation.

5.3 Performance Evaluation

To estimate the performance of CONIKS, we collect both theoretical and real performance characteristics of our prototype implementation. We evaluate client and server overheads with the following parameters:

- A single provider might support $N \approx 2^{32}$ users.
- Epochs occur roughly once per hour.
- Up to 1% of users change or add keys per day, meaning $n \approx 2^{21}$ directory updates in an average epoch.
- Servers use a 128-bit cryptographic security level.

Server Overheads. To measure how long it takes for a server to compute the changes for an epoch, we evaluated our server prototype on a 2.4 GHz Intel Xeon E5620 machine with 64 GB of RAM allotted to the OpenJDK 1.7 JVM. We executed batches of 1000 insertions (roughly 3 times the expected number of directory updates per epoch) into a Merkle prefix with 10 M users, and measured the time it took for the server to compute the next epoch.

Figure 7 shows the time to compute a version of the directory with 1000 new entries as the size of the original namespace varies. For a server with 10 M users, computing a new Merkle tree with 1000 insertions takes on average 2.6 *s*. As epochs only need to be computed every hour, this is not cumbersome for a large service provider. These numbers indicate that even with a relatively unoptimized implementation, a single machine is able to handle the additional overhead imposed by CONIKS for workloads similar in scale to a medium-sized communication providers (e.g., TextSecure) today.

While our prototype server implementation on a commodity machine comfortably supports 10M users, we note that due to the statistically random allocation of users to indices and the recursive nature of the tree structure, the task parallelizes near-perfectly and it would be trivial to scale horizontally with additional identical servers to compute a directory with billions of users.

Lookup Cost. Every time a client looks up a user's binding, it needs to download the current STR, a proof of of inclusion consisting of about $\lg_2(N)+1$ hashes plus one 96-byte VUF proof (proving the validity of the binding's private index). This will require downloading $32 \cdot (\lg_2(N)+1)+96 \approx 1216$ bytes. Verifying the proof will require up to $\lg_2(N)+1$ hash verifications on the authentication path as well as one VUF verification. On a 2 GHz Intel Core i7 laptop, verifying the authentication path returned by a server with 10 million users, required on average 159 μs (sampled over 1000 runs, with $\sigma = 30$). Verifying the signature takes approximately 400 μs, dominating the cost of verifying the authentication path. While mobile-phone clients would require more computation time, we do not believe this overhead presents a significant barrier to adoption.

Monitoring Cost. In order for any client to monitor the consistency of its own binding, it needs fetch proof that this binding is validly included in the epoch's STR. Each epoch's STR signature (64 bytes) must be downloaded and the client must fetch its new authentication path. However, the server can significantly compress the length of this path by only sending the hashes on the user's path which have changed since the last epoch. If n changes are made to the tree, a given authentication path will have $\lg_2(n)$ expected changed nodes. (This is the expected longest prefix match between the n changed indices and the terminating index of the given authentication path.) Therefore each epoch requires downloading an average of $64+\lg_2(n) \cdot 32 \approx 736$ bytes. Verification time will be similar to verifying another user's proof, dominated by the cost of signature verification. While clients need to fetch each STR from the server, they are only required to *store* the most recent STR (see §5.3).

To monitor a binding for a day, the client must download a total of about 19.1 kB. Note that we have assumed users update randomly throughout the day, but for a fixed number of updates this is actually the worst-case scenario for bandwidth consumption; bursty updates will actually lead to a lower amount of bandwidth as each epoch's proof is $\lg_2(n)$ for n changes. These numbers indicate that neither bandwidth nor computational overheads pose a significant burden for CONIKS clients.

Auditing cost. For a client or other auditor tracking all of a provider's STRs, assuming the policy field changes rarely, the only new data in an STR is the new timestamp, the new tree root and signature (the previous STR and epoch number can be inferred and need not be transmitted). The total size of each STR in minimal form is just 104 bytes (64 for the signature, 32 for the root and 8 for a timestamp), or 2.5 kB per day to audit a specific provider.

	#	#	#	approx. download size					
	VUFs	sigs.	hashes	RSA		EC		BLS	
lookup (per binding)	1	1	$\lg N+1$	1568	B	1216	B	1120	B
monitor (epoch)	0	1	$\lg n$	928	B	726	B	704	B
monitor (day)	1	$k^{\dagger}$	$k\lg n$	22.6	kB	17.6	kB	16.1	kB
audit (epoch, per STR)	0	1	1	288	B	96	B	64	B
audit (day, per STR)	0	$k^{\dagger}$	k	6.9	kB	2.3	kB	0.8	kB

Table 1: Client bandwidth requirements, based the number of signatures, VUFs and hashes downloaded for lookups, monitoring, and auditing. Sizes are given assuming a $N \approx 2^{32}$ total users, $n \approx 2^{21}$ changes per epoch, and $k \approx 24$ epochs per day. Signatures that can be aggregated into a single signature to transmit in the BLS signature scheme are denoted by †.

6 Discussion

6.1 Coercion of Identity Providers

Government agencies or other powerful adversaries may attempt to coerce identity providers into malicious behavior. Recent revelations about government surveillance and collection of user communications data world-wide have revealed that governments use mandatory legal process to demand access to information providers' data about users' private communications and Internet activity [9, 23, 24, 51, 52]. A government might demand that an identity provider equivocate about some or all name-to-key bindings. Since the identity provider is the entity actually mounting the attack, a user of CONIKS has no way of technologically differentiating between a malicious insider attack mounted by the provider itself and this coerced attack [18]. Nevertheless, because of the consistency and non-equivocation checks CONIKS provides, users could expose such attacks, and thereby mitigate their effect.

Furthermore, running a CONIKS server may provide some legal protection for service providers under U.S. law for providers attempting to fight legal orders, because complying with such a demand will produce public evidence that may harm the provider's reputation. (Legal experts disagree about whether and when this type of argument shelters a provider[45].)

6.2 Key Loss and Account Protection

CONIKS clients are responsible for managing their private keys. However, CONIKS can provide account protection for users who enforce the paranoid key change policy and have forfeit their username due to key loss. Even if Alice's key is lost, her identity remains secure; she can continue performing consistency checks on her old binding. Unfortunately, if a future attacker manages to obtain her private key, that attacker may be able to assume her "lost identity".

In practice, this could be prevented by allowing the provider to place a "tombstone" on a name with its own signature, regardless of the user's key policy. The provider would use some specific out-of-band authorization steps to authorize such an action. Unlike allowing providers to issue key change operations, though, a permanent account deactivation does not require much additional trust in the provider, because a malicious provider could already render an account unusable through denial of service.

6.3 Protocol Extensions

Limiting the effects of denied service. Sufficiently powerful identity providers may refuse to distribute STRs to providers with which they do not collude. In these cases, clients who query these honest providers will be unable to obtain explicit proof of equivocation. Fortunately, clients may help circumvent this by submitting observed STRs to these honest identity providers. The honest identity providers can verify the other identity provider's signature, and then store and redistribute the STR.

Similarly, any identity provider might ignore requests about individual bindings in order to prevent clients from performing consistency checks or key changes. In these cases, clients may be able to circumvent this attack by using other providers to proxy their requests, with the caveat that a malicious provider may ignore all requests for a name. This renders this binding unusable for as long as the provider denies service. However, this only allows the provider to deny service, any modification to the binding during this attack would become evident as soon as the service is restored.

Obfuscating the social graph. As an additional privacy requirement, users may want to conceal with whom they are in communication, or providers may want to offer anonymized communication. In principle, users could use Tor to anonymize their communications. However, if only few users in CONIKS use Tor, it is possible for providers to distinguish clients connecting through Tor from those connecting to the directly.

CONIKS could leverage the proxying mechanism described in §6.3 for obfuscating the social graph. If Alice would like to conceal with whom she communicates, she could require her client to use other providers to proxy any requests for her contacts' bindings or consistency proofs. Clients could choose these proxying providers uniformly at random to minimize the amount of information any single provider has about a particular user's contacts. This can be further improved the more providers agree to act as proxies. Thus, the only way for providers to gain information about whom a given user is contacting would be to aggregate collected requests. For system-wide Tor-like anonymization, CONIKS providers could form a mixnet [13], which would provide much higher privacy guarantees but would likely hamper the deployability of the system.

Randomizing the order of directory entries. Once a user learns the lookup index of a name, this position in the tree is known for the rest of time because the index is a deterministic value. If a user has an authentication path for two users *bob@foo.com* and *alice@foo.com* which share a common prefix in the tree, the Bob's authentication path will leak any changes to Alice's binding if his key has not changed, and vice-versa. *foo.com* can prevent this information leakage by randomizing the ordering of entries periodically by including additional data when computing their lookup indices. However, such randomized reordering of all directory entries would require a complete reconstruction of the tree. Thus, if done every epoch, the identity provider would be able to provide enhanced privacy guarantees at the expense of efficiency. The shorter the epochs, the greater the tradeoff between efficiency and privacy. An alternative would be to reorder all entries every n epochs to obtain better efficiency.

Key Expiration. To reduce the time frame during which a compromised key can be used by an attacker, users may want to enforce key expiration. This would entail including the epoch in which the public key is to expire as part of the directory entry, and clients would need to ensure that such keys are not expired when checking the consistency of bindings. Furthermore, CONIKS could allow users to choose whether to enforce key expiration on their binding, and provide multiple security options allowing users to set shorter or longer expiration periods. When the key expires, clients can automatically change the expired key and specify the new expiration date according to the user's policies.

Support for Multiple Devices. Any modern communication system must support users communicating from multiple devices. CONIKS easily allows users to bind multiple keys to their username. Unfortunately, device pairing has proved cumbersome and error-prone for users in practice [32, 67]. As a result, most widely-deployed chat applications allow users to simply install software to a new device which will automatically create a new key and add it to the directory via password authentication.

The tradeoffs for supporting multiple devices are the same as for key change. Following this easy enrollment procedure requires that Alice enforce the cautious key change policy, and her client will no longer be able to automatically determine if a newly observed key has been maliciously inserted by the server or represents the addition of a new device. Users can deal with this issue by requiring that any new device key is authenticated with a previously-registered key for a different device. This means that clients can automatically detect if new bindings are inconsistent, but will require users to execute a manual pairing procedure to sign the new keys as part of the paranoid key change policy discussed above.

7 Related Work

Certificate validation systems. Several proposals for validating SSL/TLS certificates seek to detect fraudulent certificates via transparency logs [4, 34, 38, 39, 53], or observatories from different points in the network [4, 34, 54, 58, 68]. Certificate Transparency (CT) [39] publicly logs all certificates as they are issued in a signed append-only log. This log is implemented as a chronologically-ordered Merkle binary search tree. Auditors check that each signed tree head represents an extension of the previous version of the log and gossip to ensure that the log server is not equivocating.

This design only maintains a set of issued certificates, so domain administrators must scan the entire list of issued certificates (or use a third-party monitor) in order to detect any newly-logged, suspicious certificates issued for their domain. We consider this a major limitation for user communication as independent, trustworthy monitors may not exist for small identity providers. CT is also not privacy-preserving; indeed it was designed with the opposite goal of making all certificates publicly visible.

Enhanced Certificate Transparency (ECT) [60], which was developed concurrently [46] extends the basic CT design to support efficient queries of the current set of valid certificates for a domain, enabling built-in revocation. Since ECT adds a second Merkle tree of currently valid certificates organized as a binary search tree sorted lexicographically by domain name, third-party auditors must verify that no certificate appears in only one of the trees by mirroring the entire structure and verifying all insertions and deletions.

Because of this additional consistency check, auditing in ECT requires effort linear in the total number of changes to the logs, unlike in CT or CONIKS, which only require auditors to verify a small number of signed tree roots. ECT also does not provide privacy: the proposal suggests storing users in the lexicographic tree by a hash of their name, but this provides only weak privacy as most usernames are predictable and their hash can easily be determined by a dictionary attack.

Other proposals include public certificate observatories such as Perspectives [54, 58, 68], and more complex designs such as Sovereign Keys [53] and AKI/ARPKI [4, 34] which combine append-only logs with policy specifications to require multiple parties to sign key changes and revocations to provide proactive as well as reactive security.

All of these systems are designed for TLS certificates, which differ from CONIKS in a few important ways. First, TLS has many certificate authorities sharing a single, global namespace. It is not required that the different CAs offer only certificates that are consistent or non-overlapping. Second, there is no notion of certificate or name privacy in the TLS setting,[11] and as a result, they use data structures making the entire name-space public. Finally, stronger assumptions, such as maintaining a private key forever or designating multiple parties to authorize key changes, might be feasible for web administrators but are not practical for end users.

Key pinning. An alternative to auditable certificate systems are schemes which limit the set of certificate authorities capable of signing for a given name, such as certificate pinning [16] or TACK [44]. These approaches are brittle, with the possibility of losing access to a domain if an overly strict pinning policy is set. Deployment of pinning has been limited due to this fear and most web administrators have set very loose policies [35]. This difficulty of managing keys, experienced even by technically savvy administrators, highlights how important it is to require no key management by end users.

Identity and key services. As end users are accustomed to interacting with a multitude of identities at various online services, recent proposals for online identity verification have focused on providing a secure means for consolidating these identities, including encryption keys.

Keybase [37] allows users to consolidate their online account information while also providing semi-automated consistency checking of name-to-key bindings by verifying control of third-party accounts. This system's primary function is to provide an easy means to consolidate online identity information in a publicly auditable log. It is not designed for automated key verification and it does not integrate seamlessly into existing applications.

Nicknym [56] is designed to be purely an end-user key verification service, which allows users to register existing third-party usernames with public keys. These bindings are publicly auditable by allowing clients to query any Nicknym provider for *individual* bindings they observe. While equivocation about bindings can be detected in this manner in principle, Nicknym does not maintain an authenticated history of published bindings which would provide more robust consistency checking as in CONIKS.

Cryptographically accountable authorities. Identity-based encryption inherently requires a trusted private-key generator (PKG). Goyal [28] proposed the accountable-authority model, in which the PKG and a user cooperate to generate the user's private key in such a way that the PKG does not know what private key the user has chosen. If the PKG ever runs this protocol with another party to generate a second private key, the existence of two private keys would be proof of misbehavior. This concept was later extended to the black-box accountable-authority model [29, 61], in which even issuing a black-box decoder algorithm is enough to prove misbehavior. These schemes have somewhat different security goals than CONIKS in that they require discovering two *private* keys to prove misbehavior (and provide no built-in mechanism for such discovery). By contrast, CONIKS is designed to provide a mechanism to discover if two distinct *public* keys have been issued for a single name.

VUFs and dictionary attacks. DNSSEC [15] provides a hierarchical mapping between domains and signing keys via an authenticated linked list. Because each domain references its immediate neighbors lexicographically in this design, it is possible for an adversary to enumerate the entire set of domains in a given zone via *zone walking* (repeatedly querying neighboring domains). In response, the NSEC3 extension [40] was added; while it prevents trivial enumeration, it suffers a similar vulnerability to ECT in that likely domain names can be found via a dictionary attack because records are sorted by the hash of their domain name. Concurrent with our work on CONIKS, [27] proposed NSEC5, effectively using a verifiable unpredictable function (also in the form of a deterministic RSA signature) to prevent zone enumeration.

8 Conclusion

We have presented CONIKS, a key verification system for end users that provides consistency and privacy for users' name-to-key bindings, all without requiring explicit key management by users. CONIKS allows clients to efficiently monitor their own bindings and quickly detect equivocation with high probability. CONIKS is highly scalable and is backward compatible with existing secure communication protocols. We have built a prototype

[11] Some organizations use "private CAs" which members manually install in their browsers. Certificate transparency specifically exempts these certificates and cannot detect if private CAs misbehave.

CONIKS system which is application-agnostic and supports millions of users on a single commodity server.

As of this writing, several major providers are implementing CONIKS-based key servers to bolster their end-to-end encrypted communications tools. While automatic, decentralized key management without least a semi-trusted key directory remains an open challenge, we believe CONIKS provides a reasonable baseline of security that any key directory should support to reduce user's exposure to mass surveillance.

Acknowledgments

We thank Gary Belvin, Yan Zhu, Arpit Gupta, Josh Kroll, David Gil, Ian Miers, Henry Corrigan-Gibbs, Trevor Perrin, and the anonymous USENIX reviewers for their feedback. This research was supported by NSF Award TC-1111734. Joseph Bonneau is supported by a Secure Usability Fellowship from OTF and Simply Secure.

References

[1] Pidgin. `http://pidgin.im`, Retr. Apr. 2014.

[2] Protocol Buffers. `https://code.google.com/p/protobuf`, Retr. Apr. 2014.

[3] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid. Special Publication 800-57 rev. 3. *NIST*, 2012.

[4] D. Basin, C. Cremers, T. H.-J. Kim, A. Perrig, R. Sasse, and P. Szalachowski. ARPKI: attack resilient public-key infrastructure. *ACM CCS*, 2014.

[5] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2), 2012.

[6] D. J. Bernstein, M. Hamburg, A. Krasnova, and T. Lange. Elligator: Elliptic-curve points indistinguishable from uniform random strings. *ACM CCS*, 2013.

[7] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the weil pairing. *ASIACRYPT*, 2001.

[8] N. Borisov, I. Goldberg, and E. Brewer. Off-the-record communication, or, why not to use PGP. *WPES*, 2004.

[9] S. Braun, A. Flaherty, J. Gillum, and M. Apuzzo. Secret to Prism program: Even bigger data seizure. Associated Press, Jun. 2013.

[10] P. Bright. Another fraudulent certificate raises the same old questions about certificate authorities. Ars Technica, Aug. 2011.

[11] P. Bright. Independent Iranian hacker claims responsibility for Comodo hack. Ars Technica, Mar. 2011.

[12] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer. RFC 2440 OpenPGP Message Format, Nov. 1998.

[13] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2), Feb. 1981.

[14] D. Chaum and T. P. Pedersen. Wallet databases with observers. *CRYPTO*, 1993.

[15] D. Eastlake. RFC 2535: Domain Name System Security Extensions. 1999.

[16] C. Evans, C. Palmer, and R. Sleevi. Internet-Draft: Public Key Pinning Extension for HTTP. 2012.

[17] P. Everton. Google's Gmail Hacked This Weekend? Tips To Beef Up Your Security. Huffington Post, Jul. 2013.

[18] E. Felten. A Court Order is an Insider Attack, Oct. 2013.

[19] T. Fox-Brewster. WhatsApp adds end-to-end encryption using TextSecure. The Guardian, Nov. 2014.

[20] M. Franklin and H. Zhang. Unique ring signatures: A practical construction. *Financial Cryptography*, 2013.

[21] P. Gallagher and C. Kerry. FIPS Pub 186-4: Digital signature standard, DSS. NIST, 2013.

[22] S. Gaw, E. W. Felten, and P. Fernandez-Kelly. Secrecy, flagging, and paranoia: Adoption criteria in encrypted email. *CHI*, 2006.

[23] B. Gellman. The FBI's Secret Scrutiny. The Wasington Post, Nov. 2005.

[24] B. Gellman and L. Poitras. U.S., British intelligence mining data from nine U.S. Internet companies in broad secret program. The Washington Post, Jun. 2013.

[25] S. Gibbs. Gmail does scan all emails, new Google terms clarify. The Guardian, Apr. 2014.

[26] I. Goldberg, K. Hanna, and N. Borisov. pidgin-otr. `http://sourceforge.net/p/otr/pidgin-otr/ci/master/tree/`, Retr. Apr. 2014.

[27] S. Goldberg, M. Naor, D. Papadopoulos, L. Reyzin, S. Vasant, and A. Ziv. NSEC5: Provably Preventing DNSSEC Zone Enumeration. *NDSS*, 2015.

[28] V. Goyal. Reducing trust in the pkg in identity based cryptosystems. *CRYPTO*, 2007.

[29] V. Goyal, S. Lu, A. Sahai, and B. Waters. Black-box accountable authority identity-based encryption. *ACM CCS*, 2008.

[30] T. Icart. How to hash into elliptic curves. *CRYPTO*, 2009.

[31] J. Jonsson and B. Kaliski. RFC 3447 Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1, Feb. 2003.

[32] R. Kainda, I. Flechais, and A. W. Roscoe. Usability and Security of Out-of-band Channels in Secure Device Pairing Protocols. *SOUPS*, 2009.

[33] J. Katz. Analysis of a proposed hash-based signature standard. `https://www.cs.umd.edu/~jkatz/papers/HashBasedSigs.pdf`, 2014.

[34] T. H.-J. Kim, L.-S. Huang, A. Perrig, C. Jackson, and V. Gligor. Accountable key infrastructure (AKI): a proposal for a public-key validation infrastructure. *WWW*, 2013.

[35] M. Kranch and J. Bonneau. Upgrading HTTPS in midair: HSTS and key pinning in practice. *NDSS*, 2015.

[36] D. W. Kravitz. Digital signature algorithm, 1993. US Patent 5,231,668.

[37] M. Krohn and C. Coyne. Keybase. `https://keybase.io`, Retr. Feb. 2014.

[38] B. Laurie and E. Kasper. Revocation Transparency. `http://sump2.links.org/files/RevocationTransparency.pdf`, Retr. Feb. 2014.

[39] B. Laurie, A. Langley, E. Kasper, and G. Inc. RFC 6962 Certificate Transparency, Jun. 2013.

[40] B. Laurie, G. Sisson, R. Arends, and D. Black. RFC 5155: DNS Security (DNSSEC) Hashed Authenticated Denial of Existence. 2008.

[41] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). *OSDI*, 2004.

[42] G. Lindberg. RFC 2505 Anti-Spam Recommendations for SMTP MTAs, Feb. 1999.

[43] M. Madden. Public Perceptions of Privacy and Security in the Post-Snowden Era. *Pew Research Internet Project*, Nov. 2014.

[44] M. Marlinspike and T. Perrin. Internet-Draft: Trust Assertions for Certificate Keys. 2012.

[45] J. Mayer. Surveillance law. Available at `https://class.coursera.org/surveillance-001`.

[46] M. S. Melara. CONIKS: Preserving Secure Communication with Untrusted Identity Providers. Master's thesis, Princeton University, Jun 2014.

[47] S. Micali, M. Rabin, and S. Vadhan. Verifiable random functions. *FOCS*, 1999.

[48] N. Perloth. Yahoo Breach Extends Beyond Yahoo to Gmail, Hotmail, AOL Users. New York Times Bits Blog, Jul. 2012.

[49] T. Pornin. RFC 6979: Deterministic usage of the digital signature algorithm (DSA) and elliptic curve digital signature algorithm (ECDSA). 2013.

[50] Electronic Frontier Foundation. Secure Messaging Scorecard. `https://www.eff.org/secure-messaging-scorecard`, Retr. 2014.

[51] Electronic Frontier Foundation. National Security Letters - EFF Surveillance Self-Defense Project. `https://ssd.eff.org/foreign/nsl`, Retr. Aug. 2013.

[52] Electronic Frontier Foundation. National Security Letters. `https://www.eff.org/issues/national-security-letters`, Retr. Nov. 2013.

[53] Electronic Frontier Foundation. Sovereign Keys. `https://www.eff.org/sovereign-keys`, Retr. Nov. 2013.

[54] Electronic Frontier Foundation. SSL Observatory. `https://www.eff.org/observatory`, Retr. Nov. 2013.

[55] Internet Mail Consortium. S/MIME and OpenPGP. `http://www.imc.org/smime-pgpmime.html`, Retr. Aug. 2013.

[56] LEAP Encryption Access Project. Nicknym. `https://leap.se/en/docs/design/nicknym`, Retr. Feb. 2015.

[57] Reuters. At Sina Weibo's Censorship Hub, 'Little Brothers' Cleanse Online Chatter, Nov. 2013.

[58] Thoughtcrime Labs Production. Convergence. `http://convergence.io`, Retr. Aug. 2013.

[59] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[60] M. D. Ryan. Enhanced certificate transparency and end-to-end encrypted email. *NDSS*, Feb. 2014.

[61] A. Sahai and H. Seyalioglu. Fully secure accountable-authority identity-based encryption. In *Public Key Cryptography–PKC 2011*, pages 296–316. Springer, 2011.

[62] B. Schneier. Apple's iMessage Encryption Seems to Be Pretty Good. `https://www.schneier.com/blog/archives/2013/04/apples_imessage.html`, Retr. Feb. 2015.

[63] C.-P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3), 1991.

[64] C. Soghoian and S. Stamm. Certified Lies: Detecting and Defeating Government Interception Attacks against SSL. *Financial Crypto'*, 2012.

[65] R. Stedman, K. Yoshida, and I. Goldberg. A User Study of Off-the-Record Messaging. *SOUPS*, Jul. 2008.

[66] N. Unger, S. Dechand, J. Bonneau, S. Fahl, H. Perl, I. Goldberg, and M. Smith. SoK: Secure Messaging. *IEEE Symposium on Security and Privacy*, 2015.

[67] B. Warner. Pairing Problems, 2014.

[68] D. Wendlandt, D. G. Andersen, and A. Perrig. Perspectives: improving SSH-style host authentication with multi-path probing. In *Usenix ATC*, Jun. 2008.

[69] A. Whitten and J. D. Tygar. Why Johnny can't encrypt: a usability evaluation of PGP 5.0. *USENIX Security*, 1999.

[70] P. R. Zimmermann. *The official PGP user's guide*. MIT Press, Cambridge, MA, USA, 1995.

A Discrete-log Based VRF Construction

We propose a simple discrete-log based VRF in the random oracle model. By definition, this scheme is also a VUF as required. This construction was described by Franklin and Zhang [20] although they considered it already well-known. Following Micali et al.'s outline [47], the basic idea is to publish a commitment c to the seed k of a pseudo-random function, compute $y = f_k(x)$ as the VUF, and issue non-interactive zero-knowledge proofs that $y = f_k(x)$ for some k to which c is a commitment of. The public key and private key are c and k.

Parameters. For a group[12] $\mathscr{G}$ with generator g of prime order q, the prover chooses a random $k \overset{R}{\leftarrow} (1,q)$ as their private key and publishes $G = g^k$ as their public key. We require two hash functions: one which maps to curve points [6, 30] $\mathbf{H}_1 : * \to \mathscr{G}$ and one which maps to integers $\mathbf{H}_2 : * \to (1,q)$ which are modeled as random oracles.

VRF computation. The VRF is defined as:

$$\mathrm{VRF}_k(m) = \mathbf{H}_1(m)^k$$

Non-interactive proof The prover must show in zero-knowledge that there is some x for which $G = g^k$ and $H = h^k$ for $h = \mathbf{H}_1(m)$. The proof is a standard Sigma proof of equality for two discrete logarithms made non-interactive using the Fiat-Shamir heuristic [14]. The prover chooses $r \overset{R}{\leftarrow} (1,q)$ and transmits $s = \mathbf{H}_1(m, g^r, h^r)$ and $t = r - sk \mod q$.

To verify that $\mathrm{VRF}_k(m) = \mathbf{H}_1(m)^k$ is a correct VRF computation given a proof (s,t), the verifier checks that

$$s = \mathbf{H}_1\left(m, g^t \cdot G^s, H(m)^t \cdot \mathrm{VRF}_k(m)^s\right)$$

We refer the reader to [14, 20] for proof that this scheme satisfies the properties of a VRF. Note that the pseudorandomness

[12] Note that we use multiplicative group notation here, though this scheme applies equally to elliptic-curve groups.

reduces to the Decisional Diffie-Hellman assumption. The tuple $(\mathbf{H}_1(m), G = g^k, \mathrm{VRF}_k(m) = \mathbf{H}_1(m)^k)$ is a DDH triple, therefore an attacker that could distinguish $\mathrm{VRF}_k(m)$ from random could break the DDH assumption for $\mathscr{G}$.

Efficiency. Proofs consist of a group elements (the VRF result $\mathbf{H}_1(m)^k$) and two integer which is the size of the order of the group $((s,t))$. For 256-bit elliptic curve, this leads to proofs of size 768 bits (96 bytes).

B Analysis of Equivocation Detection

CONIKS participants check for non-equivocation by consulting auditors to ensure that they both see an identical STR for a given provider P. Clients perform this cross-verification by choosing uniformly at random a small set of auditors from the set of known auditors, querying them for the observed STRs from P, and comparing these observed STRs to the signed tree root presented directly to the client by P. If any of the observed STRs differ from the STR presented to the client, the client is sure to have detected an equivocation attack.

B.1 Single Equivocating Provider

Suppose that *foo.com* wants to allow impersonation of a user Alice to hijack all encrypted messages that a user Bob sends her. To mount this attack, *foo.com* equivocates by showing Alice STR A, which is consistent with Alice's *valid* name-to-key binding, and showing Bob STR B, which is consistent with a fraudulent binding for Alice.

If Bob is the only participant in the system to whom *foo.com* presents STR B, while all other users and auditors receive STR A, Alice will not detect the equivocation (unless she compares her STR directly with Bob's). Bob, on the other hand, will detect the equivocation immediately because performing the non-equivocation check with a single randomly chosen auditor is sufficient for him to discover a diverging STR for *foo.com*.

A more effective approach for *foo.com* is to choose a subset of auditors who will be presented STR A, and to present the remaining auditors with STR B. Suppose the first subset contains a fraction f of all auditors, and the second subset contains the fraction $1-f$. If Alice and Bob each contact k randomly chosen providers to check consistency of *foo.com*'s STR, the probability that Alice fails to discover an inconsistency is f^k, and the probability that Bob fails to discover an inconsistency is $(1-f)^k$. The probability that both will fail is $(f-f^2)^k$, which is maximized with $f=\frac{1}{2}$. Alice and Bob therefore *fail* to discover equivocation with probability

$$\varepsilon \leq \left(\frac{1}{4}\right)^k$$

In order to discover the equivocation with probability $1-\varepsilon$, Alice and Bob must perform $-\frac{1}{2}\log\frac{\varepsilon}{2}$ checks. After performing 5 checks each, Alice and Bob would have discovered an equivocation with 99.9% probability.

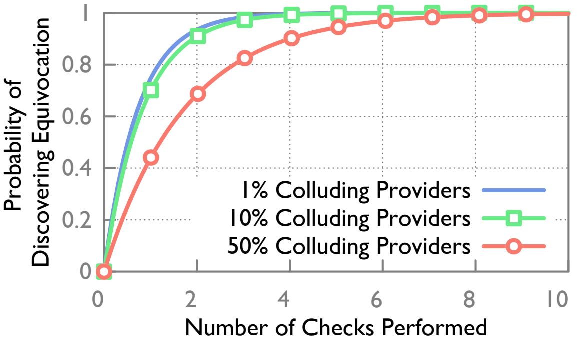


Figure 8: This graph shows the probability that Alice and Bob will detect an equivocation after each performing k checks with randomly chosen auditors.

B.2 Colluding Auditors

Now suppose that *foo.com* colludes with auditors in an attempt to better hide its equivocation about Alice's binding. The colluding auditors agree to tell Alice that *foo.com* is distributing STR A while telling Bob that *foo.com* is distributing STR B. As the size of the collusion increases, Alice and Bob become less likely to detect the equivocation. However, as the number of auditors in the system (and therefore, the number of auditors not participating in the collusion) increases, the difficulty of detecting the attack decreases.

More precisely, we assume that *foo.com* is colluding with a proportion p of all auditors. The colluding auditors behave as described above, and *foo.com* presents STR A to a fraction f of the non-colluding providers. Alice and Bob each contacts k randomly chosen providers. The probability of Alice failing to detect equivocation within k checks is therefore $(p+(1-p)f)^k$ and the probability of Bob failing to detect equivocation within k checks is $(p+(1-p)(1-f))^k$. The probability that neither Alice nor Bob detects equivocation is then

$$\varepsilon = ((p+(1-p)f)(p+(1-p)(1-f)))^k$$

As before, this is maximized when $f=\frac{1}{2}$, so the probability that Alice and Bob *fail* to detect the equivocation is

$$\varepsilon \leq \left(\frac{1+p}{2}\right)^{2k}$$

If $p = 0.1$, then by doing 5 checks each, Alice and Bob will discover equivocation with 99.7% probability.

Figure 8 plots the probability of discovery as p and k vary. If fewer than 50% of auditors are colluding, Alice and Bob will detect an equivocation within 5 checks with over 94% probability. In practice, large-scale collusion is unexpected, as today's secure messaging services have many providers operating with different business models and under many different legal and regulatory regimes. In any case, if Alice and Bob can agree on a single auditor whom they both trust to be honest, then they can detect equivocation with certainty if they both check with that trusted auditor.

Investigating the Computer Security Practices and Needs of Journalists

Susan E. McGregor
Tow Center for Digital Journalism
Columbia Journalism School

Polina Charters, Tobin Holliday
Master of HCI + Design, DUB Group
University of Washington

Franziska Roesner
Computer Science & Engineering
University of Washington

Abstract

Though journalists are often cited as potential users of computer security technologies, their practices and mental models have not been deeply studied by the academic computer security community. Such an understanding, however, is critical to developing technical solutions that can address the real needs of journalists and integrate into their existing practices. We seek to provide that insight in this paper, by investigating the general and computer security practices of 15 journalists in the U.S. and France via in-depth, semi-structured interviews. Among our findings is evidence that existing security tools fail not only due to usability issues but when they actively interfere with other aspects of the journalistic process; that communication methods are typically driven by sources rather than journalists; and that journalists' organizations play an important role in influencing journalists' behaviors. Based on these and other findings, we make recommendations to the computer security community for improvements to existing tools and future lines of research.

1 Introduction

In recent decades, improved digital communication technologies have reduced barriers to journalism worldwide. Security weaknesses in these same technologies, however, have put journalists and their sources increasingly at risk of identification, prosecution, and persecution by powerful entities, threatening efforts in investigative reporting, transparency, and whistleblowing.

Recent examples of such threats include intensifying U.S. leak prosecutions (e.g. [46, 54]), the secret seizure of journalists' phone records by the U.S. Justice Department [55], the collection of journalists' emails by the British intelligence agency GCHQ [11], politically-motivated malware targeting journalists (among others) [13, 36, 41, 45], and other types of pervasive digital surveillance [34]. In the U.S., these developments have led to a documented "chilling effect", leading sources to reduce communication with journalists even on non-sensitive issues [25, 40]. Elsewhere, risks to journalists and sources cross the line from legal consequences to the potential for physical harm [42, 57, 58].

Responses to these escalating threats have included guides to best computer security practices for journalists (e.g., [17, 43, 47, 62]), which recommend the use of tools like PGP [67], Tor [22], and OTR [14]. More generally, the computer security community has developed many secure or anonymous communication tools (e.g., [4, 10, 14, 21–23, 63, 67]). These tools have seen relatively little adoption within the journalism community, however, even among the investigative journalists that should arguably be their earliest adopters [48].

To design and build tools that will successfully protect journalist-source communications, it is critical that the technical computer security community understand the practices, constraints, and needs of journalists, as well as the successes and failures of existing tools. However, the journalistic process has not been deeply studied by the academic computer security community. We seek to fill that gap in this paper, which is the result of a collaboration between researchers in the journalism and computer security communities, and which is targeted at a technical computer security audience.

To achieve this, we develop a grounded understanding of the journalistic process from a computer security perspective via in-depth, semi-structured interviews. Following accepted frameworks for qualitative research [18, 30, 35], we focus closely on a small number of participants. We interviewed 15 journalists employed in a range of well-respected journalistic institutions in the United States and France, analyzing these interviews using a grounded theory approach [18, 30]. We then synthesize these findings to shed light on the general practices (Section 4.3), security concerns (Section 4.4), defensive strategies (Section 4.5), and needs (Section 4.6) of journalists in their communications with sources.

Our interviews offer a glimpse into journalistic processes that deal with information and sources of a range of sensitivities. Some of our participants report being the direct targets of threats like eavesdropping and data theft: for example, one participant received threatening letters and had his laptop (and nothing else) stolen from his home while working on sensitive government-related stories. Others discuss their perceived or hypothetical security concerns, which we systematize in Section 4.4 — along with threats that participants tended to overlook, such as the trustworthiness of third-party services.

By cataloguing the computer security tools that our participants do and don't use (Section 4.5), we reveal new reasons for their successes or failures. For example, built-in disk encryption is widely used among our participants because it is both easy-to-use and does not require explicit installation. However, we find that many security tools are not used regularly by our participants. Beyond the expected usability issues, we find that the most critical failures arise when security tools interfere with another part of a journalist's process. For example, anonymous communication tools fail when they compromise a journalist's ability to verify the authenticity of a source or information. As one participant put it: *"If I don't know who they are and can't check their background, I'm not going to use the information they give."* This requirement limits the effectiveness even of tools developed specifically for journalists — such as SecureDrop [26], which supports anonymous document drops — and highlights how crucial it is for computer security experts who design tools for journalists to understand and respect the requirements of the journalistic process.

Based on our findings, we make recommendations for technical computer security researchers focusing on journalist-source communications, including:

- *Focus on sources:* Journalists often choose communication methods based on sources' comfort with and access to technology, rather than the sensitivity of information — particularly when sources are on the other side of a "digital divide" (e.g., low-income populations with limited access to technology).
- *Consider journalistic requirements:* Security tools that impede essential aspects of the journalistic process (e.g., source authentication) will struggle to see widespread adoption. Meanwhile, unfulfilled technical needs (e.g., the absence of a standard knowledge management tool for notes) may cause journalists to introduce vulnerabilities into their process (e.g., reliance on third-party cloud tools not supported by their organization). These unfulfilled needs, however, present opportunities to integrate computer security seamlessly into new tools with broader applicability to the field of journalism.
- *Beyond journalist-source communications:* A journalist's organization and colleagues play an important role in the security of his or her practices; security tools must consider this broader ecosystem.

We consider these and other lessons and recommendations in more detail below. Taken together, our findings suggest that further collaboration between the computer security and journalism communities is critical, with our work as an important first step in informing and grounding future research in computer security around journalist-source communications.

2 Related Work

We provide context for our study through a survey of three types of related works: studies of journalists and computer security, computer security guidelines developed specifically for journalists, and secure communication tools.

Studies of journalists and computer security. Several recent studies interviewed or surveyed journalists (among others) in Mexico [58], Pakistan [42], Tibet [15] and Vietnam [57] to shed light on the risks associated with their work, as well as their use and understanding of computer security technologies (such as encryption). Despite the different context, our findings echo some of the findings in these studies: for example, that maintaining communication with sources may take precedence over security [57], that meeting in person may be preferable to digital communication [15], and that the use of more sophisticated computer security tools is typically limited even in the face of real threats, including risk of physical harm [42, 57, 58]. These prior studies primarily recommended increased computer security education and training for journalists; though we concur, our work focuses more on technical recommendations.

Though most journalists in countries like the United States do not face physical harm, recent interviews of U.S. journalists and lawyers [40] revealed a distinct chilling effect in these fields resulting from revelations about widespread government surveillance. For example, journalists reported increased reluctance by sources to discuss even non-sensitive topics. Another recent report [48] provides quantitative survey data about the use of computer security tools by investigative journalists, suggesting (as we also find) that sophisticated computer security tools have seen limited adoption. These studies begin to paint a picture of the computer security mental models and needs of journalists; we expand on that understanding in this work and distill from it concrete technical and research recommendations.

The computer security community has previously studied the usability and social challenges with encryption among other populations (e.g., [27, 65]). Where applicable, we draw comparisons or highlight differences to the findings of these works.

Computer security guidelines for journalists. Recent concerns about government surveillance have prompted journalists in the U.S. and elsewhere to weigh computer security more seriously. For example, several groups have developed computer security guidelines and best practices for journalists [17, 43, 47, 62]. Online guides for journalists and other technology users (e.g., [16]) also abound. These efforts highlight the need for engagement between the journalism and computer security communities, but generally take the approach of educating journalists to use existing available tools, such as GPG and Tor. The goal of our work is to provide the developers of new technologies with a deep, grounded understanding of the needs and security concerns of journalists.

Secure communication. A large body of work exists on secure communication and data storage, both commercially and in the computer security research literature. For example, various smartphone applications aim to provide secure text messaging or calling [6, 60, 64]; a range of desktop applications provide disk encryption and cleaning [1, 5, 8]; Tor [22, 61] aims to provide anonymous web surfing; Tails [4] aims to provide a private and anonymous operating system; and tools like GPG and CryptoCat provide encryption for email and chat messages respectively [2, 31]. Several email providers have also attempted to provide secure and anonymous email [3, 44]. Though valuable, most of these tools and techniques have known weaknesses: anonymous email, for example, lacks essential legal protections [38, 51]. Tor and Tails do not protect against all threats and present usability challenges (e.g., [49]). Finally, many applications that appear to provide certain security properties fail to provide those guarantees in the face of government requests [33, 56].

While the above-mentioned commercial tools are among those frequently recommended to journalists, the computer security research community has also considered anonymous communications in depth. These efforts include developing, analyzing, and attacking systems like trusted relays, mix systems, and onion routing such as that used in Tor. Good summaries of these bodies of work can be found in [21] and [23]. Secure messaging in general is summarized in [63]. There have also been a number of efforts toward creating self-destructing data, including early work by Perlman [52] and more recent work on Vanish [28, 29]. An analysis of different approaches for secure data deletion appears in [53]. There have also been significant efforts toward ephemeral and secure two-way communications, such as the off-the-record (OTR) messaging system [14, 32].

Though the above-mentioned technologies are valuable, our research suggests that many of them require steps or actions at odds with substantive aspects of the journalistic process or technical access issues of journalists and/or their sources. Moreover, these access issues are often most acute among the most vulnerable source populations with whom journalists work (e.g., sources involved in the criminal justice system).

Though some journalism-specific tools have been developed and deployed, notably SecureDrop [20, 26] and similar systems, our findings suggests that such anonymous document drops — while more secure — comprise only a small portion of journalists' source material. In a similar vein, Witness [7] and the Syria Accountability Project [59] focus on collecting and securely storing sensitive eyewitness data, but are not necessarily designed to protect the kind of ongoing communications that our research and other sources [37, 39] suggest commonly drives sensitive reporting.

3 Methodology

To make possible a sufficiently rigorous qualitative, grounded theory based [18, 30] analysis of the general and computer security needs and practices of journalists, we followed the recommendation of Guest et al. [35] to conduct 12-20 interviews, until new themes stopped emerging [18]. Our in-depth, semi-structured interviews were conducted with 15 journalists. Table 1 summarizes our participants and interviews.

Human subjects and ethics. Our study was approved by the human subjects review boards (IRBs) of our institutions before any research activities began. We obtained informed written or verbal consent from all participants, both to participate in the study as well as to have the interviews audio recorded. We transmitted and stored these audio files only in encrypted form. We did not record or store any explicitly identifying metadata (e.g., the name of a journalist or organization), nor do we report those here. Though we asked participants to reflect on recent source communications, including those that touched on sensitive information, we explicitly asked them *not* to reveal identifying information about specific sources or stories. As journalists are normally responsible for protecting source identities, these constraints were not out of the ordinary; indeed, we felt that the resulting interviews did not contain unnecessarily sensitive details.

3.1 Recruitment

We recruited our participants via our existing connections to journalistic institutions, usually via verbal or email contact with a staff member followed by an email containing our recruitment blurb. For better anonymity, participants at each organization were not recruited directly but were selected by our contact person according to individuals' availability at the time of the interviews. In communicating with the main organizational contacts, we stressed a desire for balance in terms of participants' technical skill and the sensitivity of their work. The vast majority of interviews were conducted in-person, though a few were conducted via Skype.

For the purposes of this study, we limited our search for participants to journalists directly employed by well-respected journalistic institutions rather than freelance journalists. This focus allows us to explore the role of a journalist's employer in his or her computer security practices (or lack thereof). Our interviewees came from six different news organizations. Of these, four represent newsrooms and journalists who deal regularly with international (including non-Western) sources and stories of national and/or international profile. So while the organizations themselves are based in the U.S. and/or France, their work involves sources outside of those countries as well. The remaining organizations have a primarily U.S.-focused source base.

Nine interviews were conducted in France with journalists from French and U.S. journalistic institutions. Two of these interviews were conducted in French and were translated to English by another researcher. Both the interviewer and the translator are proficient in French. Due to our qualitative interview method and corresponding small sample size, we do not attempt to draw conclusions about differences between French and U.S. journalists in this work.

We do note that our participants are not necessarily representative of all journalists. It may be that journalists who agreed to speak with us are more (or less) security-conscious than those who declined, or that that the experiences of U.S. and French journalists differ from those of journalists in other countries. We also expect that the practices of freelance journalists differ from those of institutional journalists. Future work should study these questions; nevertheless, our interviews give us a valuable glimpse into the computer security practices and needs of a significant subset of the journalistic community.

3.2 Interview Procedure

One of the researchers conducted all of the interviews in the period from November 2014 through February 2015. Interviews were audio recorded and later transcribed and coded (more details below) by the remaining (non-interviewing) researchers. Each interview took between 15-45 minutes and had two parts:

Part 1: Questions about a specific story
We first prompted participants to tell us about the practices and tools that they use as journalists by asking them to think about a specific recent example. We asked:

> *Please think about a specific story that you have published in approximately the last year for which you spoke with a source. (There is no need to tell us the specific story or source, unless you believe this information is not sensitive and would like to share it.)*

In this context, we then asked about:

- Whether they had a relationship with the source prior to this story;
- How they first contacted the source about the story;
- Primary form(s) of communication with the source;
- Whether they would feel comfortable asking this source to use a specific communication method; and
- How representative this example is of their communication with sources in general.

Part 2: General questions
We then asked participants more general questions about their work as a journalist, including questions about:

- Their note-taking and storage process, and whether they take any steps to protect or share their notes;
- Problems that might arise if their digital notes or communications were revealed;
- Any non-technological strategies they use to protect themselves or their sources;
- Whether someone has ever recommended they use security-related technology in their work;
- How they define "sensitive" information or sources in their work;
- Any specific security-related problems to which they wish they had a solution;
- What kinds of devices they use, and who owns and/or administers them;
- Whether they have anyone, inside or outside of their organization, to whom they can go for help with computer security or other technologies; and
- Their self-described comfort level with technology and security-related technology.

Finally, we gave participants an opportunity to share any additional thoughts with us and to ask us any questions.

Throughout the interviews, we allowed participants to elaborate and ask clarification questions, and we asked follow-up questions where appropriate. As a result, the interviews did not necessarily proceed in the same order nor did they address identical questions.

3.3 Coding

To analyze the interviews, we used a grounded theory [18, 30] approach in which we developed a set of themes, or "codes", via an iterative process. After the interviewing researcher had conducted nearly half of the interviews, three additional researchers each independently listened to and transcribed several interviews. These researchers then met in person to develop, test, and iteratively modify an initial set of codes. Two researchers then independently coded each interview. As additional interviews were performed, the researchers reexamined and modified the codebook as necessary, going back and

		Participant		Interview			Technical Expertise	
Number	*Identifier*	*Gender*	*Organization (Type)*	*Location*	*Language*	*Length*	*General*	*Security*
1	P0	Male	Large, established	France	English	32 min	High	High
2	P1	Female	Large, new	USA	English	31 min	High	Medium
3	P2	Female	Large, established	France	English	39 min	Medium	Low
4	P3	Female	Large, established	France	English	39 min	High	Medium
5	P4	Female	Large, established	France	English	42 min	Medium	Low
6	P5	Male	Large, established	France	French	24 min	Medium	Low
7	P6	Male	Large, established	France	French	23 min	Medium	Medium
8	P7	Female	Large, established	France	English	27 min	High	Low
9	P8	Male	Large, established	France	English	20 min	High	Medium
10	P9	Male	Large, new	USA	English	41 min	High	Medium
11	P10	Female	Large, new	USA	English	31 min	Medium	Medium
12	P11	Female	Large, new	USA	English	19 min	Medium	Low
13	P12	Female	Small, new	USA	English	17 min	Medium	Low
14	P13	Female	Small, new	USA	English	34 min	High	Low
15	P14	Female	Small, established	USA	English	25 min	Medium	Medium

Table 1: **Interviews.** One researcher conducted all interviews between November 2014 and February 2015, at six well-respected journalistic institutions. The two interviews conducted in French were translated to English by another researcher (both researchers are proficient in French). On the right, we report participants' general and security-specific technical expertise; these values are self-reported. Organization size descriptors are based on those used by the Online News Association (`http://journalists.org/awards/online-journalism-awards-rules-eligibility/`). "New" organizations have existed for 10 years or less.

recoding previously coded interviews. This iterative process was repeated until the final codebook was created and all interviews were coded. The researchers then met in person to reach consensus where possible. We report inter-coder agreement inline with our results.

4 Results

We now turn to a discussion of results from our interviews. In designing and analyzing our interviews, we focused on several primary research questions, around which we organize this section:

1. What are the *general practices* of journalists in communicating with their sources?
2. What are the *security concerns* and threat models of journalists with respect to source communication?
3. What, if any, *defensive strategies* (technical or otherwise) do journalists employ to protect themselves or their sources? How and why do some possible defensive strategies succeed and others fail?
4. What are the *needs of journalists* in their communications with sources that are currently hampered or unfulfilled by computer security technologies?

By applying an appropriate qualitative analysis [18, 30, 35], we identify important themes and other observations present in the interviews. Where applicable, we report the raw number of participants who discussed a certain theme in order to give a rough indication of its prevalence amongst journalists. Our results are not quantitative, however: a given participant failing to mention a particular theme does not necessarily mean that it is inapplicable to him or her.

Each interview was coded independently by two researchers: a primary coder who coded all interviews, and two additional coders who coded non-overlapping sets of 9 and 6 interviews respectively. We report raw numbers based on the primary coder, with Cohen's kappa (κ) as a measure of inter-coder agreement [19] (averaging kappas for the two sets of coders). The average kappa for all results in the paper is 0.88. Fleiss rates any value of kappa over 0.75 as excellent agreement and between 0.40 and 0.75 as intermediate to good agreement [24].

4.1 Participants

Our participants are journalists working at major journalistic institutions in both the United States and France. Table 1 summarizes our 15 interviews and participants.

As reflected in Table 1, we spoke with journalists across the spectrum of general technical and computer security expertise. Some of our participants comfortably discussed their use of security tools such as encrypted chat and email, while others did not use or mention any security technologies at all. Regardless of technical and computer security expertise, our participants work with sources and stories of varying sensitivity. Stories considered "sensitive" by our participants include those involving information provided off-the-record by government officials, leaked or stolen documents, vulnerable populations (e.g., abuse victims or homeless people), and personal information that sources did not want published.

4.2 Key Findings

Before diving into our detailed results, we briefly highlight our key findings.

First, we find that journalist-source communications are often *driven by the source*. Participants tended to select communication mechanisms based on the comfort level, capacities, and preferences of sources, deferring to them to specify the use of computer security tools rather than imposing these on sources. In this sense, the existing communication habits of sources are a primary obstacle to adoption of secure communication tools among journalists. In particular, the *digital divide*, in which source populations do not have access to or knowledge about technology, presents a serious challenge.

Additionally, our study reveals both *expected security concerns* (e.g., government surveillance, disciplinary consequences for sources) and *less expected security concerns* (e.g., financial impact on organizations) held by our participants. Participants described many *ad hoc defensive strategies* to address these concerns, including ways to authenticate sources, to obfuscate information in filenames and notes, and to obfuscate communications metadata by contacting sources through intermediaries.

Finally, beyond the expected *usability and adoption challenges* of computer security technologies, we find that a major barrier to adoption of these tools arises when they *interfere with a journalist's other professional needs*. For example, participants described the challenge of authenticating anonymous sources, and more generally, the need to reduce communication barriers with sources. Our study also reveals the need among journalists for a more general knowledge management platform, for which today's journalists use ad hoc methods based on tools like Google Docs and Evernote. This need may represent an opportunity to seamlessly integrate stronger computer security properties into journalistic practices.

4.3 General Practices

We begin by overviewing the general journalistic process described by our participants, in order to provide important context for the computer security community when it designs tools for journalists. We highlight security implications where applicable, and dive into these more deeply in later subsections.

Finding sources. Many participants discussed having long-term sources (10 of 15), particularly for sensitive information (e.g., sources in government). A different subset described finding new sources relevant to new stories (10 of 15), often by following referrals from previously known contacts. The importance of long-term sources poses security challenges: for example, it may be hard to protect metadata about communications over a long period, especially if the journalist's communication with that source is not always sensitive (and thus not always conducted over secure channels).

Communicating with sources. Our participants typically communicate with sources by email, phone, SMS, and/or in person. Security tools, such as encrypted messaging, were used only in exceptional cases where the context was known in advance to be sensitive, and both the journalist and source were sufficiently tech-savvy.

The choice of communication technology is typically determined by what is most convenient for the source, including the platform on which source is most likely to respond. Several participants discussed the importance of reducing communication barriers to sources. In the words of P13, *"taking down barriers is the most important thing to source communication."* Thus, if the source is concerned about security and sufficiently tech-savvy, the journalist may use security technologies to communicate; however, several of our participants expressed hesitation about interfering with a source's decision about what form of communication — even if insecure — is acceptable. For example, P9 said:

> *[The source] probably understand[s] the threat model they're under better than I would. So, it brings up an interesting question: do you go with what they're comfortable with? Or do you say, alright, actually let me assess what's going on and get back to you with what would be appropriate. [...] People's first impression is that they would go by what the source feels comfortable doing. As opposed to stepping in and being paternalistic about it.*

This finding suggests that the computer security community must consider sources as well as journalists when developing secure communication tools for journalism.

Building trust with sources. In order to feel comfortable providing sensitive information, a source must trust the journalist. While some trust with sources is built naturally over time, several participants mentioned explicit strategies for building trust with sources, including: speaking with people informally before they become official sources, being explicit with sources about what is "on the record," respecting sources' later requests not to include something in a story, and using security technologies to protect communications.

Communication tools. Table 2 summarizes the non-security-specific technologies participants mentioned using in their work. Primary communication tools include phone, SMS, and email, with limited use of social media to contact sources (usually as a last resort). In addition to digital communication, in-person meetings with sources are common. While some participants reported meeting in person for security reasons, most cited this as a means to gain higher quality information from sources.

Among storage technologies, we note that Google Docs/Drive is particularly popular, and that many of the tools mentioned involve syncing local data to cloud storage. Though cloud storage may have security implications (e.g., exposing sensitive data to third parties), few participants voiced these concerns explicitly.

Tool or technology	Number of participants (of 15)	Inter-coder agreement (κ)
Phone	15	1.00
Email (unencrypted)	15	1.00
Google Docs/Drive	8	1.00
Microsoft Word	8	1.00
SMS	8	1.00
Social media	7	0.83
Dropbox	4	1.00
Skype	4	1.00
Evernote	3	1.00
Text editor	2	1.00
Chat (unencrypted)	1	1.00
Scrivener	1	1.00

Table 2: **Non-security-specific tools.** This table reports the number of participants who mentioned using various non-security-specific tools or technologies in their work.

Devices and accounts. Though participants typically reported relatively strong "data hygiene" practices for email — i.e., conducting work-related communications only from a work email account — everyone we spoke to used at least one personal device or account for communicating with sources, including personal laptops and (more commonly) personal cell phones. Many participants reported using iPhones or iPads, often to take photos of documents or audio-record interviews. These devices are not necessarily encrypted, and the resulting files may be automatically backed up to cloud storage. Personal/professional distinctions were often blurred for social media accounts, and participants frequently reported using personal Google Drive, Dropbox, or Evernote accounts to sync, store and share data, particularly when the organization did not have its own enterprise Google Apps instance set up. As we discuss later, even participants who exhibited otherwise careful data security practices did not express concern about the security implications of storing data with third parties.

Many participants (7 of 15) reported that their employers have administrative access to their work computer, particularly at larger or older organizations. From a security perspective, this arrangement may allow organizations to ensure that journalists have updated systems and do not accidentally install malware, but it may also prevent journalists from installing security tools. It could also potentially expose sensitive information to the broader organization.

Two participants reported taking actions to circumvent the administrative rights of their employers: one insisted on being granted administrative access officially, while the other silently disabled his employer's remote access due to security and privacy concerns. He also mentioned being required to provide his laptop decryption key to his employer; he complied, but then re-encrypted his laptop and kept the new key to himself.

Note-taking. The journalists we spoke to described a variety of strategies for taking notes, most commonly audio-recording (13 of 15), electronic notes (12 of 15), and handwritten notes (10 of 15). We were somewhat surprised by the prevalence of audio recording, since such recordings may be particularly sensitive. Only two participants explicitly mentioned that they record audio only when intending to publish a full transcription.

We also asked participants about whether they share their notes with others. No one we spoke with ever shares notes outside of their organization, but many (13 of 15) sometimes share portions of notes within their organization. This sharing is typically done when working with another journalist on a story or for fact-checking. Most participants reported using some kind of third-party platform (e.g., Google Docs or Dropbox) for storing and sharing information. Several mentioned explicit strategies for sanitizing or redacting notes before sharing them (e.g., using codenames or omitting information); we discuss such strategies further in Section 4.5.

Knowledge management. We identify a possible opportunity for computer security in the knowledge management practices of journalists. In particular, several participants discussed strategies for organizing their notes and references for different projects and stories over time, including the use of file system folders, Google Drive, Evernote, and Scrivener. These knowledge management techniques were all ad hoc; no two participants described identical techniques. Indeed, several participants explicitly discussed the lack of a good knowledge management tool for journalists as a challenge. As we discuss in Section 4.6, this gap represents an opportunity for integrating computer security into the journalistic process.

4.4 Security Concerns

We now turn specifically to security-related issues, considering first the security concerns voiced in our interviews. Because one researcher's prior experience in the journalism community suggested that the term "threat modeling" is familiar but not widely understood, we elicited these concerns indirectly, by asking: "Of the information that you currently store digitally, would it be problematic if it were to become known to people or organizations outside of you and/or your news organization? If so, who would be at risk?" Because the concept of risk is dependent on a judgment about vulnerability, we also asked participants about their view on what kind of sources or information they considered "sensitive," whether or not they had worked with it personally.

Concrete threats experienced. A small number of participants reported encountering direct tangible threats or harms themselves in the course of their work. For example, one journalist told us that during his time report-

Category	Concern	Number of participants (of 15)	Inter-coder agreement (κ)
Threats to sources	Discovery by government	6	0.88
	Disciplinary action (e.g., lost job)	6	0.88
	Reputation/personal consequences	6	0.88
	Generally vulnerable populations (e.g., abuse victims)	4	0.65
	Discovery by others wishing to reveal identity	3	0.80
	Physical danger	3	0.86
	Prison	2	1.00
Threats to journalist or organization	Reputation consequences (incl. loss of source's trust)	9	0.89
	Being "scooped" (i.e., journalistic competition)	6	1.00
	False or misleading information from a source	4	0.36
	Physical threats (incl. theft)	2	0.50
	Financial consequences	1	1.00
Threats to others	Political / foreign relations consequences	1	0.50
	Other	1	1.00

Table 3: **Security concerns.** We report how many participants mentioned various threats to themselves, to their sources, to their organizations, or to others. These are not necessarily threats that participants have directly encountered or acted on themselves — that is, they discussed threats both in a hypothetical sense (concerns they have) and a concrete sense (real issues they have encountered).

ing on government-related scandals, his work phones had been wiretapped, his laptop (and nothing else) had been stolen from his home, and he had received letters threatening his and his family's lives and safety. Another described communications with contacts in a foreign region, in which phone communications were regularly terminated when the conversation broached what she perceived as sensitive topics. In total, 6 participants mentioned the knowledge or strong suspicion that their or their sources' digital communications had been retroactively collected or actively monitored.

General concerns. In addition to these concrete attacks and threats, participants mentioned a range of risks that they consider in communications with sources. These concerns are organized and summarized in Table 3.

Many of the general security concerns reported by participants were in line with our expectations: governments attempting to identify sources, reputational threats or harms, and legal or disciplinary consequences. The most common concern involved reputational harm and loss of credibility by the journalist and his or her organization, largely characterized as a compromised ability to gain access to and establish trust with future sources.

Participants also mentioned several threats that we had not initially anticipated. For example, one participant discussed the possible financial consequences to his organization when it reported on a scandal involving a major advertiser. Several participants mentioned concern about being "scooped" by other journalists if they lost their competitive advantage in having early access to certain information. One participant worried that her web searches on sensitive work-related topics would make her a surveillance target in her personal life, so she avoided doing those searches on her home computer.

Overlooked concerns. We identify several security concerns that were generally overlooked by our participants, despite being well-known to computer security experts.

Third parties. Only one participant expressed concern about the trustworthiness of major third parties, such as Apple, Google, or Microsoft. While some participants expressed hesitation about how secure a certain practice is, they did not explicitly discuss these major technology providers as being a possible security risk. Unfortunately, this implicit trust assumption may not be warranted — e.g., consider reports of government or other compromises of major companies [34, 66] and the FBI's National Security Letters compelling service providers to release information [50].

Metadata. While a few participants expressed concerns about the metadata connecting them to their sources (discussed further in Section 4.5 below), most did not discuss metadata as a threat even implicitly. Indeed, even those who explicitly took steps to protect their notes or communications (e.g., using encryption) did not generally discuss the need to similarly protect metadata.

Legal concerns. Finally, there was virtually no mention in any of the interviews of the risk of lawsuit resulting from or discovery of digitally stored or communicated information. There are several possible explanations for this, though comments from most of those interviewed suggest that they did not feel their own work was ever likely to be the subject of a government investigation.

4.5 Defensive Strategies

Whether or not they had experienced concrete threats, most participants reported using some defensive strategies, including security technologies as well as non-technical or technology-avoidant strategies. Figure 4 systematizes these strategies, and Table 5 summarizes participants' use of specific security technologies.

Category	Defense	Number of participants (of 15)	Inter-coder agreement (κ)
Technical defenses	Encrypting digital notes	6	1.00
	Keeping files local (not in the cloud)	5	0.89
	Encrypted communication with colleagues	3	0.81
	Circumventing organization's admin rights on computer	2	0.50
	Encrypted communication with sources	2	0.50
	Anonymous communication (e.g., over Tor)	2	1.00
	Air-gapping a computer (keeping it off the internet)	1	1.00
	Using additional, secret devices or temporary burner phones	1	1.00
	Visually obscuring information in photos/videos (e.g., blurring)	1	0.50
Ad hoc non-technical strategies	Using code names in communications or notes	8	1.00
	Claiming bad handwriting as a defense for written notes	3	1.00
	Contacting sources through intermediaries	2	0.81
	Citing multiple sources to create plausible deniability	1	1.00
	Using some method to authenticate source	1	1.00
Explicitly avoiding technology	Communicating in person	7	0.72
	Self-censoring (avoiding saying things in notes/email)	6	0.86
	Communicating only vague information electronically	5	0.83
	Physically mailing digital data (e.g., on USB stick)	2	1.00
Physical defenses	Home alarm system	1	1.00
	Physical safe (e.g., to store notes)	1	1.00
	Shredding paper documents	1	1.00

Table 4: **Defensive techniques.** We report the number of participants who mentioned using various defensive techniques to protect themselves, their notes, and/or their sources.

	Number of participants (of 15)				Inter-coder
Security tool or technology	*Use regularly*	*Tried but don't use*	*Haven't tried*	*Not mentioned*	agreement (κ)
Dispatch	0	0	1	14	1.00
Encrypted chat (e.g., OTR, CryptoCat)	5	0	1	9	0.90
Encrypted email (e.g., GPG, Mailvelope)	4	4	1	6	0.92
Encrypted messaging (e.g., Wickr, Telegram)	0	1	0	14	1.00
Encrypted phone (e.g., SilentCircle)	0	2	0	13	1.00
Other encryption (e.g., hard drive, cloud)	5	1	0	9	1.00
Password manager	1	0	1	13	1.00
SecureDrop	0	0	1	14	1.00
Tor	2	1	0	12	0.89
VPN	2	1	0	12	1.00

Table 5: **Security tools.** This table lists security technologies discussed by participants. We report on the number who regularly use, have tried but don't regularly use, and haven't tried each tool. We consider use to be "regular" even if it depends on the sensitivity of the source or story, i.e., if the journalist regularly employs that tool when appropriate, even if not in every communication.

Non-technical defensive strategies. Since not all of our participants were computer security experts — and certainly most journalists are not — we were particularly interested in non-technical or otherwise ad hoc strategies that they have developed to protect themselves, their notes, or their sources. As reflected in Table 4, a commonly mentioned non-technical strategy is avoiding technology entirely, e.g., meeting sources face-to-face, physically mailing digital data, and/or communicating only vague information electronically. For example, P6 told us (translated from French):

> *I don't use phones, I don't send email. Sometimes I send SMS messages, but these messages are very vague.* [Later in the interview he adds:] *I don't use technical methods [to protect my sources]. I prefer to work in an old fashioned way. A little bit like Bin Laden did.*

The reference to Bin Laden echoes an issue raised in a recent report about U.S. journalists, which describes how concerns about surveillance and increased leak investigations have caused journalists to feel like they must "act like criminals" to communicate with sources [40].

Some of these non-technical strategies, however, were cited specifically for their journalistic rather than their security value. In explaining the choice to meet a source primarily in person, participant P11 noted:

> *I think it's always preferable because of the level of intimacy and information that you gain. You get better results and [...] you can sort of verify in different ways the stories that they're telling you.*

Ad hoc defensive strategies. We also uncovered a number of ad hoc strategies that make incidental use of technology. For example, participant P0 described his strategy for authenticating a source whose email address he found on a public mailing list: he asked that source to post a particular sentence on Twitter, allowing P0 to verify that the email and Twitter accounts indeed belonged to the same individual. In another example, P5 described a strategy for hiding the connection between himself and a sensitive source in the government by contacting the source through an intermediary. In particular, P5 called the source's assistant at previous job and stated a false name; when the assistant passed this message on, the source knew whom to contact.

These strategies of avoiding technology entirely or using ad hoc methods for specific cases suggest that our participants (and/or their sources) are not always comfortable with existing security technologies, and/or that these technologies do not meet their security needs in a straightforward way, as we discuss further in Section 4.6.

Technical defensive strategies. As reflected in Table 4, several participants explicitly mentioned using security technologies to protect themselves, their notes, or their sources. Table 5 summarizes specific security technologies mentioned, broken down by how often participants mentioned using these technologies.

Most commonly, participants mentioned using encryption to protect communications or stored data. Even participants with low computer security expertise often mentioned and even used encryption. For example, P5, who otherwise mentioned no technical security strategies, uses the Mac Disk Utility to encrypt virtual drives on his machine. Indeed, several participants mentioned using built-in file or disk encryption of this sort, suggesting that these tools are reasonably discoverable and usable. The lack of installation overhead may also contribute to their prevalence among our participants.

Participants who reported use of computer security technology for source communication fell roughly into two groups: those whose sources demanded it, and those who had participated in some kind of computer security training either through their workplace or at an external event. Sustained use, however, was seen only in intra-institutional communications (largely chat). Those who used these tools for communication with sources did so only sporadically (as required by a particular source), and reported an extended timeframe to become comfortable using them (particularly GPG and OTR).

We observe several security technologies that were under-represented in our interviews. For example, SecureDrop [26] and Dispatch [12], which were designed specifically for journalists, were mentioned by only one participant who did not report ever having used them.

Reasons for not using security technologies. We asked participants whether anyone (a source, a colleague, or anyone else) had ever recommended that they use any computer security tools or technologies. Of our 15 participants, 10 replied that they had received such a recommendation. Of those, however, only four began regularly using any of the recommended tools.

For participants who had never tried, or tried but did not continue using tools mentioned in the interview (see Table 5), we coded the interviews for reasons for not using security technologies. These reasons are summarized in Table 6, and we highlight a few important issues here.

Usability, reliability, and education. Echoing findings from prior studies (e.g., [65]), many participants discussed challenges related to usability of security tools and the need for education of journalists and sources about security issues. These challenges result in limited adoption of these tools among sources and colleagues, reducing their utility to even the most technically savvy journalist. For example, one participant described a situation where he and his colleagues worked with sensitive data; as the size of the group grew and included less security-versed individuals, it became harder to maintain strict data security practices (echoing prior findings about the social context surrounding such tools [27]).

In addition to the well-known usability challenges with many security tools, participant P10 described the difficulty of knowing which tools to trust:

> *A lot of services out there say they're secure, but having to know which ones are actually audited and approved by security professionals — it takes a lot of work to find that out.*

Digital divide. A challenge frequently mentioned in our interviews (by 4 of 15 participants) is the "digital divide": many sources do not understand or even have access to computer security technology, making it infeasible for journalists to use technical tools to secure their communications with these sources. As our participants described, this challenge applies particularly to vulnerable populations, such as low-income communities, abuse victims, homeless people, etc. To take just one example, P12 discussed the digital divide as follows:

> *Most of the [sensitive sources] I've worked with [are] also people who probably aren't very tech-savvy. Like, entry-level people in prisons, or something like that. So if they were really concerned about communication, I don't quite know what a secure, non-intimidatingly-techy way would be. [...] Some of them don't even necessarily have email addresses.*

Lack of institutional support for computer security. Another important challenge for some journalists attempting to use security technologies is a lack of institutional support. Though some participants described supportive organizations, 9 of 15 mentioned that they did *not* have

Category	Reasons for not using security technology	Number of participants (of 15)	Inter-coder agreement (κ)
Usability and adoption	Not enough people using it	5	0.79
	Digital divide: sources don't have/understand technology	4	0.86
	Security technology is too complicated	3	1.00
	Hard to evaluate credibility/security of a tool	2	0.50
Interference with journalism	Creates barrier to communication with sources	5	0.64
	Doesn't want to impose on sources	5	0.83
	Interferes with some other part of their work	3	1.00
Other	Work isn't sensitive enough / no one is looking	8	0.41
	Uses a non-technical strategy instead	6	0.70
	Insufficient support from organization	2	0.80
	Tool doesn't provide the needed defense	1	1.00

Table 6: **Reasons journalists report not using security technologies.** We report the number of participants who mentioned various reasons for why they haven't tried or don't regularly use computer security technologies. Note that some of these themes may overlap (i.e., a single statement made by a participant may have been coded with more than one of the themes in this table).

anyone to go to for help with computer security issues who was both within their organization and whose role explicitly involved providing technical support of this nature. Instead, 5 participants had no one to ask for help or had to go outside their organizations, while 4 received help from other journalists within their organization who happened to be knowledgable about these issues (e.g., because they cover related stories). Similarly, many participants (6 of 15) explicitly reported not having administrative privileges on their work computers, making it difficult or impossible to install security tools not officially supported by the organizations.

Inconsistencies and vulnerabilities. Finally, we reflect on several inconsistencies or vulnerabilities that we observed in the described behaviors of our participants.

A common inconsistency (observed in 5 of 15 interviews) involved protecting data effectively in one context but insufficiently in another. For example, participant P5 (quoted above) avoids using technology to communicate with sources due to real threats he has encountered (including eavesdropping, laptop theft, and death threats) — but uses his iPad (with no mention of encryption) to photograph sensitive documents provided without permission by sources.

Participants also frequently discussed or acknowledged the potential danger in a particular practice, but did not change their behavior. For example, P10 told us: *"I should have a separate work [Gmail] account but I just use my personal one"* — a sentiment echoed by other participants. As another example, when asked if he takes steps to protect his notes, P5 responded: *"I should. But no."* In another case, though a participant considered herself "comfortable" with computer security technology and worked with sensitive information, she did not use and seemingly could not name any security tools.

We also identified several vulnerabilities present in the behaviors of participants but not explicitly acknowledged by any of them. For example, while some participants explicitly mentioned meeting with sources face-to-face for security reasons (in addition to journalistic reasons), they did not mention taking precautions like leaving behind or turning off electronic devices at these meetings. Indeed, many participants (though not necessarily those using face-to-face meetings for security reasons) mentioned using their iPhones or other devices to audio-record in-person conversations with sources. Participants also frequently use document management services that sync data to a third-party cloud service, such as Google Docs and Evernote.

4.6 Needs of Journalists

A major goal of our study is to inform future efforts by the computer security community to develop tools to protect journalist-source communications. To that end, we identify needs of journalists in their communications with sources that are hampered or unfulfilled by current computer security technologies. Needs that are still unfulfilled present immediate opportunities for future work, while needs that are hampered suggest reasons why existing technologies have failed to find greater adoption.

Functions impeded by security technology. One of the reasons that participants noted for why they have not tried or do not regularly use certain security technologies is that they interfere with some component of the journalistic process. As reflected in Table 6, 3 of 15 participants mentioned this reason. Taking a closer look at which functions are impeded by existing security technologies (and should be considered in future tools for journalists), our participants mentioned the following problems:

- Anonymous communications may make it difficult for journalists to authenticate sources, or to authenticate themselves to sources.
- Using security tools may impede communications with colleagues who don't use or understand them.
- Constraints on communications with sources may reduce the quality of information journalists can get.

For example, P13 described the tension between anonymous sources and authenticity:

> *If I don't know who they are and can't check their background, I'm not going to use the information they give. Anonymous sourcing is fine if I know who they are, and I've checked who they are, and my editor knows who they are, but they can't keep that from me and then expect me to use the information they provide.*

In other words, a source's communications must be anonymous to everyone but the journalist with whom they are communicating, and that journalist must be able to prove the authenticity of that source to others (e.g., their editor). This need suggests that tools like SecureDrop [26], which supports anonymous document drops for journalists, are unlikely to be widely adopted in isolation — highlighting the need for the computer security community to interface with the journalism community.

On the flip side, P6 discussed the need for sources to authenticate him when he attempts to reach them, describing how sources are unlikely to answer the phone if they cannot see who is calling them.

In order to develop computer security technologies that will be widely adopted by journalists, the computer security community must understand such failures of existing tools. We emphasize that these failures are not merely the result of computer security tools being hard to use (a common culprit [65]) but often arise when a tool did not sufficiently account for functions important in a journalist's process, such as the ability to authenticate sources. In Section 5, we discuss what the specific failures above mean for where technologists should focus their efforts in this space.

Security needs unfulfilled by technology. In the previous paragraphs, we described needs of journalists that we infer from their reasons for not using certain security technologies. In addition to making these inferences, we also asked participants to report specifically on any concerns or issues related to computer security to which they have not yet found a good technical solutions (i.e., "I wish somebody would build a tool that does X"). From the responses to this question, we extract several technical security-related needs currently unaddressed.

Usability, education, and adoption. As discussed above, several participants mentioned usability concerns (the need for more usable security tools) and education concerns (the need for education about these issues for both sources and journalists), both for themselves and to increase the adoption of security technologies among others. Specifically, participants asked for better and easier-to-use tools or services for encrypted email, encrypted file sharing, and encrypted phone calls, as well as ways to prevent emails from being accidentally forwarded and to keep sensitive data off the Internet (e.g., air-gapping).

Mutual authentication and first contact. Some participants discussed ad hoc strategies to authenticate sources, or to authenticate themselves to sources. As noted above, current security tools for journalists may hamper these needs, rather than addressing them. Participant P0 spoke in particular about the tension between anonymity and authentication in first contact:

> *The first contact is never or very rarely anonymous or protected. If someone wants to give me some information and we don't already know each other, how would he do it? He could send me an email, yeah, okay — but then how could I be sure it's him? Unless he contacts me with his real identity first. It's very difficult to have the first contact secure.*

In this "first contact" problem, it is nearly impossible for journalists to entirely avoid *some* metadata trail when communicating with a source, since their initial contact will almost universally take place over a channel whose metadata is associated with the journalist's professional identity (e.g. telephone, email, or social media). Given the pivotal role that metadata has played in recent leak prosecutions [54], this is a significant security concern.

Digital divide. As discussed above, several participants expressed the need for better security technologies that work across the digital divide, in order to protect their communications with sources who have low technical expertise and/or limited access to technology.

These unfulfilled needs represent immediate opportunities for future work on secure journalist-source communications within the computer security community, with varying types and degrees of challenge. We discuss these new directions further in Section 5.

Other technical needs. Though we asked participants specifically about unaddressed issues related to computer security, a few also (or instead) expressed more general technical needs that have security implications.

For example, several participants discussed the difficulty of manually transcribing audio recordings of interviews and expressed a desire for better machine transcription. Our interviews show this unaddressed need led to at least one insecure practice by a participant, who described planning to use her iPhone's or Mac's speech-to-text feature to transcribe audio recordings of interviews with sources, seemingly unaware that this might send the audio of potentially sensitive interviews to the cloud [9]. Thus, as journalists develop ad hoc workarounds for tasks where a technical solution is missing from their toolset, they may unintentionally introduce vulnerabilities into their process.

More generally, as mentioned above, several participants discussed the need for a systematic knowledge management tool for journalists. P11 was most explicit:

> *There were different kinds of litigation software that I was familiar with as a lawyer, where, let's say, you have a massive case, where you have a document dump that has 15,000 documents. [...] There are programs that help you consolidate and put them into a secure database. So it's searchable [and provides a secure place where you can see everything related to a story at once]. I don't know of anything like that for journalism.*

This absence of a dedicated knowledge management tool for journalists represents an opportunity for computer security. If such a knowledge management tool seamlessly integrated computer security techniques to protect stored data and communications without significant effort on the part of the journalist, it would significantly raise the bar for the security of journalist-source communications.

5 Discussion

We elaborate on the implications of our findings for the computer security community and make concrete recommendations for how those considering journalist-source communications can most fruitfully direct their efforts.

5.1 Key Take-Aways

From the perspective of the computer security community, we consider the following take-aways to be the most important ones from our findings:

- Journalists commonly make decisions about how to communicate with sources based on the technical access and comfort level of the sources themselves. Thus, limited adoption of technical security tools for journalist-source communications stems in large part from the limited technical access and expertise of certain vulnerable populations.
- Journalists face technical challenges unrelated to computer security, including the lack of systematic knowledge management tools and limited technical support for transcription. In developing ad hoc strategies to deal with these challenges, journalists sometimes introduce additional security vulnerabilities into their practices.
- A journalist's organization plays an important role in his or her access to and competence with computer security technologies. Organizations that restrict a journalist's ability to install security (or other software) tools, or where many employees have limited technical expertise, reduce the effectiveness and adoption of security and other technologies.
- An important reason for the failure of some security tools in the journalistic context is their incompatibility with some essential aspect of the journalistic process. A tool that increases barriers to communication or prevents a journalist from determining the authenticity of a source will see limited adoption.

5.2 Recommendations

In addition to supporting ongoing efforts at educating and training journalists with respect to existing computer security technologies (e.g., [17, 43, 47, 62]), we distill from our findings the following recommendations for where the computer security community should focus its efforts.

First contact and authentication. The challenge of securing (or retroactively protecting) a journalist's first contact with a source remains a hard problem, especially given the tension between anonymity and mutual authentication. Determining authenticity, both of sources and of journalists, is of fundamental importance in the journalistic context and should be addressed explicitly by anonymous communication tools. For instance, successful approaches might leverage existing identity networks, as with the participant who asked his source to post a specific sentence on Twitter — similar to social authenticity proofs used by Keybase (`https://keybase.io/`).

Metadata protection. Protecting metadata of journalist-source communications is crucial, especially in light of successful leak prosecutions based on metadata information [54]. In practice, metadata is both legally and technically unprotected: none of the defensive strategies described by our participants was truly foolproof, especially with respect to metadata. Protecting metadata is challenging because it requires that both journalists and sources understand the risk, because it is brittle (e.g., a single failure to communicate securely can compromise dozens or hundreds of exchanges), and because it can conflict with other journalistic needs (e.g., the need for authentication in first contact). The computer security community should consider metadata protection in this context and develop effective, usable, and transparent solutions that can account for long-term communications of varying sensitivity.

Focus on sources. Since the methods and security of journalist-source communications often depend on the technical expertise and access of sources, the computer security community should focus not only on educating and building tools for journalists but also for sources. Enabling and improving access to computer security technologies for low-income and vulnerable populations (e.g., through a collaboration with public libraries and/or by supporting "dumb" phones or other access methods) will provide benefits to these communities far beyond their interaction with journalists. Meanwhile, future studies should also interview and/or survey sources to shed light on their perspectives and needs.

Knowledge management. Our findings suggest that journalists desire — but lack — a solution for systematic knowledge management to support storing, organizing, searching, and indexing story-related data and documents. This need presents an opportunity for computer security: if security techniques and tools are seamlessly and usably integrated into a well-designed knowledge management tool for journalists, these could see wide adoption within the industry and significantly raise the bar for the security of journalistic practices. For example, given the reliance among our interviewees on third-party cloud storage, a secure (and easy-to-use) cloud storage solution integrated into such a knowledge management tool would provide significant benefits. A knowledge management tool that also supports secure communication — such as encrypted chat or email within the organization — would also benefit affiliated but non-staff members of the organization (e.g., freelancers).

Understanding the journalistic process. We encourage the technical computer security community to continue engaging closely with the journalism community. While many of the themes observed in our interviews and highlighted in this paper may be well-known within the journalism community, several of them were surprising to us. The prevalence of ad hoc defensive strategies among our participants suggests mismatches between existing computer security tools and the needs and understandings of journalists. To create technical designs that address journalists' most significant security problems without compromising necessary professional practices, the computer security community must develop a deep understanding of the journalistic process. These efforts are likely to be most valuable if they are iterative, involving the development of tools that are then evaluated and refined in the field among the target population.

Broader applicability. Finally, successful techniques for securing journalist-source communications are likely to apply to — or provide lessons for — other contexts as well, such as communications between lawyers and their clients, between doctors and patients, in government operations, among dissidents and activists, and for other everyday users of technology.

6 Conclusion

Though journalists are often considered likely users and beneficiaries of secure communication and data storage tools, their practices have not been been studied in depth by the academic computer security community. To close this gap and to inform ongoing and future work on computer security for journalists, we conducted an in-depth, qualitative study of 15 journalists at well-respected journalistic institutions in the U.S. and France.

Our findings provide insight into the general journalistic practices and specific security concerns of journalists, as well as the successes and failures of existing security technologies within the journalistic context. Perhaps most importantly, we find that existing security tools have seen limited adoption not just due to usability issues (a common culprit) but because of a mismatch between between the assumed and actual practices, priorities, and constraints of journalists. This mismatch suggests that secure journalistic practices depend on a meaningful collaboration between the computer security and the journalism communities; we take an important step towards such a collaboration in this work.

Acknowledgements

We gratefully acknowledge our anonymous reviewers for their helpful feedback. We also thank Greg Akselrod and Kelly Caine for valuable discussions; Raymong Cheng, Roxana Geambasu, Tadayoshi Kohno, and Sam Sudar for feedback on earlier drafts; and Tamara Denning for guidance on interview coding. Most importantly, we thank our interviewees very much for their participation in our study. This research is supported in part by NSF Award CNS-1463968.

References

[1] CCleaner. http://ccleaner.en.softonic.com/.

[2] Cryptocat: Chat with privacy. https://crypto.cat/.

[3] Silent Circle. https://silentcircle.com/.

[4] Tails: The Amnesic Incognito Live System. https://tails.boum.org/.

[5] TrueCrypt. http://truecrypt.sourceforge.net/.

[6] Wickr. https://wickr.com/.

[7] WITNESS, 2014. http://witness.org.

[8] APPLE. FileVault. http://support.apple.com/kb/ht4790.

[9] APPLE. OS X Mavericks: Use Dictation to create messages and documents, May 2014. http://support.apple.com/kb/PH14361.

[10] ARDAGNA, C. A., JAJODIA, S., SAMARATI, P., AND STAVROU, A. Providing Mobile Users' Anonymity in Hybrid Networks. In *European Symposium on Research in Computer Security (ESORICS)* (2010).

[11] BALL, J. GCHQ captured emails of journalists from top international media. The Guardian, Jan. 2015. http://www.theguardian.com/uk-news/2015/jan/19/gchq-intercepted-emails-journalists-ny-times-bbc-guardian-le-monde-reuters-nbc-washington-post.

[12] Biscuitwala, K., Bult, W., Mathias Lecuyer, T. J. P., Ross, M. K. B., Chaintreau, A., Haseman, C., Lam, M. S., and McGregor, S. E. Secure, Resilient Mobile Reporting. In *Proceedings of ACM SIGCOMM* (2013).

[13] Blond, S. L., Uritesc, A., Gilbert, C., Chua, Z. L., Saxena, P., and Kirda, E. A look at targeted attacks through the lense of an ngo. In *23rd USENIX Security Symposium* (2014).

[14] Borisov, N., Goldberg, I., and Brewer, E. Off-the-record communication, or, why not to use PGP. In *Proceedings of the ACM Workshop on Privacy in the Electronic Society* (2004).

[15] Brennan, M., Metzroth, K., and Stafford, R. Building Effective Internet Freedom Tools: Needfinding with the Tibetan Exile Community. In *7th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs)* (2014).

[16] Bump, P. So, You Want to Hide from the NSA? Your Guide to the Nearly Impossible. The Wire, July 2013. `http://www.thewire.com/technology/2013/07/so-you-want-hide-nsa-your-guide-nearly-impossible/66942/`.

[17] Carlo, S., and Kamphuis, A. Information Security for Journalists. The Centre for Investigative Journalism, July 2014. `http://www.tcij.org/resources/handbooks/infosec`.

[18] Charmaz, K. *Constructing Grounded Theory*, second ed. SAGE Publications Ltd, 2014.

[19] Cohen, J. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement 20*, 1 (1960), 37.

[20] Czeskis, A., Mah, D., Sandoval, O., Smith, I., Koscher, K., Appelbaum, J., Kohno, T., and Schneier, B. DeadDrop/StrongBox Security Assessment. Tech. Rep. UW-CSE-13-08-02, Department of Computer Science and Engineering, University of Washington, 2013.

[21] Danezis, G., and Diaz, C. A survey of anonymous communication channels. Tech. Rep. MSR-TR-2008-35, Microsoft Research, January 2008.

[22] Dingledine, R., Mathewson, N., and Syverson, P. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium* (2004).

[23] Edman, M., and Yener, B. On anonymity in an electronic society: A survey of anonymous communication systems. *ACM Computing Surveys 42*, 1 (2009).

[24] Fleiss, J. L., Levin, B., and Paik, M. C. *Statistical Methods for Rates and Proportions*, 3 ed. John Wiley & Sons, New York, 2003.

[25] Franceschi-Bicchierai, L. Meet the Man Hired to Make Sure the Snowden Docs Aren't Hacked. Mashable, May 2014. `http://mashable.com/2014/05/27/micah-lee-greenwald-snowden/`.

[26] Freedom of the Press Foundation. SecureDrop (formerly known as DeadDrop, originally developed by Aaron Swartz), 2013. `https://pressfreedomfoundation.org/securedrop`.

[27] Gaw, S., Felten, E. W., and Fernandez-Kelly, P. Secrecy, flagging, and paranoia: Adoption criteria in encrypted e-mail. In *Proceedings of CHI* (2006).

[28] Geambasu, R., Kohno, T., Krishnamurthy, A., Levy, A., Levy, H. M., Gardner, P., and Moscaritolo, V. New directions for self-destructing data. Tech. Rep. UW-CSE-11-08-01, University of Washington, 2011.

[29] Geambasu, R., Kohno, T., Levy, A., and Levy, H. M. Vanish: Increasing Data Privacy with Self-Destructing Data. In *Proceedings of the 18th USENIX Security Symposium* (2009).

[30] Glaser, B. G., and Strauss, A. L. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine Publishing Company, Chicago, 1967.

[31] GnuPG. GNU Privacy Guard. `https://www.gnupg.org/`.

[32] Goldberg, I. Off-the-record messaging. `https://otr.cypherpunks.ca/`.

[33] Greenberg, A. Whistleblowers Beware: Apps Like Whisper and Secret Will Rat You Out. Wired, May 2014. `http://www.wired.com/2014/05/whistleblowers-beware/`.

[34] Greenwald, G. *No Place To Hide: Edward Snowden, the NSA, and the U.S. Surveillance State*. Metropolitan Books, 2014.

[35] Guest, G., Bunce, A., and Johnson, L. How many interviews are enough? an experiment with data saturation and variability. *Field Methods 18*, 1 (2006).

[36] Hardy, S., Crete-Nishihata, M., Kleemola, K., Senft, A., Sonne, B., Wiseman, G., Gill, P., and Deibert, R. J. Targeted threat index: Characterizing and quantifying politically-motivated targeted malware. In *23rd USENIX Security Symposium* (2014).

[37] Heninger, N., Poitras, L., Gillum, J., and Angwin, J. How Journalists Use Crypto To Protect Sources. Panel Discussion at 31th Chaos Communication Congress (31c3) of the Chaos Computer Club (CCC), Jan. 2015. `https://www.youtube.com/watch?v=aviUKt7adU8`.

[38] Hill, K. Lavabit's Ladar Levison: 'If You Knew What I Know About Email, You Might Not Use It'. Forbes, Aug. 2013. `http://www.forbes.com/sites/kashmirhill/2013/08/09/lavabits-ladar-levison-if-you-knew-what-i-know-about-email-you-might-not-use-it/`.

[39] Holmes, H., Moser, A., and Gellman, B. Drop It Like It's Hot: Secure Sharing and Radical OpSec for Investigative Journalists. Panel Discussion at Hope X, July 2014. `http://www.hope.net/schedule.html#dropitlike`.

[40] Human Rights Watch. With Liberty to Monitor All: How Large-Scale US Surveillance is Harming Journalism, Law, and American Democracy, July 2014. `http://www.hrw.org/node/127364`.

[41] Huntley, S., and Marquis-Boire, M. Tomorrow's News is Today's Intel: Journalists as Targets and Compromise Vectors. BlackHat Asia, Mar. 2014. `https://www.blackhat.com/docs/asia-14/materials/Huntley/BH_Asia_2014_Boire_Huntley.pdf`.

[42] Internews Center for Innovation & Learning. Digital Security and Journalists: A SnapShot of Awareness and Practices in Pakistan, May 2012. `https://www.internews.org/sites/default/files/resources/Internews_PK_Secure_Journalist_2012-08.pdf`.

[43] LEE, M. Encryption Works: How to Protect Your Privacy in the Age of NSA Surveillance. Freedom of the Press Foundation, July 2013. https://pressfreedomfoundation.org/sites/default/files/encryption_works.pdf.

[44] LEVISON, L. Lavabit, 2004. http://lavabit.com/.

[45] MARCZAK, W. R., SCOTT-RAILTON, J., MARQUIS-BOIRE, M., AND PAXSON, V. When governments hack opponents: A look at actors and technology. In *23rd USENIX Security Symposium* (2014).

[46] MARIMOW, A. E. Justice Departments scrutiny of Fox News reporter James Rosen in leak case draws fire. The Washington Post, May 2013. http://www.washingtonpost.com/local/justice-departments-scrutiny-of-fox-news-reporter-james-rosen-in-leak-case-draws-fire/2013/05/20/c6289eba-c162-11e2-8bd8-2788030e6b44_story.html.

[47] MCGREGOR, S. E. Digital Security and Source Protection for Journalists. Tow Center for Digital Journalism, July 2014. http://towcenter.org/blog/digital-security-and-source-protection-for-journalists/.

[48] MITCHELL, A., HOLCOMB, J., AND PURCELL, K. Investigative journalists and digital security: Perceptions of vulnerability and changes in behavior. Pew Research Center, Feb. 2015. http://www.journalism.org/files/2015/02/PJ_InvestigativeJournalists_0205152.pdf.

[49] NORCIE, G., BLYTHE, J., CAINE, K., AND CAMP, L. J. Why Johnny Can't Blow the Whistle: Identifying and Reducing Usability Issues in Anonymity Systems. In *Proceedings of the Network and Distributed System Security Symposium (NDSS) Workshop on Usable Security (USEC)* (2014).

[50] OFFICE OF THE INSPECTOR GENERAL. A Review of the Federal Bureau of Investigation's Use of National Security Letters. U.S. Department of Justice, Aug. 2014. http://www.justice.gov/oig/reports/2014/s1408.pdf.

[51] OLSON, P. E-mail's Big Privacy Problem: Q&A With Silent Circle Co-Founder Phil Zimmermann, Aug. 2013. http://www.forbes.com/sites/parmyolson/2013/08/09/e-mails-big-privacy-problem-qa-with-silent-circle-co-founder-phil-zimmermann/.

[52] PERLMAN, R. The ephemerizer: Making data disappear. *Journal of Information System Security 1* (2005), 51–68.

[53] REARDON, J., BASIN, D., AND CAPKUN, S. SoK: Secure Data Deletion. In *Proceedings of the IEEE Symposium on Security and Privacy* (2013).

[54] SAVAGE, C. Court Rejects Appeal Bid by Writer in Leak Case. The New York Times, Oct. 2013. http://www.nytimes.com/2013/10/16/us/court-rejects-appeal-bid-by-writer-in-leak-case.html.

[55] SAVAGE, C., AND KAUFMAN, L. Phone Records of Journalists Seized by U.S. The New York Times, May 2013. http://www.nytimes.com/2013/05/14/us/phone-records-of-journalists-of-the-associated-press-seized-by-us.html.

[56] SCHAFFER, M. Who Can View My Snaps and Stories, Oct. 2013. http://blog.snapchat.com/post/64036804085/who-can-view-my-snaps-and-stories.

[57] SECONDMUSE. Information Security for Journalists, June 2014. https://speakerdeck.com/secondmuse/understanding-internet-freedom-vietnams-digital-activists.

[58] SIERRA, J. L. Digital and Mobile Security for Mexican Journalists and Bloggers. Freedom House, 2013. http://www.freedomhouse.org/report/special-reports/digital-and-mobile-security-mexican-journalists-and-bloggers.

[59] SYRIA JUSTICE AND ACCOUNTABILITY CENTRE. Violations Database, 2014. http://syriaaccountability.org/database/.

[60] THE GUARDIAN PROJECT. Secure mobile apps. https://guardianproject.info/apps.

[61] TOR. Tor Browser Bundle. https://www.torproject.org/projects/torbrowser.html.en.

[62] TOW CENTER FOR DIGITAL JOURNALISM. Journalism After Snowden. Columbia Journalism School, 2014. http://towcenter.org/journalism-after-snowden/.

[63] UNGER, N., DECHAND, S., BONNEAU, J., FAHL, S., PERL, H., GOLDBERG, I., AND SMITH, M. SoK: Secure Messaging. In *Proceedings of the IEEE Symposium on Security and Privacy* (2015).

[64] WHISPER SYSTEMS. RedPhone and TextSecure. https://whispersystems.org/.

[65] WHITTEN, A., AND TYGAR, J. D. Why johnny can't encrypt: A usability evaluation of pgp 5.0. In *Proceedings of the 8th USENIX Security Symposium* (1999).

[66] ZETTER, K. Sony got hacked hard: What we know and don?t know so far. Wired, Dec. 2014. http://www.wired.com/2014/12/sony-hack-what-we-know/.

[67] ZIMMERMANN, P. R. *The Official PGP User's Guide*. MIT Press, Cambridge, MA, USA, 1995.

Constants Count: Practical Improvements to Oblivious RAM

Ling Ren *MIT*

Christopher Fletcher *MIT*

Albert Kwon *MIT*

Emil Stefanov *UC Berkeley*

Elaine Shi *Cornell University*

Marten van Dijk *UConn*

Srinivas Devadas *MIT*

Abstract

Oblivious RAM (ORAM) is a cryptographic primitive that hides memory access patterns as seen by untrusted storage. This paper proposes Ring ORAM, the most bandwidth-efficient ORAM scheme for the small client storage setting in both theory and practice. Ring ORAM is the first tree-based ORAM whose bandwidth is independent of the ORAM bucket size, a property that unlocks multiple performance improvements. First, Ring ORAM's overall bandwidth is 2.3× to 4× better than Path ORAM, the prior-art scheme for small client storage. Second, if memory can perform simple untrusted computation, Ring ORAM achieves constant online bandwidth ($\sim 60\times$ improvement over Path ORAM for practical parameters). As a case study, we show Ring ORAM speeds up program completion time in a secure processor by 1.5× relative to Path ORAM. On the theory side, Ring ORAM features a tighter and significantly simpler analysis than Path ORAM.

1 Introduction

With cloud computing and storage gaining popularity, privacy of users' sensitive data has become a large concern. It is well known, however, that encryption alone is not enough to ensure data privacy. Even after encryption, a malicious server still learns a user's access pattern, e.g., how frequently each piece of data is accessed, if the user scans, binary searches or randomly accesses her data at different stages. Prior works have shown that access patterns can reveal a lot of information about encrypted files [14] or private user data in computation outsourcing [32, 18].

Oblivious RAM (ORAM) is a cryptographic primitive that *completely* eliminates the information leakage in memory access traces. In an ORAM scheme, a *client* (e.g., a local machine) accesses data blocks residing on a *server*, such that for any two logical access sequences of the same length, the observable communications between the client and the server are computationally indistinguishable.

ORAMs are traditionally evaluated by *bandwidth*—the number of blocks that have to be transferred between the client and the server to access one block, *client storage*—the amount of trusted local memory required at the client side, and *server storage*—the amount of untrusted memory required at the server side. All three metrics are measured as functions of N, the total number of data blocks in the ORAM.

A factor that determines which ORAM scheme to use is whether the client has a large (GigaBytes or larger) or small (KiloBytes to MegaBytes) storage budget. An example of large client storage setting is remote oblivious file servers [30, 17, 24, 3]. In this setting, a user runs on a local desktop machine and can use its main memory or disk for client storage. Given this large client storage budget, the preferred ORAM scheme to date is the SSS construction [25], which has about $1 \cdot \log N$ bandwidth and typically requires GigaBytes of client storage.

In the same file server application, however, if the user is instead on a mobile phone, the client storage will have to be small. A more dramatic example for small client storage is when the client is a remote secure processor — in which case client storage is restricted to the processor's scarce on-chip memory. Partly for this reason, all secure processor proposals [18, 16, 8, 31, 22, 7, 5, 6] have adopted Path ORAM [27] which allows for small (typically KiloBytes of) client storage.

The majority of this paper focuses on the small client storage setting and Path ORAM. In fact, our construction is an improvement to Path ORAM. However, in Section 7, we show that our techniques can be easily extended to obtain a competitive large client storage ORAM.

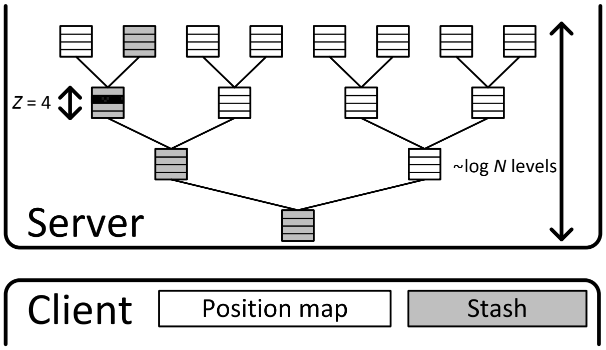


Figure 1: Path ORAM server and client storage. Suppose the **black** block is mapped to the shaded path. In that case, the block may reside in any slot along the path or in the stash (client storage).

1.1 Path ORAM and Challenges

We now give a brief overview of Path ORAM (for more details, see [27]). Path ORAM follows the tree-based ORAM paradigm [23] where server storage is structured as a binary tree of roughly $\log N$ levels. Each node in the tree is a bucket that can hold up to a small number Z of data blocks. Each path in the tree is defined as the sequence of buckets from the root of the tree to some leaf node. Each block is mapped to a random path, and must reside somewhere on that path. To access a block, the Path ORAM algorithm first looks up a position map, a table in client storage which tracks the path each block is currently mapped to, and then reads all the ($\sim Z \log N$) blocks on that path into a client-side data structure called the stash. The requested block is then remapped to a new random path and the position map is updated accordingly. Lastly, the algorithm invokes an eviction procedure which writes the same path we just read from, percolating blocks down that path. (Other tree-based ORAMs use different eviction algorithms that are less effective than Path ORAM, and hence the worse performance.)

The bandwidth of Path ORAM is $2Z \log N$ because each access reads and writes a path in the tree. To prevent blocks from accumulating in client storage, the bucket size Z has to be at least 4 (experimentally verified [27, 18]) or 5 (theoretically proven [26]).

We remind readers not to confuse the above read/write path operation with reading/writing data blocks. In ORAM, both reads and writes to a data block are served by the read path operation, which moves the requested block into client storage to be operated upon secretly. The sole purpose of the write path operation is to evict blocks from the stash and percolate blocks down the tree.

Despite being a huge improvement over prior schemes, Path ORAM is still plagued with several important challenges. First, the constant factor $2Z \geq 8$ is substantial, and brings Path ORAM's bandwidth overhead to $> 150\times$ for practical parameterizations. In contrast, the SSS construction does not have this bucket size parameter and can achieve close to $1 \cdot \log N$ bandwidth. (This bucket-size-dependent bandwidth is exactly why Path ORAM is dismissed in the large client storage setting.)

Second, despite the importance of overall bandwidth, online bandwidth—which determines response time—is equally, if not more, important in practice. For Path ORAM, half of the overall bandwidth must be incurred online. Again in contrast, an earlier work [3] reduced the SSS ORAM's online bandwidth to $O(1)$ by granting the server the ability to perform simple XOR computations. Unfortunately, their techniques do not apply to Path ORAM.

	Online Bandwidth	Overall Bandwidth
Path ORAM	$Z \log N = 4 \log N$	$2Z \log N = 8 \log N$
Ring ORAM	$\sim 1 \cdot \log N$	3-3.5$\log N$
Ring ORAM + XOR	~ 1	2-2.5$\log N$

Table 1: **Our contributions.** Overheads are relative to an insecure system. Ranges in constants for Ring ORAM are due to different parameter settings. The bandwidth cost of tree ORAM recursion [23, 26] is small ($< 3\%$) and thus excluded. XOR refers to the XOR technique from [3].

1.2 Our Contributions

In this paper, we propose Ring ORAM to address both challenges simultaneously. Our key technical achievement is to carefully re-design the tree-based ORAM such that the online bandwidth is $O(1)$, and the amortized overall bandwidth is independent of the bucket size. We compare bandwidth overhead with Path ORAM in Table 1. The major contributions of Ring ORAM include:

- **Small online bandwidth.** We provide the first tree-based ORAM scheme that achieves ~ 1 online bandwidth, relying only on very simple, untrusted computation logic on the server side. This represents at least $60\times$ improvement over Path ORAM for reasonable parameters.

- **Bucket-size independent overall bandwidth.** While all known tree-based ORAMs incur an overall bandwidth cost that depends on the bucket size, Ring ORAM eliminates this dependence, and improves overall bandwidth by $2.3\times$ to $4\times$ relative to Path ORAM.

- **Simple and tight theoretical analysis.** Using novel proof techniques based on Ring ORAM's eviction algorithm, we obtain a much simpler and tighter theoretical analysis than that of Path ORAM. Of independent interest, we note that the proof of Lemma 1 in [27], a crucial lemma for both Path ORAM and this paper, is incomplete (the lemma itself is correct). We give a rigorous proof for that lemma in this paper.

As mentioned, one main application of small client storage ORAM is for the secure processor setting. We simulate Ring ORAM in the secure processor setting and confirm that the improvement in bandwidth over Path ORAM translates to a 1.5× speedup in program completion time. Combined with all other known techniques, the average program slowdown from using an ORAM is 2.4× over a set of SPEC and database benchmarks.

Extension to larger client storage. Although our initial motivation was to design an optimized ORAM scheme under small client storage, as an interesting by-product, Ring ORAM can be easily extended to achieve competitive performance in the large client storage setting. This makes Ring ORAM a good candidate in oblivious cloud storage, because as a tree-based ORAM, Ring ORAM is easier to analyze, implement and de-amortize than hierarchical ORAMs like SSS [25]. Therefore, Ring ORAM is essentially *a united paradigm for ORAM constructions in both large and small client storage settings.*

Organization. In the rest of this introduction, we give an overview of our techniques to improve ORAM's online and overall bandwidth. Section 2 gives a formal security definition for ORAM. Section 3 explains the Ring ORAM protocol in detail. Section 4 gives a complete formal analysis for bounding Ring ORAM's client storage. Section 5 analyzes Ring ORAM's bandwidth and gives a methodology for setting parameters optimally. Section 6 compares Ring ORAM to prior work in terms of bandwidth vs. client storage and performance in a secure processor setting. Section 7 describes how to extend Ring ORAM to the large client storage setting. Section 8 gives related work and Section 9 concludes.

1.3 Overview of Techniques

We now explain our key technical insights. At a high level, our scheme also follows the tree-based ORAM paradigm [23]. Server storage is a binary tree where each node (a bucket) contains up to Z blocks and blocks percolate down the tree during ORAM evictions. We introduce the following non-trivial techniques that allow us to achieve significant savings in both online and overall bandwidth costs.

Eliminating online bandwidth's dependence on bucket size. In Path ORAM, reading a block would amount to reading and writing all Z slots in all buckets on a path. Our first goal is to *read only one block from each bucket* on the path. To do this, we randomly permute each bucket and store the permutation in each bucket as additional metadata. Then, by reading only metadata, the client can determine whether the requested block is in the present bucket or not. If so, the client relies on the stored permutation to read the block of interest from its random offset. Otherwise, the client reads a "fresh" (unread) dummy block, also from a random offset. We stress that the metadata size is typically much smaller than the block size, so the cost of reading metadata can be ignored.

For the above approach to be secure, it is imperative that each block in a bucket should be read at most once—a key idea also adopted by Goldreich and Ostrovsky in their early ORAM constructions [11]. Notice that any real block is naturally read only once, since once a real block is read, it will be invalidated from the present bucket, and relocated somewhere else in the ORAM tree. But dummy blocks in a bucket can be exhausted if the bucket is read many times. When this happens (which is public information), Ring ORAM introduces an *early reshuffle* procedure to reshuffle the buckets that have been read too many times. Specifically, suppose that each bucket is guaranteed to have S dummy blocks, then a bucket must be reshuffled every S times it is read.

We note that the above technique also gives an additional nice property: out of the $O(\log N)$ blocks the client reads, only 1 of them is a real block (i.e., the block of interest); all the others are dummy blocks. If we allow some simple computation on the memory side, we can immediately apply the XOR trick from Burst ORAM [3] to get $O(1)$ online bandwidth. In the XOR trick, the server simply XORs these encrypted blocks and sends a single, XOR'ed block to the client. The client can reconstruct the ciphertext of all the dummy blocks, and XOR them away to get back the encrypted real block.

Eliminating overall bandwidth's dependence on bucket size. Unfortunately, naïvely applying the above strategy will dramatically increase offline and overall bandwidth. The more dummy slots we reserve in each bucket (i.e., a large S), the more expensive ORAM evictions become, since they have to read and write all the blocks in a bucket. But if we reserve too few dummy slots, we will frequently run out of dummy blocks and have to call *early reshuffle*, also increasing overall bandwidth.

We solve the above problem with several additional techniques. First, we design a new eviction procedure that improves eviction quality. At a high level, Ring

ORAM performs evictions on a path in a similar fashion as Path ORAM, but eviction paths are selected based on a reverse lexicographical order [9], which evenly spreads eviction paths over the entire tree. The improved eviction quality allows us to perform evictions less frequently, only once every A ORAM accesses, where A is a new parameter. We then develop a proof that crucially shows A can approach $2Z$ while still ensuring negligible ORAM failure probability. The proof may be of independent interest as it uses novel proof techniques and is significantly simpler than Path ORAM's proof. The **amortized offline bandwidth** is now roughly $\frac{2Z}{A}\log N$, which **does not depend on the bucket size Z either**.

Second, bucket reshuffles can naturally piggyback on ORAM evictions. The balanced eviction order further ensures that every bucket will be reshuffled regularly. Therefore, we can set the reserved dummy slots S in accordance with the eviction frequency A, such that early reshuffles contribute little ($< 3\%$) to the overall bandwidth.

Putting it all Together. None of the aforementioned ideas would work alone. Our final product, Ring ORAM, stems from intricately combining these ideas in a non-trivial manner. For example, observe how our two main techniques act like two sides of a lever: (1) permuted buckets such that only 1 block is read per bucket; and (2) high quality and hence less frequent evictions. While permuted buckets make reads cheaper, they require adding dummy slots and would dramatically increase eviction overhead without the second technique. At the same time, less frequent evictions require increasing bucket size Z; without permuted buckets, ORAM reads blow up and nullify any saving on evictions. Additional techniques are needed to complete the construction. For example, early reshuffles keep the number of dummy slots small; piggyback reshuffles and load-balancing evictions keep the early reshuffle rate low. Without all of the above techniques, one can hardly get any improvement.

2 Security Definition

We adopt the standard ORAM security definition. Informally, the server should not learn anything about: 1) which data the client is accessing; 2) how old it is (when it was last accessed); 3) whether the same data is being accessed (linkability); 4) access pattern (sequential, random, etc); or 5) whether the access is a read or a write. Like previous work, we do not consider information leakage through the timing channel, such as when or how frequently the client makes data requests.

Notation	Meaning
N	Number of real data blocks in ORAM
L	Depth of the ORAM tree
Z	Maximum number of real blocks per bucket
S	Number of slots reserved for dummies per bucket
B	Data block size (in bits)
A	Eviction rate (larger means less frequent)
$\mathcal{P}(l)$	Path l
$\mathcal{P}(l,i)$	The i-th bucket (towards the root) on $\mathcal{P}(l)$
$\mathcal{P}(l,i,j)$	The j-th slot in bucket $\mathcal{P}(l,i)$

Table 2: ORAM parameters and notations.

Definition 1. (ORAM Definition) *Let*

$$\overleftarrow{y} = ((\mathsf{op}_M, \mathsf{addr}_M, \mathsf{data}_M), \ldots, (\mathsf{op}_1, \mathsf{addr}_1, \mathsf{data}_1))$$

denote a data sequence of length M, where op_i *denotes whether the i-th operation is a read or a write,* addr_i *denotes the address for that access and* data_i *denotes the data (if a write). Let* $\mathsf{ORAM}(\overleftarrow{y})$ *be the resulting sequence of operations between the client and server under an ORAM algorithm. The ORAM protocol guarantees that for any* $\overleftarrow{y}$ *and* $\overleftarrow{y}'$, $\mathsf{ORAM}(\overleftarrow{y})$ *and* $\mathsf{ORAM}(\overleftarrow{y}')$ *are computationally indistinguishable if* $|\overleftarrow{y}| = |\overleftarrow{y}'|$, *and also that for any* $\overleftarrow{y}$ *the data returned to the client by ORAM is consistent with* $\overleftarrow{y}$ *(i.e., the ORAM behaves like a valid RAM) with overwhelming probability.*

We remark that for the server to perform computations on data blocks [3], $\mathsf{ORAM}(\overleftarrow{y})$ and $\mathsf{ORAM}(\overleftarrow{y}')$ include those operations. To satisfy the above security definition, it is implied that these operations also cannot leak any information about the access pattern.

3 Ring ORAM Protocol

3.1 Overview

We first describe Ring ORAM in terms of its server and client data structures. All notation used throughout the rest of the paper is summarized in Table 2.

Server storage is organized as a binary tree of buckets where each bucket has a small number of slots to hold blocks. Levels in the tree are numbered from 0 (the root) to L (inclusive, the leaves) where $L = O(\log N)$ and N is the number of blocks in the ORAM. Each bucket has $Z + S$ slots and a small amount of metadata. Of these slots, up to Z slots may contain real blocks and the remaining S slots are reserved for dummy blocks as described in Section 1.3. Our theoretical analysis in Section 4 will show that to store N blocks in Ring ORAM, the physical ORAM tree needs roughly $6N$ to $8N$ slots. Experiments show that server storage in practice for both Ring ORAM and Path ORAM can be $2N$ or even smaller.

Client storage is made up of a position map and a stash. The position map is a dictionary that maps each block in the ORAM to a random leaf in the ORAM tree (each leaf is given a unique identifier). The stash buffers blocks that have not been evicted to the ORAM tree and additionally stores $Z(L+1)$ blocks on the eviction path during an eviction operation. We will prove in Section 4 that stash overflow probability decreases exponentially as stash capacity increases, which means our required stash size is the same as Path ORAM. The position map stores $N * L$ bits, but can be squashed to constant storage using the standard recursion technique (Section 3.7).

Main invariants. Ring ORAM has two main invariants:

1. (Same as Path ORAM): Every block is mapped to a leaf chosen uniformly at random in the ORAM tree. If a block a is mapped to leaf l, block a is contained either in the stash or in some bucket along the path from the root of the tree to leaf l.
2. **(Permuted buckets)** For every bucket in the tree, the physical positions of the $Z+S$ dummy and real blocks in each bucket are randomly permuted with respect to all past and future writes to that bucket.

Since a leaf uniquely determines a path in a binary tree, we will use leaves/paths interchangeably when the context is clear, and denote path l as $\mathcal{P}(l)$.

Access and Eviction Operations. The Ring ORAM access protocol is shown in Algorithm 1. Each access is broken into the following four steps:

1.) Position Map lookup (Lines 3-5): Look up the position map to learn which path l the block being accessed is currently mapped to. Remap that block to a new random path l'.

This first step is identical to other tree-based ORAMs [23, 27]. But the rest of the protocol differs substantially from previous tree-based schemes, and we highlight our key innovations in **bold**.

2.) Read Path (Lines 6-15): The ReadPath(l,a) operation reads all buckets along $\mathcal{P}(l)$ to look for the block of interest (block a), and then reads that block into the stash. The block of interest is then updated in stash on a write, or is returned to the client on a read. We remind readers again that both reading and writing a data block are served by a ReadPath operation.

Unlike prior tree-based schemes, our ReadPath operation **only reads one block from each bucket—the block of interest if found or a previously-unread dummy block otherwise.** This is safe because of Invariant 2, above: each bucket is permuted randomly, so the slot being read looks random to an observer. This lowers the bandwidth overhead of ReadPath (i.e., online bandwidth) to $L+1$ blocks (the number of levels in the tree) or even a single block if the XOR trick is applied (Section 3.2).

Algorithm 1 Non-recursive Ring ORAM.

```
1:  function ACCESS($a$, op, data$'$)
2:      Global/persistent variables: round

3:      $l' \leftarrow$ UniformRandom$(0, 2^L - 1)$
4:      $l \leftarrow$ PositionMap$[a]$
5:      PositionMap$[a] \leftarrow l'$

6:      data $\leftarrow$ ReadPath$(l, a)$
7:      if data $= \perp$ then
8:          $\triangleright$ If block $a$ is not found on path $l$, it must
9:          be in Stash $\triangleleft$
10:         data $\leftarrow$ read and remove $a$ from Stash
11:     if op $=$ read then
12:         return data to client
13:     if op $=$ write then
14:         data $\leftarrow$ data$'$
15:     Stash $\leftarrow$ Stash $\cup (a, l', \text{data})$

16:     round $\leftarrow$ round $+ 1 \mod A$
17:     if round $\overset{?}{=} 0$ then
18:         EvictPath()

19:     EarlyReshuffle$(l)$
```

3.) Evict Path (Line 16-18): The EvictPath operation reads Z blocks (all the remaining real blocks, and potentially some dummy blocks) from each bucket along a path into the stash, and then fills that path with blocks from the stash, trying to push blocks as far down towards the leaves as possible. The sole purpose of an eviction operation is to push blocks back to the ORAM tree to keep the stash occupancy low.

Unlike Path ORAM, eviction in Ring ORAM **selects paths in the reverse lexicographical order, and does not happen on every access. Its rate is controlled by a public parameter A: every A ReadPath operations trigger a single EvictPath operation.** This means Ring ORAM needs much fewer eviction operations than Path ORAM. We will theoretically derive a tight relationship between A and Z in Section 4.

4.) Early Reshuffles (Line 19): Finally, we perform **a maintenance task called** EarlyReshuffle **on** $\mathcal{P}(l)$**, the path accessed by** ReadPath**.** This step is crucial in maintaining blocks randomly shuffled in each bucket, which enables ReadPath to securely read only one block from each bucket.

We will present details of ReadPath, EvictPath and EarlyReshuffle in the next three subsections. We defer low-level details for helper functions needed in these three subroutines to Appendix A. We explain the security for each subroutine in Section 3.5. Finally, we discuss additional optimizations in Section 3.6 and recursion in Section 3.7.

3.2 Read Path Operation

Algorithm 2 ReadPath procedure.

```
1: function ReadPath($l, a$)
2:     data ← $\perp$
3:     for $i \leftarrow 0$ to $L$ do
4:         offset ← GetBlockOffset($\mathcal{P}(l,i), a$)
5:         data′ ← $\mathcal{P}(l,i,$offset$)$
6:         Invalidate $\mathcal{P}(l,i,$offset$)$
7:         if data′ $\neq \perp$ then
8:             data ← data′
9:         $\mathcal{P}(l,i)$.count ← $\mathcal{P}(l,i)$.count $+ 1$
       return data
```

The ReadPath operation is shown in Algorithm 2. For each bucket along the current path, ReadPath selects a *single* block to read from that bucket. For a given bucket, if the block of interest lives in that bucket, we read and invalidate the block of interest. Otherwise, we read and invalidate a randomly-chosen dummy block that is still valid at that point. The index of the block to read (either real or random) is returned by the GetBlockOffset function whose detailed description is given in Appendix A.

Reading a single block per bucket is crucial for our bandwidth improvements. In addition to reducing online bandwidth by a factor of Z, it allows us to use larger Z and A to decrease overall bandwidth (Section 5). Without this, read bandwidth is proportional to Z, and the cost of larger Z on reads outweighs the benefits.

Bucket Metadata. Because the position map only tracks the *path* containing the block of interest, the client does not know where in each bucket to look for the block of interest. Thus, for each bucket we must store the permutation in the bucket metadata that maps each real block in the bucket to one of the $Z+S$ slots (Lines 4, GetBlockOffset) as well as some additional metadata. Once we know the offset into the bucket, Line 5 reads the block in the slot, and invalidates it. We describe all metadata in Appendix A, but make the important point that the metadata is small and independent of the block size.

One important piece of metadata to mention now is a counter which tracks how many times it has been read since its last eviction (Line 9). If a bucket is read too many (S) times, it may run out of dummy blocks (i.e., all the dummy blocks have been invalidated). On future accesses, if additional dummy blocks are requested from this bucket, we cannot re-read a previously invalidated dummy block: doing so reveals to the adversary that the block of interest is not in this bucket. Therefore, we need to reshuffle single buckets on-demand as soon as they are touched more than S times using EarlyReshuffle (Section 3.4).

XOR Technique. We further make the following key observation: during our ReadPath operation, each block returned to the client is a dummy block except for the block of interest. This means our scheme can also take advantage of the XOR technique introduced in [3] to reduce online bandwidth overhead to $O(1)$. To be more concrete, on each access ReadPath returns $L+1$ blocks in ciphertext, one from each bucket, $\mathsf{Enc}(b_0, r_0), \mathsf{Enc}(b_2, r_2), \cdots, \mathsf{Enc}(b_L, r_L)$. Enc is a randomized symmetric scheme such as AES counter mode with nonce r_i. With the XOR technique, ReadPath will return *a single ciphertext* — the ciphertext of all the blocks XORed together, namely $\mathsf{Enc}(b_0, r_0) \oplus \mathsf{Enc}(b_2, r_2) \oplus \cdots \oplus \mathsf{Enc}(b_L, r_L)$. The client can recover the encrypted block of interest by XORing the returned ciphertext with the encryptions of all the dummy blocks. To make computing each dummy block's encryption easy, the client can set the plaintext of all dummy blocks to a fixed value of its choosing (e.g., 0).

3.3 Evict Path Operation

Algorithm 3 EvictPath procedure.

```
function EvictPath
    Global/persistent variables $G$ initialized to 0
    $l \leftarrow G \mod 2^L$
    $G \leftarrow G + 1$
    for $i \leftarrow 0$ to $L$ do
        Stash ← Stash $\cup$ ReadBucket($\mathcal{P}(l,i)$)
    for $i \leftarrow L$ to $0$ do
        WriteBucket($\mathcal{P}(l,i)$, Stash)
        $\mathcal{P}(l,i)$.count ← 0
```

The EvictPath routine is shown in Algorithm 3. As mentioned, evictions are scheduled statically: one evic-

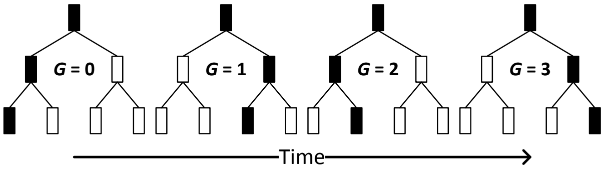


Figure 2: Reverse-lexicographic order of paths used by EvictPath. After path $G = 3$ is evicted to, the order repeats.

tion operation happens after every A reads. At a high level, an eviction operation reads all remaining real blocks on a path (in a secure fashion), and tries to push them down that path as far as possible. The leaf-to-root order in the writeback step (Lines 7) reflects that we wish to fill the deepest buckets as fully as possible. (For readers who are familiar with Path ORAM, EvictPath is like a Path ORAM access where no block is accessed and therefore no block is remapped to a new leaf.)

We emphasize two unique features of Ring ORAM eviction operations. First, evictions in Ring ORAM are performed to paths in a specific order called the *reverse-lexicographic order*, first proposed by Gentry et al. [9] and shown in Figure 2. The reverse-lexicographic order eviction aims to minimize the overlap between consecutive eviction paths, because (intuitively) evictions to the same bucket in consecutive accesses are less useful. This improves eviction quality and allows us to reduce the frequency of eviction. Evicting using this static order is also a key component in simplifying our theoretical analysis in Section 4.

Second, buckets in Ring ORAM need to be randomly shuffled (Invariant 2), and we mostly rely on EvictPath operations to keep them shuffled. An EvictPath operation reads Z blocks from each bucket on a path into the stash, and writes out $Z+S$ blocks (only up to Z are real blocks) to each bucket, randomly permuted. The details of reading/writing buckets (ReadBucket and WriteBucket) are deferred to Appendix A.

3.4 Early Reshuffle Operation

Algorithm 4 EarlyReshuffle procedure.

```
function EarlyReshuffle($l$)
    for $i \leftarrow 0$ to $L$ do
        if $\mathcal{P}(l,i)$.count $\geq S$ then
            Stash $\leftarrow$ Stash $\cup$ ReadBucket($\mathcal{P}(l,i)$)
            WriteBucket($\mathcal{P}(l,i)$, Stash)
            $\mathcal{P}(l,i)$.count $\leftarrow 0$
```

Due to randomness, a bucket can be touched $> S$ times by ReadPath operations before it is reshuffled by the scheduled EvictPath. If this happens, we call EarlyReshuffle on that bucket to reshuffle it before the bucket is read again (see Section 3.2). More precisely, after each ORAM access EarlyReshuffle goes over all the buckets on the read path, and reshuffles all the buckets that have been accessed more than S times by performing ReadBucket and WriteBucket. ReadBucket and WriteBucket are the same as in EvictPath: that is, ReadBucket reads exactly Z slots in the bucket and WriteBucket re-permutes and writes back $Z+S$ real/dummy blocks. We note that though S does not affect security (Section 3.5), it clearly has an impact on performance (how often we shuffle, the extra cost per reshuffle, etc.). We discuss how to optimally select S in Section 5.

3.5 Security Analysis

Claim 1. ReadPath *leaks no information.*

The path selected for reading will look random to any adversary due to Invariant 1 (leaves are chosen uniformly at random). From Invariant 2, we know that every bucket is randomly shuffled. Moreover, because we invalidate any block we read, we will never read the same slot. Thus, any sequence of reads (real or dummy) to a bucket between two shuffles is indistinguishable. Thus the adversary learns nothing during ReadPath. □

Claim 2. EvictPath *leaks no information.*

The path selected for eviction is chosen statically, and is public (reverse-lexicographic order). ReadBucket always reads exactly Z blocks from random slots. WriteBucket similarly writes $Z+S$ encrypted blocks in a data-independent fashion. □

Claim 3. EarlyShuffle *leaks no information.*

To which buckets EarlyShuffle operations occur is publicly known: the adversary knows how many times a bucket has been accessed since the last EvictPath to that bucket. ReadBucket and WriteBucket are secure as per observations in Claim 2. □

The three subroutines of the Ring ORAM algorithm are the only operations that cause externally observable behaviors. Claims 1, 2, and 3 show that the subroutines are secure. We have so far assumed that path remapping and bucket permutation are truly random, which gives unconditional security. If pseudorandom numbers are used instead, we have computational security through similar arguments.

3.6 Other Optimizations

Minimizing roundtrips. To keep the presentation simple, we wrote the ReadPath (EvictPath) algorithms to process buckets one by one. In fact, they can be performed for all buckets on the path in parallel which reduces the number of roundtrips to 2 (one for metadata and one for data blocks).

Tree-top caching. The idea of tree-top caching [18] is simple: we can reduce the bandwidth for ReadPath and EvictPath by storing the top t (a new parameter) levels of the Ring ORAM tree at the client as an extension of the stash[1]. For a given t, the stash grows by approximately $2^t Z$ blocks.

De-amortization. We can de-amortize the expensive EvictPath operation through a period of A accesses, simply by reading/writing a small number of blocks on the eviction path after each access. After de-amortization, worst-case overall bandwidth equals average overall bandwidth.

3.7 Recursive Construction

With the construction given thus far, the client needs to store a large position map. To achieve small client storage, we follow the standard recursion idea in tree-based ORAMs [23]: instead of storing the position map on the client, we store the position map on a smaller ORAM on the server, and store only the position map for the smaller ORAM. The client can recurse until the final position map becomes small enough to fit in its storage. For reasonably block sizes (e.g., 4 KB), recursion contributes very little to overall bandwidth (e.g., $< 5\%$ for a 1 TB ORAM) because the position map ORAMs use much smaller blocks [26]. Since recursion for Ring ORAM behaves in the same way as all the other tree-based ORAMs, we omit the details.

4 Stash Analysis

In this section we analyze the stash occupancy for a non-recursive Ring ORAM. Following the notations in Path ORAM [27], by $\mathsf{ORAM}_L^{Z,A}$ we denote a non-recursive Ring ORAM with $L+1$ levels, bucket size Z and one eviction per A accesses. The root is at level 0 and the leaves are at level L. We define the stash occupancy $\mathsf{st}(\mathcal{S}_Z)$ to be the number of real blocks in the stash after a sequence of ORAM sequences (this notation will be further explained later). We will prove that $\Pr[\mathsf{st}(\mathcal{S}_Z) > R]$ decreases exponentially in R for certain Z and A combinations. As it turns out, the deterministic eviction pattern in Ring ORAM dramatically simplifies the proof.

We note here that the reshuffling of a bucket does not affect the occupancy of the bucket, and is thus irrelevant to the proof we present here.

4.1 Proof outline

The proof consists of the two steps. The first step is the same as Path ORAM, and needs Lemma 1 and Lemma 2 in the Path ORAM paper [27], which we restate in Section 4.2. We introduce ∞-ORAM, which has an infinite bucket size and after a post-processing step has exactly the same distribution of blocks over all buckets and the stash (Lemma 1). Lemma 2 says the stash occupancy of ∞-ORAM after post-processing is greater than R if and only if there exists a subtree T in ∞-ORAM whose "occupancy" exceeds its "capacity" by more than R. We note, however, that the Path ORAM [27] paper only gave intuition for the proof of Lemma 1, and unfortunately did not capture of all the subtleties. We will rigorously prove that lemma, which turns out to be quite tricky and requires significant changes to the post-processing algorithm.

The second step (Section 4.3) is much simpler than the rest of Path ORAM's proof, thanks to Ring ORAM's static eviction pattern. We simply need to calculate the expected occupancy of subtrees in ∞-ORAM, and apply a Chernoff-like bound on their actual occupancy to complete the proof. We do not need the complicated eviction game, negative association, stochastic dominance, etc., as in the Path ORAM proof [26].

For readability, we will defer the proofs of all lemmas to Appendix B.

4.2 ∞-ORAM

We first introduce ∞-ORAM, denoted as $\mathsf{ORAM}_L^{\infty,A}$. Its buckets have infinite capacity. It receives the same input request sequence as $\mathsf{ORAM}_L^{Z,A}$. We then label buckets linearly such that the two children of bucket b_i are b_{2i} and b_{2i+1}, with the root bucket being b_1. We define the stash to be b_0. We refer to b_i of $\mathsf{ORAM}_L^{\infty,A}$ as b_i^∞, and b_i of $\mathsf{ORAM}_L^{Z,A}$ as b_i^Z. We further define ORAM *state*, which consists of the states of all the buckets in the ORAM, i.e., the blocks contained by each bucket. Let $\mathcal{S}_\infty$ be the state of $\mathsf{ORAM}_L^{\infty,A}$ and $\mathcal{S}_Z$ be the state of $\mathsf{ORAM}_L^{Z,A}$.

We now propose a new greedy post-processing algorithm G (different from the one in [27]), which by reassigning blocks in buckets makes each bucket b_i^∞ in ∞-ORAM contain the same set of blocks as b_i^Z. Formally, G takes as input $\mathcal{S}_\infty$ and $\mathcal{S}_Z$ after the same access sequence with the same randomness. For i from $2^{L+1}-1$ down to

[1] We call this optimization tree-top caching following prior work. But the word cache is a misnomer: the top t levels of the tree are *permanently* stored by the client.

1 (note that the decreasing order ensures that a parent is always processed later than its children), G processes the blocks in bucket b_i^∞ in the following way:

1. For those blocks that are also in b_i^Z, keep them in b_i^∞.
2. For those blocks that are not in b_i^Z but in some ancestors of b_i^Z, move them from b_i^∞ to $b_{i/2}^\infty$ (the parent of b_i^∞, and note that the division includes flooring). If such blocks exist and the number of blocks remaining in b_i^∞ is less than Z, raise an error.
3. If there exists a block in b_i^∞ that is in neither b_i^Z nor any ancestor of b_i^Z, raise an error.

We say $G_{\mathcal{S}_Z}(\mathcal{S}_\infty) = \mathcal{S}_Z$, if no error occurs during G and b_i^∞ after G contains the same set of blocks as b_i^Z for $i = 0, 1, \cdots 2^{L+1}$.

Lemma 1. *$G_{\mathcal{S}_Z}(\mathcal{S}_\infty) = \mathcal{S}_Z$ after the same ORAM access sequence with the same randomness.*

Next, we investigate what state $\mathcal{S}_\infty$ will lead to the stash occupancy of more than R blocks in a post-processed ∞-ORAM. We say a subtree T is a rooted subtree, denoted as $T \in \mathsf{ORAM}_L^{\infty,A}$ if T contains the root of $\mathsf{ORAM}_L^{\infty,A}$. This means that if a node in $\mathsf{ORAM}_L^{\infty,A}$ is in T, then so are all its ancestors. We define $n(T)$ to be the total number of nodes in T. We define $c(T)$ (the capacity of T) to be the maximum number of blocks T can hold; for Ring ORAM $c(T) = n(T) \cdot Z$. Lastly, we define $X(T)$ (the occupancy of T) to be the actual number of real blocks that are stored in T. The following lemma characterizes the stash size of a post-processed ∞-ORAM:

Lemma 2. *$\mathsf{st}(G_{\mathcal{S}_Z}(\mathcal{S}_\infty)) > R$ if and only if $\exists T \in \mathsf{ORAM}_L^{\infty,A}$ s.t. $X(T) > c(T) + R$ before post-processing.*

By Lemma 1 and Lemma 2, we have

$$\begin{aligned}
&\Pr[\mathsf{st}(\mathcal{S}_Z) > R] = \Pr[\mathsf{st}(G_{\mathcal{S}_Z}(\mathcal{S}_\infty)) > R] \\
&\le \sum_{T \in \mathsf{ORAM}_L^{\infty,A}} \Pr[X(T) > c(T) + R] \\
&< \sum_{n \ge 1} 4^n \max_{T : n(T) = n} \Pr[X(T) > c(T) + R]
\end{aligned} \tag{1}$$

The above inequalities used a union bound and a bound on Catalan sequences.

4.3 Bounding the Stash Size

We first give a bound on the expected bucket load:

Lemma 3. *For any rooted subtree T in $\mathsf{ORAM}_L^{\infty,A}$, if the number of distinct blocks in the ORAM $N \le A \cdot 2^{L-1}$, the expected load of T has the following upper bound:*

$$\forall T \in \mathsf{ORAM}_L^{\infty,A}, E[X(T)] \le n(T) \cdot A/2.$$

Let $X(T) = \sum_i X_i(T)$, where each $X_i(T) \in \{0, 1\}$ and indicates whether the i-th block (can be either real or stale) is in T. Let $p_i = \Pr[X_i(T) = 1]$. $X_i(T)$ is completely determined by its time stamp i and the leaf label assigned to block i, so they are independent from each other (refer to the proof of Lemma 3). Thus, we can apply a Chernoff-like bound to get an exponentially decreasing bound on the tail distribution. To do so, we first establish a bound on $E\left[e^{tX(T)}\right]$ where $t > 0$,

$$\begin{aligned}
E\left[e^{tX(T)}\right] &= E\left[e^{t\sum_i X_i(T)}\right] = E\left[\Pi_i e^{tX_i(T)}\right] \\
&= \Pi_i E\left[e^{tX_i(T)}\right] \qquad \text{(by independence)} \\
&= \Pi_i \left(p_i(e^t - 1) + 1\right) \\
&\le \Pi_i \left(e^{p_i(e^t-1)}\right) = e^{(e^t-1)\Sigma_i p_i} \\
&= e^{(e^t-1)E[X(T)]}
\end{aligned} \tag{2}$$

For simplicity, we write $n = n(T)$ and $a = A/2$. By Lemma 3, $E[X(T)] \le n \cdot a$. By the Markov Inequality, we have for all $t > 0$,

$$\begin{aligned}
\Pr[X(T) > c(T) + R] &= \Pr\left[e^{tX(T)} > e^{t(nZ+R)}\right] \\
&\le E\left[e^{tX(T)}\right] \cdot e^{-t(nZ+R)} \\
&\le e^{(e^t-1)an} \cdot e^{-t(nZ+R)} \\
&= e^{-tR} \cdot e^{-n[tZ - a(e^t-1)]}
\end{aligned}$$

Let $t = \ln(Z/a)$,

$$\Pr[X(T) > c(T) + R] \le (a/Z)^R \cdot e^{-n[Z\ln(Z/a) + a - Z]} \tag{3}$$

Now we will choose Z and A such that $Z > a$ and $q = Z\ln(Z/a) + a - Z - \ln 4 > 0$. If these two conditions hold, from Equation (1) we have $t = \ln(Z/a) > 0$ and that the stash overflow probability decreases exponentially in the stash size R:

$$\Pr[\mathsf{st}(\mathcal{S}_Z) > R] \le \sum_{n \ge 1} (a/Z)^R \cdot e^{-qn} < \frac{(a/Z)^R}{1 - e^{-q}}.$$

4.4 Stash Size in Practice

Now that we have established that $Z\ln(2Z/A) + A/2 - Z - \ln 4 > 0$ ensures an exponentially decreasing stash overflow probability, we would like to know how tight this requirement is and what the stash size should be in practice.

We simulate Ring ORAM with $L = 20$ for over 1 Billion accesses in a random access pattern, and measure the stash occupancy (excluding the transient storage of a path). For several Z values, we look for the maximum A that results in an exponentially decreasing stash overflow

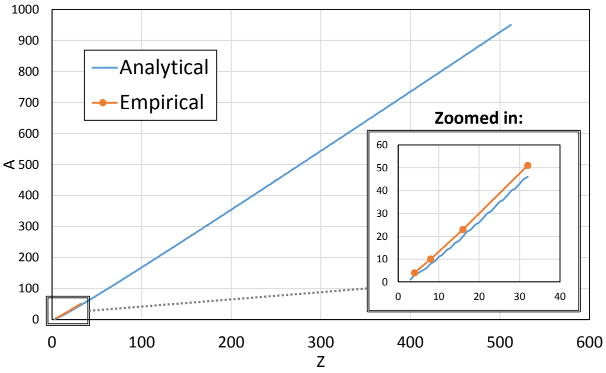


Figure 3: For each Z, determine analytically and empirically the maximum A that results in an exponentially decreasing stash failure probability.

		Z,A Parameters				
		4,3	8,8	16,20	32,46	16,23
		Max Stash Size				
λ	80	32	41	65	113	197
	128	51	62	93	155	302
	256	103	120	171	272	595

Table 3: Maximum stash occupancy for realistic security parameters (stash overflow probability $2^{-\lambda}$) and several choices of A and Z. $A = 23$ is the maximum achievable A for $Z = 16$ according to simulation.

probability. In Figure 3, we plot both the empirical curve based on simulation and the theoretical curve based on the proof. In all cases, the theoretical curve indicates a only slightly smaller A than we are able to achieve in simulation, indicating that our analysis is tight.

To determine required stash size in practice, Table 3 shows the extrapolated required stash size for a stash overflow probability of $2^{-\lambda}$ for several realistic λ. We show $Z = 16, A = 23$ for completeness: this is an aggressive setting that works for $Z = 16$ according to simulation but does not satisfy the theoretical analysis; observe that this point requires roughly 3× the stash occupancy for a given λ.

5 Bandwidth Analysis

In this section, we answer an important question: how do Z (the maximum number of real blocks per bucket), A (the eviction rate) and S (the number of extra dummies per bucket) impact Ring ORAM's performance (bandwidth)? By the end of the section, we will have a theoretically-backed analytic model that, given Z, selects optimal A and S to minimize bandwidth.

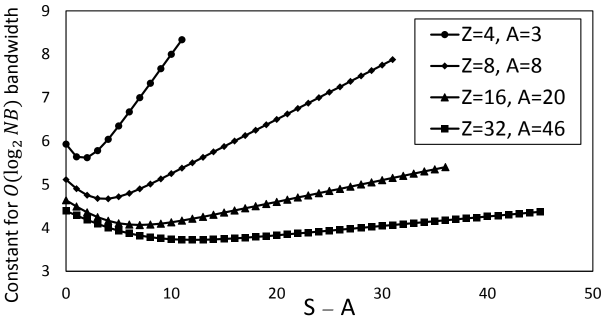


Figure 4: For different Z, and the corresponding optimal A, vary S and plot bandwidth overhead. We only consider $S \geq A$

We first state an intuitive trade-off: for a given Z, increasing A causes stash occupancy to increase and bandwidth overhead to decrease. Let us first ignore early reshuffles and the XOR technique. Then, the overall bandwidth of Ring ORAM consists of ReadPath and EvictPath. ReadPath transfers $L+1$ blocks, one from each bucket. EvictPath reads Z blocks per bucket and writes $Z+S$ blocks per bucket, $(2Z+S)(L+1)$ blocks in total, but happens every A accesses. From the requirement of Lemma 3, we have $L = \log(2N/A)$, so the ideal amortized overall bandwidth of Ring ORAM is $(1+(2Z+S)/A)\log(4N/A)$. Clearly, a larger A improves bandwidth for a given Z as it reduces both eviction frequency and tree depth L. So we simply choose the largest A that satisfies the requirement from the stash analysis in Section 4.3.

Now we consider the extra overhead from early reshuffles. We have the following trade-off in choosing S: as S increases, the early reshuffle rate decreases (since we have more dummies per bucket) but the cost to read+write buckets during an EvictPath and EarlyReshuffle increases. This effect is shown in Figure 4 through simulation: for S too small, early shuffle rate is high and bandwidth increases; for S too large, eviction bandwidth dominates.

To analytically choose a good S, we analyze the early reshuffle rate. First, notice a bucket at level l in the Ring ORAM tree will be processed by EvictPath *exactly* once for every $2^l A$ ReadPath operations, due to the reverse-lexicographic order of eviction paths (Section 3.3). Second, each ReadPath operation is to an independent and uniformly random path and thus will touch any bucket in level l with equal probability of 2^{-l}. Thus, the distribution on the expected number of times ReadPath operations touch a given bucket in level l, between two consecutive EvictPath calls, is given by a binomial distribution of $2^l A$ trials and success probability 2^{-l}. The probability that a bucket needs to be early reshuffled before an EvictPath is given by a binomial distribution cumula-

Find largest $A \leq 2Z$ such that
$Z\ln(2Z/A) + A/2 - Z - \ln 4 > 0$ holds.
Find $S \geq 0$ that minimizes
$(2Z+S)(1+\mathsf{Poiss\_cdf}(S,A))$
Ring ORAM offline bandwidth is
$\frac{(2Z+S)(1+\mathsf{Poiss\_cdf}(S,A))}{A} \cdot \log(4N/A)$

Table 4: Analytic model for choosing parameters, given Z.

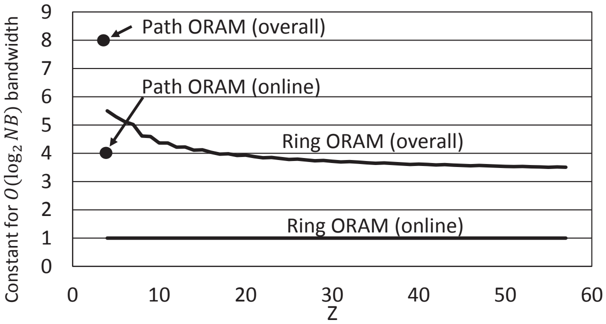


Figure 5: Overall bandwidth as a function of Z. Kinks are present in the graph because we always round A to the nearest integer. For Path ORAM, we only study $Z = 4$ since a larger Z strictly hurts bandwidth.

tive density function $\mathsf{Binom\_cdf}(S, 2^l A, 2^{-l})$.[2] Based on this analysis, the expected number of times any bucket is involved in ReadPath operations between consecutive EvictPath operations is A. Thus, we will only consider $S \geq A$ as shown in Figure 4 ($S < A$ is clearly bad as it needs too much early reshuffling).

We remark that the binomial distribution quickly converges to a Poisson distribution. So the amortized overall bandwidth, taking early reshuffles into account, can be accurately approximated as $(L+1) + (L+1)(2Z+S)/A \cdot (1 + \mathsf{Poiss\_cdf}(S,A))$. We should then choose the S that minimizes the above formula. This method always finds the optimal S and perfectly matches the overall bandwidth in our simulation in Figure 4.

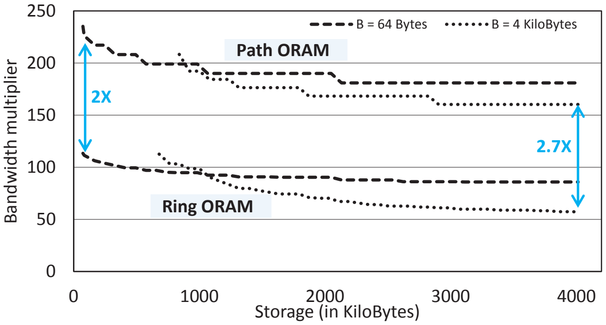


Figure 6: Bandwidth overhead vs. data block storage for 1 TB ORAM capacities and ORAM failure probability 2^{-80}.

We recap how to choose A and S for a given Z in Table 4. For the rest of the paper, we will choose A and S this way unless otherwise stated. Using this method to set A and S, we show online and overall bandwidth as a function of Z in Figure 5. In the figure, Ring ORAM does not use the XOR technique on reads. For $Z = 50$, we achieve $\sim 3.5 \log N$ bandwidth; for very large Z, bandwidth approaches $3 \log N$. Applying the XOR technique, online bandwidth overhead drops to close to 1 which reduces overall bandwidth to $\sim 2.5 \log N$ for $Z = 50$ and $2 \log N$ for very large Z.

6 Evaluation

6.1 Bandwidth vs. Client Storage

To give a holistic comparison between schemes, Figure 6 shows the best achievable bandwidth, for different client storage budgets, for Path ORAM and Ring ORAM. For each scheme in the figure, we apply all known optimizations and tune parameters to minimize overall bandwidth given a storage budget. For Path ORAM we choose $Z = 4$ (increasing Z strictly hurts bandwidth) and tree-top cache to fill remaining space. For Ring ORAM we adjust Z, A and S, tree-top cache and apply the XOR technique.

To simplify the presentation, "client storage" includes all ORAM data structures except for the position map – which has the same space/bandwidth cost for both Path ORAM and Ring ORAM. We remark that applying the recursion technique (Section 3.7) to get a small on-chip position map is cheap for reasonably large blocks. For example, recursing the on-chip position map down to 256 KiloBytes of space when the data block size is 4 KiloBytes increases overall bandwidth for Ring ORAM and Path ORAM by $< 3\%$.

The high order bit is that across different block sizes and client storage budgets, Ring ORAM consistently reduces overall bandwidth relative to Path ORAM by 2-2.7×. We give a summary of these results for several representative client storage budgets in Table 5. We remark that for smaller block sizes, Ring ORAM's improvement over Path ORAM ($\sim 2\times$ for 64 Byte blocks) is smaller relative to when we use larger blocks ($2.7\times$ for 4 KiloByte blocks). The reason is that with small blocks, the cost to read bucket metadata cannot be ignored, forcing Ring ORAM to use smaller Z.

[2] The possibility that a bucket needs to be early reshuffled twice before an eviction is negligible.

		Online, Overall Bandwidth overhead		
Block Size (Bytes)	Z, A (Ring ORAM only)	**Ring ORAM**	**Ring ORAM (XOR)**	Path ORAM
64	$10, 11$	$48\times, 144\times$	$24\times, 118\times$	$120\times, 240\times$
4096	$33, 48$	$20\times, 82\times$	$\sim 1\times, 60\times$	$80\times, 160\times$

Table 5: Breakdown between online and offline bandwidth given a client storage budget of 1000× the block size for several representative points (Section 6.1). Overheads are relative to an insecure system. Parameter meaning is given in Table 2.

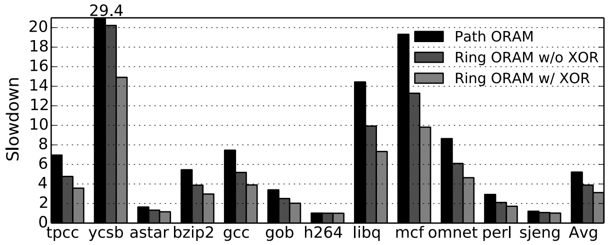


Figure 7: SPEC benchmark slowdown.

6.2 Case Study: Secure Processors

In this study, we show how Ring ORAM improves the performance of secure processors over Path ORAM. We assume the same processor/cache architecture as [5], given in Table 4 of that work. We evaluate a 4 GigaByte ORAM with 64-Byte block size (matching a typical processor's cache line size). Due to the small block size, we parameterize Ring ORAM at $Z = 5, A = 5, X = 2$ to reduce metadata overhead. We use the optimized ORAM recursion techniques [22]: we apply recursion three times with 32-Byte position map block size and get a 256 KB final position map. We evaluate performance for SPEC-int benchmarks and two database benchmarks, and simulate 3 billion instructions for each benchmark. We assume a flat 50-cycle DRAM latency, and compute ORAM latency assuming 128 bits/cycle processor-memory bandwidth. We do not use tree-top caching since it proportionally benefits both Ring ORAM and Path ORAM. Today's DRAM DIMMs cannot perform any computation, but it is not hard to imagine having simple XOR logic either inside memory, or connected to $O(\log N)$ parallel DIMMs so as not to occupy processor-memory bandwidth. Thus, we show results with and without the XOR technique.

Figure 7 shows program slowdown over an insecure DRAM. The high order bit is that using Ring ORAM with XOR results in a geometric average slowdown of 2.8× relative to an insecure system. This is a 1.5× improvement over Path ORAM. If XOR is not available, the slowdown over an insecure system is 3.2×.

We have also repeated the experiment with the unified ORAM recursion technique and its parameters [5]. The geometric average slowdown over an insecure system is 2.4× (2.5× without XOR).

7 Ring ORAM with Large Client Storage

If given a large client storage budget, we can first choose very large A and Z for Ring ORAM, which means bandwidth approaches $2\log N$ (Section 5).[3] Then remaining client storage can be used to tree-top cache (Section 3.6). For example, tree-top caching $t = L/2$ levels requires $O(\sqrt{N})$ storage and bandwidth drops by a factor of 2 to $1 \cdot \log N$—which roughly matches the SSS construction [25].

Burst ORAM [3] extends the SSS construction to handle millions of accesses in a short period, followed by a relatively long idle time where there are few requests. The idea to adapt Ring ORAM to handle bursts is to delay multiple (potentially millions of) EvictPath operations until after the burst of requests. Unfortunately, this strategy means we will experience a much higher early reshuffle rate in levels towards the root. The solution is to coordinate tree-top caching with delayed evictions: For a given tree-top size t, we allow at most 2^t delayed EvictPath operations. This ensures that for levels $\geq t$, the early reshuffle rate matches our analysis in Section 5. We experimentally compared this methodology to the dataset used by Burst ORAM and verified that it gives comparable performance to that work.

8 Related Work

ORAM was first proposed by Goldreich and Ostrovsky [10, 11]. Since then, there have been numerous follow-up works that significantly improved ORAM's efficiency in the past three decades [21, 20, 2, 1, 29, 12, 13, 15, 25, 23, 9, 27, 28]. We have already reviewed two state-of-the-art schemes with different client storage requirements: Path ORAM [27] and the SSS ORAM [25]. Circuit ORAM [28] is another recent tree-based ORAM, which requires only $O(1)$ client storage, but its bandwidth is a constant factor worse than Path ORAM.

Reducing online bandwidth. Two recent works [3, 19] have made efforts to reduce online bandwidth (response time). Unfortunately, the techniques in Burst ORAM [3] do not work with Path ORAM (or more generally any existing tree-based ORAMs). On the

[3] We assume the XOR technique because large client storage implies a file server setting.

other hand, Path-PIR [19], while featuring a tree-based ORAM, employs heavy primitives like Private Information Retrieval (PIR) or even FHE, and thus requires a significant amount of server computation. In comparison, our techniques efficiently achieve $O(1)$ online cost for tree-based ORAMs without resorting to PIR/FHE, and also improve bursty workload performance similar to Burst ORAM.

Subsequent work. Techniques proposed in this paper have been adopted by subsequent works. For example, Tiny ORAM [6] and Onion ORAM [4] used part of our eviction strategy in their design for different purposes.

9 Conclusion

This paper proposes Ring ORAM, the most bandwidth-efficient ORAM scheme for the small (constant or polylog) client storage setting. Ring ORAM is simple, flexible and backed by a tight theoretic analysis.

Ring ORAM is the first tree-based ORAM whose online and overall bandwidth are independent of tree ORAM bucket size. With this and additional properties of the algorithm, we show that Ring ORAM improves online bandwidth by 60× (if simple computation such as XOR is available at memory), and overall bandwidth by 2.3× to 4× relative to Path ORAM. In a secure processor case study, we show that Ring ORAM's bandwidth improvement translates to an overall program performance improvement of 1.5×. By increasing Ring ORAM's client storage, Ring ORAM is competitive in the cloud storage setting as well.

Acknowledgement

This research was partially by NSF grant CNS-1413996 and CNS-1314857, the QCRI-CSAIL partnership, a Sloan Fellowship, and Google Research Awards. Christopher Fletcher was supported by a DoD National Defense Science and Engineering Graduate Fellowship.

References

[1] Boneh, D., Mazieres, D., and Popa, R. A. Remote oblivious storage: Making oblivious RAM practical. Manuscript, http://dspace.mit.edu/bitstream/handle/1721.1/62006/MIT-CSAIL-TR-2011-018.pdf, 2011.

[2] Damgård, I., Meldgaard, S., and Nielsen, J. B. Perfectly secure oblivious RAM without random oracles. In *TCC* (2011).

[3] Dautrich, J., Stefanov, E., and Shi, E. Burst oram: Minimizing oram response times for bursty access patterns. In *USENIX* (2014).

[4] Devadas, S., van Dijk, M., Fletcher, C. W., Ren, L., Shi, E., and Wichs, D. Onion oram: A constant bandwidth blowup oblivious ram. Cryptology ePrint Archive, 2015. http://eprint.iacr.org/2015/005.

[5] Fletcher, C., Ren, L., Kwon, A., van Dijk, M., and Devadas, S. Freecursive oram: [nearly] free recursion and integrity verification for position-based oblivious ram. In *ASPLOS* (2015).

[6] Fletcher, C., Ren, L., Kwon, A., Van Dijk, M., Stefanov, E., Serpanos, D., and Devadas, S. A low-latency, low-area hardware oblivious ram controller. In *FCCM* (2015).

[7] Fletcher, C., Ren, L., Yu, X., Van Dijk, M., Khan, O., and Devadas, S. Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs. In *HPCA* (2014).

[8] Fletcher, C., van Dijk, M., and Devadas, S. Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In *STC* (2012).

[9] Gentry, C., Goldman, K. A., Halevi, S., Jutla, C. S., Raykova, M., and Wichs, D. Optimizing oram and using it efficiently for secure computation. In *PET* (2013).

[10] Goldreich, O. Towards a theory of software protection and simulation on oblivious rams. In *STOC* (1987).

[11] Goldreich, O., and Ostrovsky, R. Software protection and simulation on oblivious rams. In *J. ACM* (1996).

[12] Goodrich, M. T., and Mitzenmacher, M. Privacy-preserving access of outsourced data via oblivious ram simulation. In *ICALP* (2011).

[13] Goodrich, M. T., Mitzenmacher, M., Ohrimenko, O., and Tamassia, R. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA* (2012).

[14] Islam, M., Kuzu, M., and Kantarcioglu, M. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS* (2012).

[15] Kushilevitz, E., Lu, S., and Ostrovsky, R. On the (in) security of hash-based oblivious ram and a new balancing scheme. In *SODA* (2012).

[16] Liu, C., Harris, A., Maas, M., Hicks, M., Tiwari, M., and Shi, E. Ghostrider: A hardware-software system for memory trace oblivious computation. In *ASPLOS* (2015).

[17] Lorch, J. R., Parno, B., Mickens, J. W., Raykova, M., and Schiffman, J. Shroud: Ensuring private access to large-scale data in the data center. In *FAST* (2013).

[18] Maas, M., Love, E., Stefanov, E., Tiwari, M., Shi, E., Asanovic, K., Kubiatowicz, J., and Song, D. Phantom: Practical oblivious computation in a secure processor. In *CCS* (2013).

[19] Mayberry, T., Blass, E.-O., and Chan, A. H. Efficient private file retrieval by combining oram and pir. In *NDSS* (2014).

[20] OSTROVSKY, R. Efficient computation on oblivious rams. In *STOC* (1990).

[21] OSTROVSKY, R., AND SHOUP, V. Private information storage (extended abstract). In *STOC* (1997).

[22] REN, L., YU, X., FLETCHER, C., VAN DIJK, M., AND DEVADAS, S. Design space exploration and optimization of path oblivious ram in secure processors. In *ISCA* (2013).

[23] SHI, E., CHAN, T.-H. H., STEFANOV, E., AND LI, M. Oblivious ram with $o((\log n)^3)$ worst-case cost. In *Asiacrypt* (2011).

[24] STEFANOV, E., AND SHI, E. Oblivistore: High performance oblivious cloud storage. In *S&P* (2013).

[25] STEFANOV, E., SHI, E., AND SONG, D. Towards practical oblivious RAM. In *NDSS* (2012).

[26] STEFANOV, E., VAN DIJK, M., SHI, E., CHAN, T.-H. H., FLETCHER, C., REN, L., YU, X., AND DEVADAS, S. Path oram: An extremely simple oblivious ram protocol. Cryptology ePrint Archive, 2013. `http://eprint.iacr.org/2013/280`.

[27] STEFANOV, E., VAN DIJK, M., SHI, E., FLETCHER, C., REN, L., YU, X., AND DEVADAS, S. Path oram: An extremely simple oblivious ram protocol. In *CCS* (2013).

[28] WANG, X. S., CHAN, T.-H. H., AND SHI, E. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. Cryptology ePrint Archive, 2014. `http://eprint.iacr.org/2014/672`.

[29] WILLIAMS, P., AND SION, R. Single round access privacy on outsourced storage. In *CCS* (2012).

[30] WILLIAMS, P., SION, R., AND TOMESCU, A. Privatefs: A parallel oblivious file system. In *CCS* (2012).

[31] YU, X., FLETCHER, C. W., REN, L., VAN DIJK, M., AND DEVADAS, S. Generalized external interaction with tamper-resistant hardware with bounded information leakage. In *CCSW* (2013).

[32] ZHUANG, X., ZHANG, T., AND PANDE, S. HIDE: an infrastructure for efficiently protecting information leakage on the address bus. In *ASPLOS* (2004).

A Bucket Structure

Table 6 lists all the fields in a Ring ORAM bucket and their size. We would like to make two remarks. First, only the data fields are permuted and that permutation is stored in ptrs. Other bucket fields do not need to be permuted because when they are needed, they will be read in their entirety. Second, count and valids are stored in plaintext. There is no need to encrypt them since the server can see which bucket is accessed (deducing count for each bucket), and which slot is accessed in each bucket (deducing valids for each bucket). In fact, if the server can do computation and is trusted to follow the protocol faithfully, the client can let the server update count and valids. All the other structures should be probabilistically encrypted.

Algorithm 5 Helper functions.

```
count, valids, addrs, leaves, ptrs, data are fields of the
input bucket in each of the following three functions

function GetBlockOffset(bucket, $a$)
    read in valids, addrs, ptrs
    decrypt addrs, ptrs
    for $j \leftarrow 0$ to $Z-1$ do
        if $a =$ addrs[$j$] and valids[ptrs[$j$]] then
            return ptrs[$j$]                ▷ block of interest
    return a pointer to a random valid dummy

function ReadBucket(bucket)
    read in valids, addrs, leaves, ptrs
    decrypt addrs, leaves, ptrs
    $z \leftarrow 0$            ▷ track # of remaining real blocks
    for $j \leftarrow 0$ to $Z-1$ do
        if valids[ptrs[$j$]] then
            data$'$ $\leftarrow$ read and decrypt data[ptrs[$j$]]
            $z \leftarrow z+1$
            if addrs[$j$] $\neq \perp$ then
                block $\leftarrow$ (addr[$j$], leaf[$j$], data$'$)
                Stash $\leftarrow$ Stash $\cup$ block
    for $j \leftarrow z$ to $Z-1$ do
        read a random valid dummy

function WriteBucket(bucket, Stash)
    find up to $Z$ blocks from Stash that can reside
    in this bucket, to form addrs, leaves, data$'$
    ptrs $\leftarrow$ PRP($0, Z+S$)              ▷ or truly random
    for $j \leftarrow 0$ to $Z-1$ do
        data[ptrs[$j$]] $\leftarrow$ data$'$[$j$]
    valids $\leftarrow \{1\}^{Z+S}$
    count $\leftarrow 0$
    encrypt addrs, leaves, ptrs, data
    write out count, valids, addrs, leaves, ptrs, data
```

Having defined the bucket structure, we can be more specific about some of the operations in earlier sections. For example, in Algorithm 2 Line 5 means reading $\mathcal{P}(l,i)$.data[offset], and Line 6 means setting $\mathcal{P}(l,i)$.valids[offset] to 0.

Now we describe the helper functions in detail. GetBlockOffset reads in the valids, addrs, ptrs field, and looks for the block of interest. If it finds the block of interest, meaning that the address of a still valid block matches the block of interest, it returns the permuted location of that block (stored in ptrs). If it does not find the block of interest, it returns the permuted location of a random valid dummy block.

Notation	Size (bits)	Meaning
count	$\log(S)$	# of times this bucket has been touched by ReadPath since it was last shuffled
valids	$(Z+S)*1$	Indicates whether each of the $Z+S$ blocks is valid
addrs	$Z*\log(N)$	Address for each of the Z (potentially) real blocks
leaves	$Z*L$	Leaf label for each of the Z (potentially) real blocks
ptrs	$Z*\log(Z+S)$	Offset in the bucket for each of the Z (potentially) real blocks
data	$(Z+S)*B$	Data field for each of the $Z+S$ blocks, **permuted** according to ptrs
EncSeed	λ (security parameter)	Encryption seed for the bucket; count and valids are stored in the clear

Table 6: Ring ORAM bucket format. All logs are taken to their ceiling.

ReadBucket reads all of the remaining real blocks in a bucket into the stash. For security reasons, ReadBucket always reads *exactly* Z blocks from that bucket. If the bucket contains less than Z valid real blocks, the remaining blocks read out are random valid dummy blocks. Importantly, since we allow at most S reads to each bucket before reshuffling it, it is guaranteed that there are at least Z valid (real + dummy) blocks left that have not been touched since the last reshuffle.

WriteBucket evicts as many blocks as possible (up to Z) from the stash to a certain bucket. If there are $z' \leq Z$ real blocks to be evicted to that bucket, $Z+S-z'$ dummy blocks are added. The $Z+S$ blocks are then randomly shuffled based on either a truly random permutation or a Pseudo Random Permutation (PRP). The permutation is stored in the bucket field ptrs. Then, the function resets count to 0 and all valid bits to 1, since this bucket has just been reshuffled and no blocks have been touched. Finally, the permuted data field along with its metadata are encrypted (except count and valids) and written out to the bucket.

B Proof of the Lemmas

To prove Lemma 1, we made a little change to the Ring ORAM algorithm. In Ring ORAM, a ReadPath operation adds the block of interest to the stash and replaces it with a dummy block in the tree. Instead of making the block of interest in the tree dummy, we turn it into a *stale* block. On an EvictPath operation to path l, all the stale blocks that are mapped to leaf l are turned into dummy blocks. Stale blocks are treated as real blocks in both $\mathsf{ORAM}_L^{Z,A}$ and $\mathsf{ORAM}_L^{\infty,A}$ (including G_Z) until they are turned into dummy blocks. Note that this trick of stale blocks is only to make the proof go through. It hurts the stash occupancy and we will not use it in practice. With the stale block trick, we can use induction to prove Lemma 1.

Proof of Lemma 1. Initially, the lemma obviously holds. Suppose $G_{\mathcal{S}_Z}(\mathcal{S}_\infty) = \mathcal{S}_Z$ after some accesses. We need to show that $G_{\mathcal{S}'_Z}(\mathcal{S}_\infty) = \mathcal{S}'_Z$ where $\mathcal{S}'_Z$ and $\mathcal{S}'_\infty$ are the states after the next operation (either ReadPath or EvictPath). A ReadPath operation adds a block to the stash (the root bucket) for both $\mathsf{ORAM}_L^{Z,A}$ and $\mathsf{ORAM}_L^{\infty,A}$, and does not move any blocks in the tree except turning a real block into a stale block. Since stale blocks are treated as real blocks, $G_{\mathcal{S}'_Z}(\mathcal{S}_\infty) = \mathcal{S}'_Z$ holds.

Now we show the induction holds for an EvictPath operation. Let EP_l^Z be an EvictPath operation to $\mathcal{P}(l)$ (path l) in $\mathsf{ORAM}_L^{Z,A}$ and EP_l^∞ be an EvictPath operation to $\mathcal{P}(l)$ in $\mathsf{ORAM}_L^{\infty,A}$. Then, $\mathcal{S}'_Z = \mathsf{EP}_l^Z(\mathcal{S}_Z)$ and $\mathcal{S}'_\infty = \mathsf{EP}_l^\infty(\mathcal{S}_\infty)$. Note that EP_l^Z has the same effect as EP_l^∞ followed by post-processing, so

$$\begin{aligned}\mathcal{S}'_Z = \mathsf{EP}_l^Z(\mathcal{S}_Z) &= G_{\mathcal{S}'_Z}(\mathsf{EP}_l^\infty(\mathcal{S}_Z)) \\ &= G_{\mathcal{S}'_Z}(\mathsf{EP}_l^\infty(G_{\mathcal{S}_Z}(\mathcal{S}_\infty)))\end{aligned}$$

The last equation is due to the induction hypothesis.

It remains to show that

$$G_{\mathcal{S}'_Z}(\mathsf{EP}_l^\infty(G_{\mathcal{S}_Z}(\mathcal{S}_\infty))) = G_{\mathcal{S}'_Z}(\mathsf{EP}_l^\infty(\mathcal{S}_\infty)),$$

which is $G_{\mathcal{S}'_Z}(\mathcal{S}'_\infty)$. To show this, we decompose G into steps for each bucket, i.e., $G_{\mathcal{S}_Z}(\mathcal{S}_\infty) = g_1 g_2 \cdots g_{2^{L+1}}(\mathcal{S}_\infty)$ where g_i processes bucket b_i^∞ in reference to b_i^Z. Similarly, we decompose $G_{\mathcal{S}'_Z}$ into $g'_1 g'_2 \cdots g'_{2^{L+1}}$ where each g'_i processes bucket $b_i'^\infty$ of $\mathcal{S}'_\infty$ in reference to $b_i'^Z$ of $\mathcal{S}'_Z$. We now only need to show that for any $0 < i < 2^{L+1}$, $G_{\mathcal{S}'_Z}(\mathsf{EP}_l^\infty(g_1 g_2 \cdots g_i(\mathcal{S}_\infty))) = G_{\mathcal{S}'_Z}(\mathsf{EP}_l^\infty(g_1 g_2 \cdots g_{i-1}(\mathcal{S}_\infty)))$. This is obvious if we consider the following three cases separately:

1. If $b_i \in \mathcal{P}(l)$, then g_i before EP_l^∞ has no effect since EP_l^∞ moves all blocks on $\mathcal{P}(l)$ into the stash before evicting them to $\mathcal{P}(l)$.
2. If $b_i \notin \mathcal{P}(l)$ and $b_{i/2} \notin \mathcal{P}(l)$ (neither b_i nor its parent is on Path l), then g_i and EP_l^∞ touch non-overlapping buckets and do not interfere with each other. Hence, their order can be swapped, $G_{\mathcal{S}'_Z}(\mathsf{EP}_l^\infty(g_0 g_1 g_2 \cdots g_i(\mathcal{S}_\infty))) = G_{\mathcal{S}'_Z} g_i(\mathsf{EP}_l^\infty(g_0 g_1 g_2 \cdots g_{i-1}(\mathcal{S}_\infty)))$. Furthermore,

$b_i^Z = b_i'^Z$ (since EP_l^∞ does not change the content of b_i), so g_i has the same effect as g_i' and can be merged into $G_{\mathcal{S}_Z'}$.

3. If $b_i \notin \mathcal{P}(l)$ but $b_{i/2} \in \mathcal{P}(l)$, the blocks moved into $b_{i/2}$ by g_i will stay in $b_{i/2}$ after EP_l^∞ since $b_{i/2}$ is the highest intersection (towards the leaf) that these blocks can go to. So g_i can be swapped with EP_l^∞ and can be merged into $G_{\mathcal{S}_Z'}$ as in the second case.

We remind the readers that because we only remove stale blocks that are mapped to $\mathcal{P}(l)$, the first case is the only case where some stale blocks in b_i may turn into dummy blocks. And the same set of stale blocks are removed from $\mathsf{ORAM}_L^{Z,A}$ and $\mathsf{ORAM}_L^{\infty,A}$.

This shows

$$\begin{aligned} G_{\mathcal{S}_Z'}\left(\mathsf{EP}_l^\infty\left(G_{\mathcal{S}_Z}\left(\mathcal{S}_\infty\right)\right)\right) &= G_{\mathcal{S}_Z'}\left(\mathsf{EP}_l^\infty\left(\mathcal{S}_\infty\right)\right) \\ &= G_{\mathcal{S}_Z'}\left(\mathcal{S}_\infty'\right) \end{aligned}$$

and completes the proof. □

The proof of Lemma 2 remains unchanged from the Path ORAM paper [27], and is replicated here for completeness.

Proof of Lemma 2. If part: Suppose $T \in \mathsf{ORAM}_L^{\infty,A}$ and $X(T) > c(T) + R$. Observe that G can assign the blocks in a bucket only to an ancestor bucket. Since T can store at most $c(T)$ blocks, more than R blocks must be assigned to the stash by G.

Only if part: Suppose that $\mathsf{st}\left(G_{\mathcal{S}_Z}\left(\mathcal{S}_\infty\right)\right) > R$. Let T be the maximal rooted subtree such that all the buckets in T contain exactly Z blocks after post-processing G. Suppose b is a bucket not in T. By the maximality of T, there is an ancestor (not necessarily proper ancestor) bucket b' of b that contains less than Z blocks after post-processing, which implies that no block from b can go to the stash. Hence, all blocks that are in the stash must have originated from T. Therefore, it follows that $X(T) > c(T) + R$. □

Proof of Lemma 3. For a bucket b in $\mathsf{ORAM}_L^{\infty,A}$, define $Y(b)$ to be the number of blocks in b before post-processing. It suffices to prove that $\forall b \in \mathsf{ORAM}_L^{\infty,A}$, $E[Y(b)] \leq A/2$.

If b is a leaf bucket, the blocks in it are put there by the last EvictPath operation to that leaf/path. Note that only real blocks could be put in b by that operation, although some of them may have turned into stale blocks. Stale blocks can never be moved into a leaf by an EvictPath operation, because that EvictPath operation would remove all the stale blocks mapped to that leaf. There are at most N distinct real blocks and each block has a probability of 2^{-L} to be mapped to b independently. Thus $E[Y(b)] \leq N \cdot 2^{-L} \leq A/2$.

If b is not a leaf bucket, we define two variables m_1 and m_2: the last EvictPath operation to b's left child is the m_1-th EvictPath operation, and the last EvictPath operation to b's right child is the m_2-th EvictPath operation. Without loss of generality, assume $m_1 < m_2$. We then time-stamp the blocks as follows. When a block is accessed and remapped, it gets time stamp m^*, which is the number of EvictPath operations that have happened. Blocks with $m^* \leq m_1$ will not be in b as they will go to either the left child or the right child of b. Blocks with $m^* > m_2$ will not be in b as the last access to b (m_2-th) has already passed. Therefore, only blocks with time stamp $m_1 < m^* \leq m_2$ will be put in b by the m_2-th access. (Some of them may be accessed again after the m_2-th access and become stale, but this does not affect the total number of blocks in b as stale blocks are treated as real blocks.) There are at most $d = A|m_1 - m_2|$ such blocks, and each goes to b independently with a probability of $2^{-(i+1)}$, where i is the level of b. The deterministic nature of evictions in Ring ORAM ensures $|m_1 - m_2| = 2^i$. (One way to see this is that a bucket b at level i will be written every 2^i EvictPath operations, and two consecutive EvictPath operations to b always travel down the two different children of b.) Therefore, $E[Y(b)] \leq d \cdot 2^{-(i+1)} = A/2$ for any non-leaf bucket as well. □

Raccoon: Closing Digital Side-Channels through Obfuscated Execution

Ashay Rane, Calvin Lin
Department of Computer Science,
The University of Texas at Austin
$\{$*ashay,lin*$\}$ *@cs.utexas.edu*

Mohit Tiwari
Dept. of Electrical and Computer Engineering
The University of Texas at Austin
tiwari@austin.utexas.edu

Abstract

Side-channel attacks monitor some aspect of a computer system's behavior to infer the values of secret data. Numerous side-channels have been exploited, including those that monitor caches, the branch predictor, and the memory address bus. This paper presents a method of defending against a broad class of side-channel attacks, which we refer to as *digital side-channel attacks*. The key idea is to obfuscate the program at the source code level to provide the illusion that many extraneous program paths are executed. This paper describes the technical issues involved in using this idea to provide confidentiality while minimizing execution overhead. We argue about the correctness and security of our compiler transformations and demonstrate that our transformations are safe in the context of a modern processor. Our empirical evaluation shows that our solution is 8.9× faster than prior work (GhostRider [20]) that specifically defends against memory trace-based side-channel attacks.

1 Introduction

It is difficult to keep secrets during program execution. Even with powerful encryption, the values of secret variables can be inferred through various side-channels, which are mechanisms for observing the program's execution at the level of the operating system, the instruction set architecture, or the physical hardware. Side-channel attacks have been used to break AES [26] and RSA [27] encryption schemes, to break the Diffie-Hellman key exchange [15], to fingerprint software libraries [46], and to reverse-engineer commercial processors [18].

```
function SQUARE_AND_MULTIPLY($m, s, n$)
    $z \leftarrow 1$
    for bit $b$ in $s$ from left to right do
        if $b = 1$ then
            $z \leftarrow m \cdot z^2 \bmod n$
        else
            $z \leftarrow z^2 \bmod n$
        end if
    end for
return $z$
end function
```

Figure 1: Source code to compute $m^s \bmod n$.

To understand side-channel attacks, consider the pseudocode in Figure 1, which is found in old implementations of both the encryption and decryption steps of RSA, DSA, and other cryptographic systems. In this function, s is the secret key, but because the Taken branch is computationally more expensive than the Not Taken branch, an adversary who can measure the time it takes to execute an iteration of the loop can infer whether the branch was Taken or Not Taken, thereby inferring the value of s one bit at a time [31, 5]. This particular block of code has also been attacked using side-channels involving the cache [44], power [16], fault injection [3, 41], branch predictor [1], electromagnetic radiation [11], and sound [32].

Over the past five decades, numerous solutions [20, 30, 21, 42, 35, 22, 40, 14, 43, 37, 39, 38, 23, 45, 25, 34, 9, 33, 10] have been proposed for defending against side-channel attacks. Unfortunately, these defenses provide point solutions that leave the program open to other side-channel attacks. Given the vast number of possible side-channels, and given the high overhead that comes from composing multiple solutions, we ideally would find a single solution that simultaneously closes a broad class of side-channels.

In this paper, we introduce a technique that does just this, as we focus on the class of *digital side-channels*, which we define as side-channels that carry information over discrete bits. These side-channels are visible to the adversary at the level of both the program state and the instruction set architecture (ISA). Thus, address traces, cache usage, and data size are examples of digital side-channels, while power draw, electromagnetic radiation, and heat are not.

Our key insight is that *all* digital side-channels emerge from variations in program execution, so while other solutions attempt to hide the symptoms—for example, by normalizing the number of instructions along two paths of a branch—we instead attack the root cause by executing extraneous program paths, which we refer to as *decoy* paths. Intuitively, after obfuscation, the adversary's view through any digital side-channel appears the same as if the program were run many times with different inputs. Of course, we must ensure that our system records the output of only the real path and not the decoy paths, so our solution uses a transaction-like system to update memory. On the real paths, each `store` operation first reads the old value of a memory location before writing the new value, while the decoy paths read the old value and write the same old value.

The only distinction between real and decoy paths lies in the values written to memory: Decoy and real paths will write different values, but unless an adversary can break the data encryption, she cannot distinguish decoy from real paths by monitoring digital side-channels. Our solution does *not* defend against non-digital side-channel attacks, because analog side-channels might reveal the difference between the encrypted values that are stored. For example, a decoy path might "increment" some variable x multiple times, and an adversary who can precisely monitor some non-digital side-channel, such as power-draw, might be able to detect that the "increments" to x all write the same value, thereby revealing that the code belongs to a decoy path.

Nevertheless, our new approach offers several advantages. First, it defends against almost all digital side-channel attacks.[1] Second, it does not require that the programs themselves be secret, just the data. Third, it obviates the need for special-purpose hardware. Thus, standard processor features such as caches, branch predictors and prefetchers do not need to be disabled. Finally, in contrast with previous solutions for hiding specific side channels, it places few fundamental restrictions on the set of supported language features.

This paper makes the following contributions:

1. We design a set of mechanisms, embodied in a system that we call Raccoon,[2] that closes digital side-channels for programs executing on commodity hardware. Raccoon works for both single- and multi-threaded programs.

2. We evaluate the security aspects of these mechanisms in several ways. First, we argue that the obfuscated data- and control-flows are correct and are always kept secret. Second, we use information flows over inference rules to argue that Raccoon's own code does not leak information. Third, as an example of Raccoon's defense, we show that Raccoon protects against a simple but powerful side-channel attack through the OS interface.

3. We evaluate the performance overhead of Raccoon and find that its overhead is 8.9× smaller than that of GhostRider, which is the most similar prior work [20].[3] Unlike GhostRider, Raccoon defends against a broad range of side-channel attacks and places many fewer restrictions on the programming language, on the set of applicable compiler optimizations, and on the underlying hardware.

This paper is organized as follows. Section 2 describes background and related work, and Section 3 describes our assumed threat model. We then describe our solution in detail in Section 4 before presenting our security evaluation and our performance evaluation in Sections 5 and 6, respectively. We discuss the implications of Raccoon's design in Section 7, and we conclude in Section 8.

2 Background and Related Work

Side-channel attacks through the OS, the underlying hardware, or the processor's output pins have been a subject of vigorous research. Formulated as the "confinement problem" by Lampson in 1973 [19], such attacks have become relevant for cloud infrastructures where the adversary and victim VMs can be co-resident [29] and also for settings where adversaries have physical access to the processor-DRAM interface [46, 22].

Side-Channels through OS and Microarchitecture. Some application-level information leaks are beyond the application's control, for example, an adversary reading a victim's secrets through the `/proc` filesystem [13], or a victim's floating point registers that are not cleared on a context switch [2]. In addition to such *explicit* information leaks, *implicit* flows rely on *contention* for shared resources, as observed by Wang and Lee [39] for cache channels and extended by Hunger et al. [37] to all microarchitectural channels.

Defenses against such attacks either partition resources [40, 14, 43, 37], add noise [39, 38, 23, 45], or

[1] Section 3 (Threat Model) clarifies the specific side-channels closed by our approach.

[2] Raccoons are known for their clever ability to break their scent trails to elude predators. Raccoons introduce spurious paths as they climb and descend trees, jump into water, and create loops.

[3] GhostRider [20] was evaluated with non-optimized programs executing on embedded CPUs, which results in an unrealistically low overhead (∼10×). Our measurements instead use a modern CPU with an aggressively optimized binary as the baseline.

normalize the channel [17, 20] to curb side-channel capacity. Raccoon's defenses complement prior work that modifies the hardware and/or OS. Molnar et al. [25] describe a transformation that prevents control-flow side-channel attacks, but their approach does not apply to programs that contain function calls and it does not protect against data-flow-based side-channel attacks.

Physical Access Attacks and Secure Processors. Execute-only Memory (XOM) [36] encrypts portions of memory to prevent the adversary from reading secret data or instructions from memory. The AEGIS [35] secure processor provides the notion of tamper-evident execution (recognizing integrity violations using a merkle tree) and tamper-resistant computing (preventing an adversary from learning secret data using memory encryption). Intel's Software Guard Extensions (SGX) [24] create "enclaves" in memory and limit accesses to these enclaves. Both XOM and SGX are only partially successful in prevent the adversary from accessing code because an adversary can still disassemble the program binary that is stored on the disk. In contrast, Raccoon permits release of the transformed code to the adversary. Hence Raccoon never needs to encrypt code memory.

Oblivious RAM. AEGIS, XOM, and Intel SGX do not prevent information leakage via memory address traces. Memory address traces can be protected using Oblivious RAM, which re-encrypts and re-shuffles data after each memory access. The Path ORAM algorithm [34] is a tree-based ORAM scheme that adds two secret on-chip data structures, the *stash* and *position map*, to piggyback multiple writes to the in-memory data structure. While Raccoon uses a modified version of the Path ORAM algorithm, the specific ORAM implementation is orthogonal to the Raccoon design.

The Ascend [9] secure processor encrypts memory contents and uses the ORAM construct to hide memory access traces. Similarly, Phantom [22] implements ORAM to hide memory access traces. Phantom's memory controller leverages parallelism in DRAM banks to reduce overhead of ORAM accesses. However, both Phantom and Ascend assume that the adversary can only access code by reading the contents of memory. By contrast, Raccoon hides memory access traces via control flow obfuscation and software ORAM while still permitting the adversary to read the code. Ascend and Phantom rely on custom memory controllers whereas Memory Trace Oblivious systems that build on Phantom [20] rely on a new, deterministic processor pipeline. In contrast, Raccoon protects off-chip data on commodity hardware.

Memory Trace Obliviousness. GhostRider [20, 21] is a set of compiler and hardware modifications that transforms programs to satisfy Memory Trace Obliviousness (MTO). MTO hides control flow by transforming programs to ensure that the memory access traces are the same no matter which control flow path is taken by the program. GhostRider's transformation uses a type system to check whether the program is fit for transformation and to identify security-sensitive program values. It also pads execution paths along both sides of a branch so that the length of the execution does not reveal the branch predicate value.

However, unlike Raccoon, GhostRider cannot execute on generally-available processors and software environments because GhostRider makes strict assumptions about the underlying hardware and the user's program. Specifically, GhostRider (1) requires the use of new instructions to load and store data blocks, (2) requires substantial on-chip storage, (3) disallows the use of dynamic branch prediction, (4) assumes in-order execution, and (5) does not permit use of the hardware cache (it instead uses a scratchpad memory controlled by the compiler). GhostRider also does not permit the user code to contain pointers or to contain function calls that use or return secret information. By contrast, Raccoon runs on SGX-enabled Intel processors (SGX is required to encrypt values on the data bus) and permits user programs to contain pointers, permits the use of possibly unsafe arithmetic statements, and allows the use of function calls that use or return secret information.

3 Threat Model and System Guarantees

This section describes our assumptions about the underlying hardware and software, along with Raccoon's obfuscation guarantees.

Hardware Assumptions. We assume that the adversary can monitor and tamper with any digital signals on the processor's I/O pins. We also assume that the processor is a sealed chip [35], that all off-chip resources (including DRAM, disks, and network devices) are untrusted, that all read and written values are encrypted, and that the integrity of all reads and writes is checked.

Software Assumptions. We assume that the adversary can run malicious applications on the same operating system and/or hardware as the victim's application. We allow malicious applications to probe the victim application's run-time statistics exposed by the operating system (*e.g.* the stack pointer in `/proc/pid/stat`). However, we assume that the operating system is trusted, so Iago attacks [7] are out of scope.

The Raccoon design assumes that the input program is free of errors, *i.e.* (1) the program does not contain bugs that will induce application crashes, (2) the program does not exhibit undefined behavior, and (3) if multi-threaded, then the program is data-race free. Under these assumptions, Raccoon does not introduce new termination-channel leaks, and Raccoon correctly obfuscates multi-threaded programs.

Raccoon statically transforms the user code into an obfuscated binary; we assume that the adversary has access to this transformed binary code and to any symbol table and debug information that may be present.

In its current implementation, Raccoon does not support all features of the C99 standard. Specifically, Raccoon cannot obfuscate I/O statements[4] and non-local `goto` statements. While `break` and `continue` statements do not present a fundamental challenge to Raccoon, our current implementation does not obfuscate these statements. Raccoon cannot analyze libraries since their source code is not available when compiling the end-user's application.

As with related solutions [30, 20, 21], Raccoon does not protect information leaks from loop trip counts, since naïvely obfuscating loop back-edges would create infinite loops. For the same reason, Raccoon does not obfuscate branches that represent terminal cases of recursive function calls. However, to address these issues, it is possible to adapt complementary techniques designed to close timing channels [42], which can limit information leaks from loop trip counts and recursive function calls.

Raccoon includes static analyses that check if the input program contains these unsupported language constructs. If such constructs are found in the input program, the program is rejected.

System Guarantees. Within the constraints listed above, Raccoon protects against all digital side-channel attacks. Raccoon guarantees that an adversary monitoring the digital signals of the processor chip cannot differentiate between the real path execution and the decoy path executions. Even after executing multiple decoy program paths, Raccoon guarantees the same final program output as the original program.

Raccoon guarantees that its obfuscation steps will not introduce new program bugs or crashes, so Raccoon does not introduce new information leaks over the termination channel.

Assuming that the original program is race-free, Raccoon's code transformations respect the original program's control and data dependences. Moreover, Raccoon's obfuscation code uses thread-local storage. Thus, Raccoon's obfuscation technique works seamlessly with multi-threaded applications because it does not introduce new data dependences.

```
1: $p \leftarrow \&a$;
2: if $secret$ = true then
3:     ...                                   ▷ Real path.
4: else
5:     ...                                   ▷ Decoy path.
6:     $p \leftarrow \&b$;        ▷ Dummy instructions do not update $\boldsymbol{p}$.
7:     $*p \leftarrow 10$;          ▷ Accesses variable $\boldsymbol{a}$ instead of $\boldsymbol{b}$!
8: end if
```

Figure 2: Illustrating the importance of Property 2. This code fragment shows how solutions that do not update memory along decoy paths may leak information. If the decoy path is not allowed to update memory, then the dereferenced pointer in line 7 will access $\boldsymbol{a}$ instead of accessing $\boldsymbol{b}$, which reveals that the statement was part of a decoy path.

4 Raccoon Design

This section describes the design and implementation of Raccoon from the bottom-up. We start by describing the two critical properties of Raccoon that distinguish it from other obfuscation techniques. Then, after describing the key building block upon which higher-level oblivious operations are built, we describe each of Raccoon's individual components: (1) a taint analysis that identifies program statements that require obfuscation (Section 4.3), (2) a runtime transaction-like memory mechanism for buffering intermediate results along decoy paths (Section 4.4), (3) a program transformation that obfuscates control-flow statements (Section 4.5), and (4) a code transformation that uses software Path ORAM to hide array accesses that depend on secrets (Section 4.6). We then describe Raccoon's program transformations that ensure crash-free execution (Section 4.7). Finally, we illustrate with a simple example the synergy among Raccoon's various obfuscation steps (Section 4.8).

4.1 Key Properties of Our Solution

Two key properties of Raccoon distinguish it from other branch-obfuscating solutions [20, 21, 25, 8]:

- **Property 1:** Both real and decoy paths execute actual program instructions.
- **Property 2:** Both real and decoy paths are allowed to update memory.

Property 1 produces decoy paths that—from the perspective of an adversary monitoring a digital side-channel—are indistinguishable from from real paths.

[4] Various solutions have been proposed that allow limited use of "transactional" I/O statements through runtime systems [6], operating systems [28], or the underlying hardware [4].

Without this property, previous solutions can close one side-channel while leaving other side-channels open. To understand this point, we refer back to Figure 1 and consider a solution that normalizes execution time along the two branch paths in the Figure by adding NOP instructions to the Not Taken path. This solution closes the timing channel but introduces different instruction counts along the two branch paths. On the other hand, the addition of dummy instructions to normalize instruction counts will likely result in different execution time along the two branch paths, since (on commodity hardware) the NOP instructions will have a different execution latency than the multiply instruction.

Property 2 is a special case of Property 1, but we include it because the ability to update memory is critical to Raccoon's ability to obfuscate execution. For example, Figure 2 shows that if the decoy path does not update the pointer ***p***, then the subsequent decoy statement will update ***a*** instead of ***b***, revealing that the assignment to ****p*** was part of a decoy path.

4.2 Oblivious Store Operation

Raccoon's key building block is the oblivious store operation, which we implement using the CMOV x86 instruction. This instruction accepts a condition code, a source operand, and a destination operand; if the condition is true, it moves the source operand to the destination. When both the source and the destination operands are in registers, the execution of this instruction does not reveal information about the branch predicate (hence the name *oblivious* store operation).[5] As we describe shortly, many components in Raccoon leverage the oblivious store operation. Figure 3 shows the x86 assembly code for the CMOV wrapper function.

4.3 Taint Analysis

Raccoon requires the user to annotate secret variables using the __attribute__ construct. With these secret variables identified, Raccoon performs inter-procedural taint analysis to identify branches and data access statements that require obfuscation. Raccoon propagates taint across both implicit and explicit flow edges. The result of the taint analysis is a list of memory accesses and branch statements that must be obfuscated to protect privacy.

```
cmov(uint8_t pred, uint32_t t_val, uint32_t f_val) {
    uint32_t result;
    __asm__ volatile (
        "mov    %2, %0;"
        "test   %1, %1;"
        "cmovz  %3, %0;"
        "test   %2, %2;"
        : "=r" (result)
        : "r" (pred), "r" (t_val), "r" (f_val)
        : "cc"
    );
    return result;
}
```

Figure 3: CMOV wrapper

4.4 Transaction Management

To support Properties 1 and 2, Raccoon executes each branch of an obfuscated if-statement in a transaction. In particular, Raccoon buffers load and store operations along each path of an if-statement, and Raccoon writes values along the real path to DRAM using the oblivious store operation. If a decoy path tries to write a value to the DRAM, Raccoon uses the oblivious store operation to read the existing value and write it back. At compile time, Raccoon transforms load and store operations so that they will be serviced from the transaction buffers. Figure 4 shows pseudocode that implements transactional loads and stores. Loads and stores that appear in non-obfuscated code do not use the transaction buffers.

4.5 Control-Flow Obfuscation

To obfuscate control flow, Raccoon forces control flow along both paths of an obfuscated branch, which requires three key facilities: (1) a method of perturbing the branch outcome, (2) a method of bringing execution control back from the end of the if-statement to the start of the if-statement so that execution can follow along the unexplored path, and (3) a method of ensuring that memory updates along decoy path(s) do not alter non-transactional memory. The first facility is implemented by the obfuscate() function (which forces sequential execution of both paths arising out of a conditional branch instruction). Although Raccoon executes both branch paths, it evaluates the (secret) branch predicate only once. This ensures that the execution of the first path does not unexpectedly change the value of the branch predicate. The second facility is implemented by the epilog() function (which transfers control-flow from the post-dominator of the if-statement to the beginning of the if-statement). Finally the third facility is implemented using the oblivious store operation described earlier. The control-flow obfuscation functions

[5] Contrary to the pseudocode describing the CMOV instruction in the Intel 64 Architecture Software Developer's Manual, our assembly code tests reveal that in 64-bit operating mode when the operand size is 16-bit or 32-bit, the instruction resets the upper 32 bits regardless of whether the predicate is true. Thus the instruction does not leak the value of the predicate via the upper 32 bits, as one might assume based on the manual.

```
// Writes a value to the transaction buffer.
tx_write(address, value) {
    if (threaded program)
        lock();

    // Write to both the transaction buffer
    // and to the non-transactional storage.
    tls->gl_buffer[address] = value;
    *address = cmov(real_idx == instance,
            value, *address);

    if (threaded program)
        unlock();
}

// Fetches a value from the transaction buffer.
tx_read(address) {
    if (threaded program)
        lock();

    value = *address;
    if (address in tls->gl_buffer)
        value = tls->gl_buffer[address];

    value = cmov(real_idx == instance,
            *address, value);

    if (threaded program)
        unlock();

    return value;
}
```

Figure 4: Pseudocode for transaction buffer accesses. Equality checks are implemented using XOR operation to prevent the compiler from introducing an explicit branch instruction.

(obfuscate() and epilog()) use the libc setjmp() and longjmp() functions to transfer control between program points.

Safety of setjmp() and longjmp() Operations. The use of setjmp() and longjmp() is safe as long as the runtime system does not destroy the activation record of the caller of setjmp() prior to calling longjmp(). Thus, the function that invokes setjmp() should not return until longjmp() is invoked. To work around this limitation, Raccoon copies the stack contents along with the register state (identified by the jmp_buff structure) and restores the stack before calling longjmp(). To avoid perturbing the stack while manipulating the stack, Raccoon manipulates the stack using C macros and global variables.

As an additional safety requirement, the runtime system must not remove the code segment containing the call to setjmp() from instruction memory before the call to longjmp(). Because both obfuscate()—which calls setjmp()—and epilog()—which calls longjmp()—are present in the same program module, we know that that the code segment will not vanish before calling longjmp().

Obfuscating Nested Branches. Nested branches are obfuscated in Raccoon by maintaining a stack of transaction buffers that mimics the nesting of transactions. Unlike traditional transactions, transactions in Raccoon are easier to nest because Raccoon can determine whether to commit the results or to store them temporarily in the transaction buffer *at the beginning of the transaction* (based on the secret value of the branch predicate).

4.6 Software Path ORAM

Raccoon's implementation of the Path ORAM algorithm builds on the oblivious store operation. Since processors such as the Intel x86 do not have a trusted memory (other than a handful of registers) for implementing the *stash*, we modify the Path ORAM algorithm from its original form [34]. Raccoon's Path ORAM implementation cannot directly index into arrays that represent the *position map* or the *stash*, so Raccoon's implementation streams over the *position map* and *stash* arrays and uses the oblivious store operation to selectively read or update array elements. Raccoon implements both recursive [33] as well as non-recursive versions of Path ORAM. Our software implementation of Path ORAM permits flexible sizes for both the *stash memory* and the *position map*.

Section 6.3 compares recursive and non-recursive ORAM implementations with an implementation that streams over the entire data array. Raccoon uses AVX vector intrinsic operations for streaming over data arrays. We find that even with large data sizes, it is faster to stream over the array than perform a single ORAM access.

4.7 Limiting Termination Channel Leaks

By executing instructions along decoy paths, Raccoon might operate on incorrect values. For example, consider the statement if (y != 0) { z = x / y; }. If $y = 0$ for a particular execution and if Raccoon executes the decoy path with $y = 0$, then the program will crash due to a division-by-zero error, and the occurrence of this crash in an otherwise bug-free program would reveal that the program was executing a decoy path (and, consequently, that $y = 0$).

To avoid such situations, Raccoon prevents the program from terminating abnormally due to exceptions. For each integer division that appears in a transaction (along both real and decoy paths), Raccoon instruments the operation so that it obliviously (using cmov) replaces

```
/* Sample user code. */
01: int array[512] __attribute__((annotate ("secret")));
02: if (array[mid] <= x) {
03:     l = mid;
04: } else {
05:     r = mid;
06: }

/* Transformed pseudocode. */
07: r1 = stream_load(array, mid);
08: r2 = r1 <= x;
09: key = obfuscate(r2, r3);

10: if (r3) {
11:     tx_write(l, mid);
12: } else {
13:     tx_write(r, mid);
14: }

15: epilog(key);
```

Figure 5: Sample code and transformed pseudocode.

the divisor with a non-zero value. To prevent integer division overflow, Raccoon checks whether the dividend is equal to INT_MIN and whether the divisor is equal to -1; if so, Raccoon obliviously substitutes the divisor to prevent a division overflow. Raccoon also disables floating point exceptions using `fedisableexcept()`. Similarly, array `load` and `store` operations appearing on the decoy path are checked (again, obliviously, using `cmov`) for out-of-bounds accesses. Thus, to ensure that the execution of decoy paths does not crash the program, Raccoon patches unsafe operations. Section 5.3 demonstrates that this process of patching unsafe operations does not leak secret information to the adversary.

4.8 Putting It All Together

We now explain how Raccoon transforms the code shown in Figure 5. Here, the `secret` annotation informs Raccoon that the contents of `array` are secret.

Static taint analysis then reveals that the branch predicate (line 2) depends on the secret value, so Raccoon obfuscates this branch. Similarly, implicit flow edges from the branch predicate to the two assignment statements (at lines 3 and 5) indicate that Raccoon should use the oblivious `store` operation for both assignment statements.

Accordingly, Raccoon replaces direct memory stores for `l` and `r` with function calls that write into transaction buffers in lines 11 and 13 of the transformed pseudocode. The access to `array` in line 1 is replaced by an oblivious streaming operation in line 7. Finally, the branch in line 2 is obfuscated by inserting the `obfuscate()` and `epilog()` function calls. The `epilog()` and `obfuscate()` function calls are coordinated over the `key` variable. To prevent the compiler from deleting or optimizing security-sensitive code sections, Raccoon marks security-sensitive functions, variables, and memory access operations as `volatile` (not shown in the transformed IR).[6]

At runtime, the transformed code executes the following steps:

1. Line 7 streams over the array and uses ORAM to load a single element (identified by `mid`) of the array.

2. Line 8 calculates the actual value of the branch predicate.

3. The key to this obfuscation lies in the `epilog()` function on line 15, which forces the transformed code to execute twice. The first time this function is called, it transfers control back to line 9. The second time this function is called, it simply returns, and program execution proceeds to other statements in the user's code.

4. Line 9 obfuscates the branch outcome. The first time the `obfuscate()` function returns, it stores 0 in `r3`, and control is transferred to the statement at line 13, where the `tx_write()` function call updates the transaction buffer. Non-transactional memory is updated only if this path corresponds to the real path.

   The second time the `obfuscate()` function returns, it stores 1 in `r3`, and control is transferred to the statement at line 11, again calling the `tx_write()` function to update the transaction buffer. Again, non-transactional memory is updated only if this path corresponds to the real path.

5 Security Evaluation

In this section, we first demonstrate that the control-flows and data-flows in obfuscated programs are correct and that they are independent of the secret value. Then, using type-rules that track information flows, we argue that Raccoon's own code does not leak secret information. We then illustrate Raccoon's defenses against termination channels by reasoning about exceptions in x86 processors. Finally, we evaluate Raccoon's ability to prevent side-channel attacks via the `/proc` filesystem.

5.1 Security of Obfuscated Code

In this section, we argue that the obfuscated control-flows and data-flows (1) preserve the original program's

[6] The C99 standard states that any "any expression referring to [a volatile object] shall be evaluated *strictly* according to the rules of the abstract machine", and the abstract machine is defined in a manner that considers that "issues of optimization are irrelevant".

dependences and (2) do not reveal any secret information. We only describe scalar loads and stores, since all array-loads and array-stores are obfuscated by simply streaming over the array. To simplify the explanation, the following arguments describe a top-level (*i.e.* a non-nested) branch. The same arguments can be extended to nested branches by maintaining a stack of transaction buffers.

Correctness of Obfuscated Data-Flow. To ensure correct data-flow, Raccoon uses a combination of transaction buffers and non-transactional storage (*i.e.* main memory). Raccoon sets up a fresh transaction buffer for each thread that executes a new path. Figure 4 shows the implementation of buffered `load` and `store` operations for use with transactions. The `store` operations along real paths write to both transaction buffers and non-transactional storage (since threads cannot share data that is stored in thread-local transaction buffers).

Consider a non-obfuscated program that stores a value to a memory location *m* in line 10 and loads a value from the same location in line 20. We now consider four possible arrangements of these two `load` and `store` operations in the obfuscated program, where each operation may reside either inside or outside a transaction. Our goal is to ensure that the `load` operation always reads the correct value, whether the correct value resides in a transactional buffer or in non-transactional storage.

- **`store` outside transaction, `load` inside transaction:** This implies that there is no `store` operation on *m* within the transaction. Thus, the transaction buffer does not contain an entry for *m*, and the `load` operation reads the value from the non-transactional storage.

- **`store` inside transaction, `load` inside transaction:** Since the transaction has previously written to *m*, the transaction buffer contains an entry for *m*, and the `load` operation fetches the value from the transaction buffer.

- **`store` inside transaction, `load` outside transaction:** This implies that the `store` operation must lie along the real path. Real-path execution updates non-transactional storage. Since `load` operations outside of transactions always fetch from non-transactional storage, the `load` operation reads the correct value of *m*.

- **`store` outside transaction, `load` outside transaction:** Raccoon does not change `load` or `store` operations that are located outside of the transactions. Hence the non-obfuscated reaching definition remains unperturbed.

Raccoon correctly obfuscates multi-threaded code as well. In programs obfuscated by Raccoon, decoy paths only update transactional buffers. Thus, only the `store` operations on real path affect reaching definitions of the obfuscated program. Furthermore, `store` (or `load`) operations along real path immediately update (or fetch) non-transactional storage and do not wait until the transaction execution ends. Thus, memory updates from execution of real paths are immediately visible to all threads, ensuring that inter-thread dependences are not masked by transactional execution. Finally, all transactional `load` and `store` operations use locks to ensure that these accesses are atomic. Put together, `load` and `store` operations on real paths are atomic and globally-visible, whereas `store` operations on decoy paths are only locally-visible and get discarded upon transaction termination. We thus conclude that the obfuscated code maintains correct data-flows for both single- and multi-threaded programs.

Concealing Obfuscated Data-Flow. Raccoon always performs two `store` operations for every transactional write operation, regardless of whether the write operation belongs to a real path or a decoy path. Moreover, by leveraging the oblivious `store` operation, Raccoon hides the specific value written to the transactional buffer or to the non-transactional storage. Although the `tx_read()` function uses an `if`-statement, the predicate of the `if`-statement is not secret, since an adversary can simply inspect the code and differentiate between repeated and first-time memory accesses. Thus, we conclude that the data-flows exposed to the adversary do not leak secret information.

Concealing Obfuscated Control-Flow. Raccoon converts control flow that depends on secret values into static (*i.e.* deterministically repeatable) control-flow that does not depend on secret values. Given a conditional branch instruction and two branch targets in the LLVM Intermediate Representation (IR), Raccoon always forces execution along the first target and then the second target. Thus, the sequence of executed branch targets depends on the (static) encoding of the branch instruction and not on the secret predicate.

5.2 Security of Obfuscation Code

Raccoon's own code should never leak secret information, so in this section, we demonstrate the security of the secret information maintained by Raccoon. Because the Raccoon code exposes only a handful of APIs (Table 1) to user applications, we can perform a detailed analysis of the code's entry- and exit-points to ensure that these

$$\text{T-LOAD}\quad \frac{l_r(p) = \mathsf{L}, A = pts(p), m = \max_{a \in A} l_a(a)}{\langle x = \mathbf{load} p; c, l_a, l_r\rangle \rightarrow \langle c, l_a, l_r[x \mapsto m]\rangle}$$

$$\text{T-STORE}\quad \frac{l_r(p) = \mathsf{L}, A = pts(p)}{\langle \mathbf{store}(x,p); c, l_a, l_r\rangle \rightarrow \langle c, \bigcup_{a \in A} l_a[a \mapsto max(l_a(a), l_r(x)), l_r]\rangle}$$

$$\text{T-BINOP}\quad \frac{}{\langle v = \textbf{binary-op}(x,y); c, l_a, l_r\rangle \rightarrow \langle c, l_a, l_r[v \mapsto \mathbf{max}(l_r(x), l_r(y))]\rangle}$$

$$\text{T-UNOP}\quad \frac{}{\langle v = \textbf{unary-op}(x); c, l_a, l_r\rangle \rightarrow \langle c, l_a, l_r[v \mapsto l_r(x)]\rangle}$$

$$\text{T-BRANCH}\quad \frac{l_r(p) = \mathsf{L}, \langle c_t; c, l_a, l_r\rangle \rightarrow \langle c, l_a', l_r'\rangle, \langle c_f; c, l_a, l_r\rangle \rightarrow \langle c, l_a'', l_r''\rangle}{\langle \mathbf{branch}(p, c_t, c_f); c, l_a, l_r\rangle \rightarrow \langle c, M(l_a', l_a''), M(l_r', l_r'')\rangle}$$

$$\text{T-CMOV}\quad \frac{}{\langle v = \mathbf{cmov}(p,t,f); c, l_a, l_r\rangle \rightarrow \langle c, l_a, l_r[v \mapsto \mathsf{L}]\rangle}$$

$$\text{T-SKIP}\quad \frac{}{\langle v = \mathbf{skip}; c, l_a, l_r\rangle \rightarrow \langle c, l_a, l_r\rangle}$$

$$\text{T-SEQUENCE}\quad \frac{\langle c_0, l_a, l_r\rangle \rightarrow \langle c_0', l_a', l_r'\rangle}{\langle c_0; c_1, l_a, l_r\rangle \rightarrow \langle c_0'; c_1, l_a', l_r'\rangle}$$

$$M(l', l'') = \forall_{x \,\in\, \{K(l') \,\cup\, K(l'')\}}\ (x,\ max(l'(x), l''(x))) \qquad K(l) = \{x \mid (x,s)\ \in\ l\}$$

Figure 6: Typing rules and supporting functions that check security of Raccoon's code.

Category	Functions	Secret info.
Control-flow obfuscation.	`obfuscate()`, `epilog()`.	Predicate value
Wrapper functions for unsafe operations.	`stream_load()`, `stream_store()`, `div_wrapper()`.	Array index, division operands.
Registering stack and array information.	`reg_memory()`, `reg_stack_base()`.	-
Initialization and clean-up functions.	`init_handler()`, `exit_handler()`.	-

Table 1: Entry-points of Raccoon's library.

interfaces never spill secret information outside of Raccoon's own code.

Type System for Tracking Information Flows. Figure 6 shows a subset of the typing rules used for checking the IR of Raccoon's own code. These rules express small-step semantics that track security labels. We assume the existence of a functions $l_r : v \rightarrow \gamma$ and $l_a : \Delta \rightarrow \gamma$ that map LLVM's virtual registers (v) and addresses (Δ) to security labels (γ), respectively. Security labels can be of two types: `L` represents low-context (or public) information, while `H` represents high-context (or secret) information. Secret information listed in Table 1 is assigned the `H` security label, while all other information is assigned the `L` security label. We also assume the existence of a function $pts : r \rightarrow \{\Delta\}$ that returns the points-to set for a given virtual register r.

Our goal is to ensure that Raccoon does not leak secret information either through control-flow (`branch` instructions) or data-flow (`load` and `store` instructions). The typing rules in Figure 6 verify that information labeled as secret never appears as an address in a `load` or `store` instruction and never appears as a predicate in a `branch` instruction. Otherwise, the typing rules would result in a stuck transition. To prevent information leaks, Raccoon passes the secret information through the declassifier (`cmov`) before executing a `load`, `store`, or `branch` operation with a secret value. Due to its oblivious nature, the `cmov` operation resets the security label of its destination to `L`.

Security Evaluation of the `cmov` Operation. The tiny code size of the `cmov` operation (Figure 3) permits us to thoroughly inspect each instruction for possible information leaks. We use the Intel 64 Architecture Software Developer's Manual to understand the side-effects of each instruction.

Since the code operates on the processor registers only and never accesses memory, it operates within the (trusted) boundary of the sealed processor chip. The secret predicate is loaded into the `%1` register. The `mov` instruction in line 4 initializes the destination register with `t_val`. The `test` instruction at line 5 checks if `pred` is zero and updates the Zero flag (`ZF`), Sign flag (`SF`), and the Parity flag (`PF`) to reflect the comparison. The subsequent `cmovz` instruction copies `f_val` into the destination register *only if* `pred` is zero. At this point, `ZF`, `SF`, and `PF` still contain the results of the comparison. The `test` instruction at line 7 overwrites these flags by comparing known non-secret values.

Since none of the instructions ever accesses memory, these instructions can never raise a General Protection Fault, Page Fault, Stack Exception Fault, Segment Not Present exception, or Alignment Check exception. None of these instructions uses the `LOCK` prefix, so they will never generate an Invalid Opcode (#UD) exception. As per the Intel Software Developer's Manual, the above instructions cannot raise any other exception besides the ones listed above. Through a manual analysis of the descriptions of 253 performance events[7] supported

[7] Intel 64 and IA-32 Architectures Software Developers Manual, Section 19.5.

by our target platform, we discovered that only two performance events are directly relevant to the code in Figure 3: `PARTIAL_RAT_STALLS.FLAGS_MERGE_UOP` and `UOPS_RETIRED.ALL`. The first event (`FLAGS_MERGE_UOP`), which counts the number of performance-sensitive flags-merging micro-operations, produces the same value for our code, no matter whether the predicate is true or false. The second event (`UOPS_RETIRED.ALL`) counts the number of retired micro-operations. Since details of micro-operation counts for x86 instructions are not publicly available, we used an unofficial source of instruction tables[8] to verify that the micro-operation count for a `cmov` instruction is independent of the instruction's predicate. We thus conclude that the code in Figure 3 does not leak the secret predicate value.

Category	Interrupt list
Arithmetic errors	Division by zero, invalid operands, overflow, underflow, inexact results.
Memory access interrupts	Stack exception fault, general protection fault, page fault.
Debugging interrupts	Single-step, breakpoint.
Privileged operations	Invalid TSS, segment not present.
Coprocessor (legacy) interrupts	No coprocessor, coprocessor overrun, coprocessor error.
Other	Non-maskable interrupt, invalid opcode, double-fault abort.

Table 2: Categorized list of x86 hardware exceptions.

5.3 Termination Leaks

In Section 4.7, we explained how Raccoon patches division operations and memory access instructions to prevent the program from crashing along decoy paths. We now explain why these patches are sufficient in preventing the introduction of new termination leaks. Table 2 shows a categorized list of exception conditions arising in Intel x86 processors[9] that may terminate programs. Among these interrupts, Raccoon transparently handles arithmetic and memory access interrupts.

Debugging interrupts are irrelevant for the program safety discussion because they do not cause the program to terminate. Our threat model does not apply obfuscation to OS or kernel code. Since we do not expect user programs to contain privileged instructions, Raccoon does not need to mask interrupts from privileged operations. Coprocessor interrupts are relevant to Numeric Processor eXtensions (NPX), which are no longer used today. Non-maskable interrupts are not caused by software events and thus need not be hidden by Raccoon. Branches in Raccoon always jump to the start of valid basic blocks, so invalid opcodes can never occur in an obfuscated version of a correct program. A double-fault exception occurs when the processor encounters an exception while invoking the handler for a previous exception. Aborts due to double-fault need not be hidden by Raccoon because none of the primary exceptions in an obfuscated program will leak secret information. In conclusion, Raccoon prevents abnormal program termination, thus guaranteeing that Raccoon's execution of decoy paths will never cause information leaks over the termination channel.

5.4 Defense Against Side-Channel Attacks

We have argued in Sections 5.1 and 5.2 that Raccoon closes digital side-channels. We now show a concrete example of a simple but powerful side-channel attack, and we use basic machine-learning techniques to visually illustrate Raccoon's defense against this attack. We model the adversary as a process that observes the instruction pointer (IP) values of the victim process. Both the victim process and the adversary process run on the same machine. The `driver` process starts the `victim` process and immediately pauses the `victim` process by sending a `SIGSTOP` signal. The `driver` process then starts the `adversary` process and sends it the process ID of the paused `victim` process. This `adversary` process polls for the instruction pointer of the `victim` process every 5ms via the `kstkeip` field in `/proc/pid/stat`. When the `victim` process finishes execution, the `driver` process sends a `SIGINT` signal to the `adversary` process, signalling it to save its collection of instruction pointers to a file. We run the `victim` programs with various secret inputs and each run produces a (sampled) trace of instruction pointers. Each such trace is labelled with the name of the program and an identifier for the secret input. We collect 300 traces for each label. For the sake of brevity, we show results for only three programs from our benchmark suite.

The labelled traces are then passed through a Support Vector Machine for k-fold cross-validation (we choose $k = 10$) using LIBSVM v3.18. Using the prediction data, we construct a confusion matrix for each program, which conveys the accuracy of a classification system by counting the number of correctly-predicted and mis-predicted values (see Figure 7). The confusion matrices show that for the non-secure executions, the classifier is able to label instruction pointer traces with high accuracy. By contrast, when using traces from obfuscated execution, the classifier's accuracy is significantly lower.

6 Performance Evaluation

Methodology. Raccoon is implemented in the LLVM compiler framework v3.6. In our test setup, the host op-

[8] `http://www.agner.org/optimize/instruction_tables.pdf`

[9] `http://www.x86-64.org/documentation/abi.pdf`

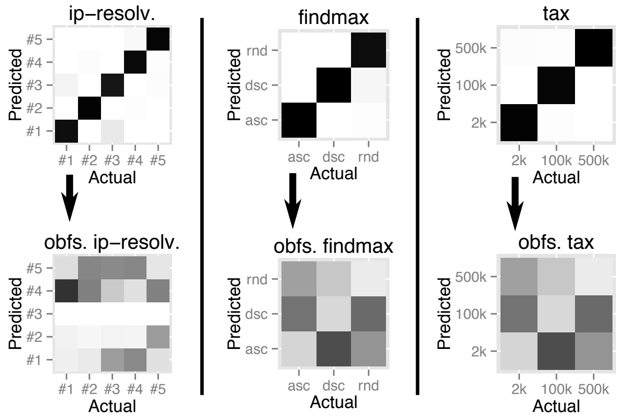


Figure 7: Confusion matrices for `ip-resolv`, `find-max` and `tax`. The top matrices describe original execution. The bottom matrices describe obfuscated execution.

erating system is CentOS 6.3. To evaluate performance, we use 15 programs (eight small kernels and seven small applications). Table 3 summarizes their characteristics and the associated input data sizes. The bottom eight programs in the table are the same programs used to evaluate GhostRider [20, 21], and we use these to compare Raccoon's overhead against that of GhostRider. To simplify the comparison between Raccoon and GhostRider, we use data sizes that are similar to those used to evaluate GhostRider [20]. Raccoon uses the `__attribute__` construct to mark secret variables—which mandates that the input programs are written in C/C++. However the rest of Raccoon operates entirely on the LLVM IR and does not use any source-language features. Thus, Raccoon can easily be ported to work with any language that can be compiled to the LLVM IR. All tests use the LLVM/Clang compiler toolchain.

We run all experiments on a machine with two Intel Xeon (Sandy Bridge) processors and with 32 GB (8 × 4 GB) DDR3 memory. Each processor has eight cores with 256 KB private L2 caches. The eight cores on a processor chip share a 20 MB L3 cache. Streaming encryption/decryption hardware makes the cost of accessing memory from encrypted RAM banks almost the same as the cost of accessing a DRAM bank. The underlying hardware does not support encrypted RAM banks, but we do not separately add any encryption-related overhead to our measurements because the streaming access cost is almost the same with or without encryption.

Performance measurements of our simulated ORAM use the native hardware performance event—`UNHALTED_CORE_CYCLES`. We measure overhead using `clock_gettime()`. Our software Path ORAM implementation is configured with a block size of 64 bytes. Each node in the Path ORAM tree stores 10 blocks. The stash size is selected at ORAM initialization time and is set to $\frac{\textit{ORAM block count}}{100}$ or 64 entries, whichever is higher.

Name	Lines	Data size
Classifier	86	5 features, 5 records
IP resolver	247	3,500 records
Medical risk analysis	92	3,200 records
CRC32	76	10 KB
Genetic algorithm	446	pop. size = 1 KB
Tax calculator	350	-
Radix sort	675	256K elements
Binary search	35	10K elements
Dijkstra	50	1K edges
Find max	27	1K elements
Heap add	24	1K elements
Heap pop	42	10K elements
Histogram	40	1K elements
Map	29	1K elements
Matrix multiplication	28	500 x 500 values

Table 3: Benchmark programs used for performance evaluation of Raccoon. The bottom eight programs are also used to evaluate GhostRider. The remaining seven programs cannot be transformed by GhostRider because these programs use pointers and invoke functions in the secret context.

6.1 Obfuscation Overhead

There are two main sources of Raccoon overhead: (1) the cost of the ORAM operations (or streaming) and (2) the cost of control-flow obfuscation (including the cost of buffering transactional memory accesses, the cost of copying program stack and CPU registers, and the cost of obliviously patching arithmetic and memory access instructions). We account for ORAM/streaming overhead over both real and decoy paths. Of course, the overhead varies with program characteristics, such as size of the input data, number of obfuscated statements, and number of memory access statements. Figure 8 shows the obfuscation overhead for the benchmark programs when compared with an aggressively optimized (compiled with `-O3`) non-obfuscated binary executable. The geometric mean of the overhead is ∼16.1×. Applications closer to the left end of the spectrum had low overheads due to Raccoon's ability to leverage existing compiler optimizations (if-conversion, automatic loop unrolling, and memory to register promotion). In most applications with high obfuscation overhead, a majority of the overhead arises from transactional execution in control-flow obfuscation.

6.2 Comparison with GhostRider

To place our work in the context of similar solutions to side-channel defenses, we compare Raccoon with the

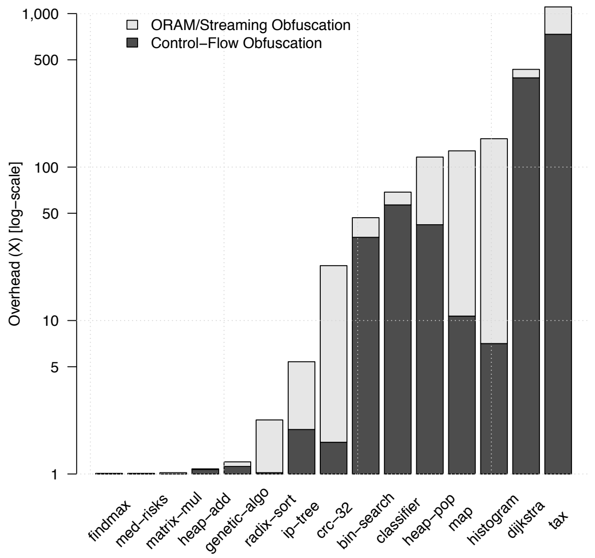


Figure 8: Sources of obfuscation overhead.

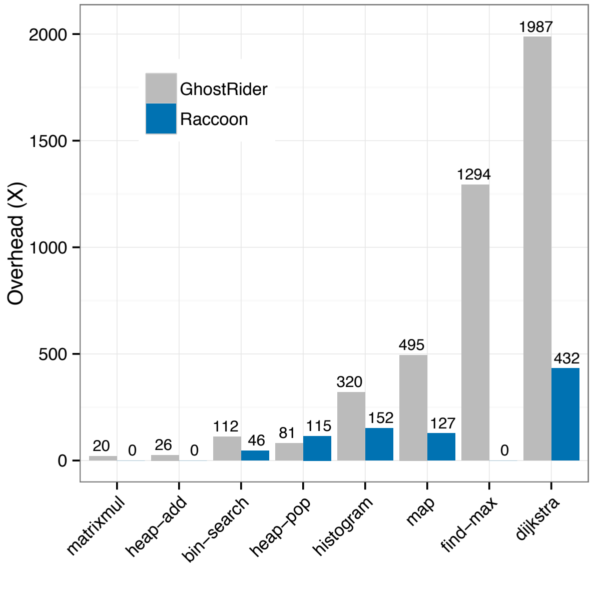


Figure 9: Overhead comparison on GhostRider's benchmarks. Even when we generously underestimate GhostRider's overhead, GhostRider sees an average overhead of 195×, while Raccoon's overhead is 21.8×.

GhostRider hardware/software framework [20, 21] that implements Memory Trace Obliviousness. This section focuses on the performance aspects of the two systems, but as mentioned in Section 2, Raccoon provides significant benefits over GhostRider beyond performance. First, Raccoon provides a broad coverage against many different side-channel attacks. Second, the dynamic obfuscation scheme used in Raccoon strengthens the threat model, since it allows the transformed code to be released to the adversary. Third, Raccoon does not require special-purpose hardware. Finally, since GhostRider adds instructions to mimic address traces in both branch paths, it requires that address traces from obfuscated code be known at compile-time, which significantly limits the programs that GhostRider can obfuscate. Raccoon relaxes this requirement by executing actual code, so Raccoon can transform more complex programs than GhostRider.

Methodology. We now describe our methodology for simulating the GhostRider solution. As with our Raccoon setup, we compare GhostRider's obfuscated program with an aggressively optimized (compiled with `-O3`) non-obfuscated version of the same program. Various compiler optimizations (dead code elimination, vectorization, constant merging, constant propagation, global value optimizations, instruction combining, loop-invariant code motion, and promotion of memory to registers) interfere with GhostRider's security guarantees, so we disable optimizations for the obfuscated program. We manually apply the transformations implemented in the GhostRider compiler. We simulate a processor that is modelled after the GhostRider processor, so we use a single-issue in-order processor that does not allow prefetching into the cache.

There are four reasons why our methodology significantly underestimates GhostRider's overhead. The first three reasons stem from our inability to faithfully simulate all features of the GhostRider processor: (1) We simulate variable-latency instructions, (2) we simulate the use of a dynamic branch predictor, and (3) we simulate a perfect cache for non-ORAM memory accesses. All three of these discrepancies give GhostRider an unrealistically fast hardware platform. The fourth reason arises because our simulator does not support AVX vector instructions, so we are unable to compare GhostRider against a machine that can execute AVX vector instructions.

The non-obfuscated execution uses a 4-issue, out-of-order core with support for Access Map Pattern Matching prefetching scheme [12] for the L1, L2 and L3 data caches. In all other respects, the two processor configurations are identical. Both processors are clocked at 1 GHz. The processor configuration closely matches the configuration described by Fletcher et al. [10], and based on their measurements, we assume that the latency to all ORAM banks is 1,488 cycles per cache line. We run GhostRider's benchmarks on this modified Marss86 simulator and manually add the cost of each ORAM access to the total program execution latency.

Performance Comparison. Figure 9 compares the overhead of GhostRider on the simulated processor and the overhead of Raccoon. Only those benchmark programs that meet GhostRider's assumptions are used in this comparison. The remaining seven applications cannot be transformed by the GhostRider solution because they use pointers or because they invoke functions in the secret context. We see that Raccoon's overhead (geometric mean of 16.1× over all 15 benchmarks, geometric mean of 21.8× over GhostRider-only benchmarks) is significantly lower than GhostRider's overhead (geometric mean of 195×), even when giving GhostRider's processor substantial benefits (perfect caching, lack of AVX-vector support in the baseline processor, and dynamic branch prediction).

6.3 Software Path ORAM

This section considers choices for Raccoon's ORAM implementation. In particular, to run on typical general-purpose processors, we need to modify the Path ORAM algorithm to assume just a tiny amount of trusted memory, which forces us to stream the *position map* and *stash* multiple times to obliviously copy or update elements.

We thus consider three possible implementations. The first, recursive ORAM [33], places the *position map* in a smaller ORAM until the *position map* of the smallest ORAM fits in the CPU registers. The second is a non-recursive solution that streams over a single large *position map*. The third uses AVX intrinsic operations and streams over the entire array to access a single element.

Figure 10(a) compares the cost of ORAM initialization for different ORAM sizes in our recursive and non-recursive ORAM implementations. On this log-log scale, we see that the non-recursive ORAM is significantly faster than the recursive ORAM for all sizes. Figure 10(b) compares our non-recursive ORAM implementation against the streaming approach. In particular, it measures the cost of accessing a single element and the cost of 64 single-element random accesses using ORAM and streaming. We see that the streaming implementation is orders of magnitude faster than our non-recursive ORAM.

In summary, our software implementation of Path ORAM requires non-trivial changes to the original Path ORAM algorithm. Unfortunately, these changes impose a prohibitively large memory bandwidth requirement, making the modified software Path ORAM far costlier than streaming over arrays. Raccoon's obfuscation technique is compatible with the use of dedicated ORAM memory controllers, and Raccoon's overhead can be further reduced by using such special purpose hardware [22].

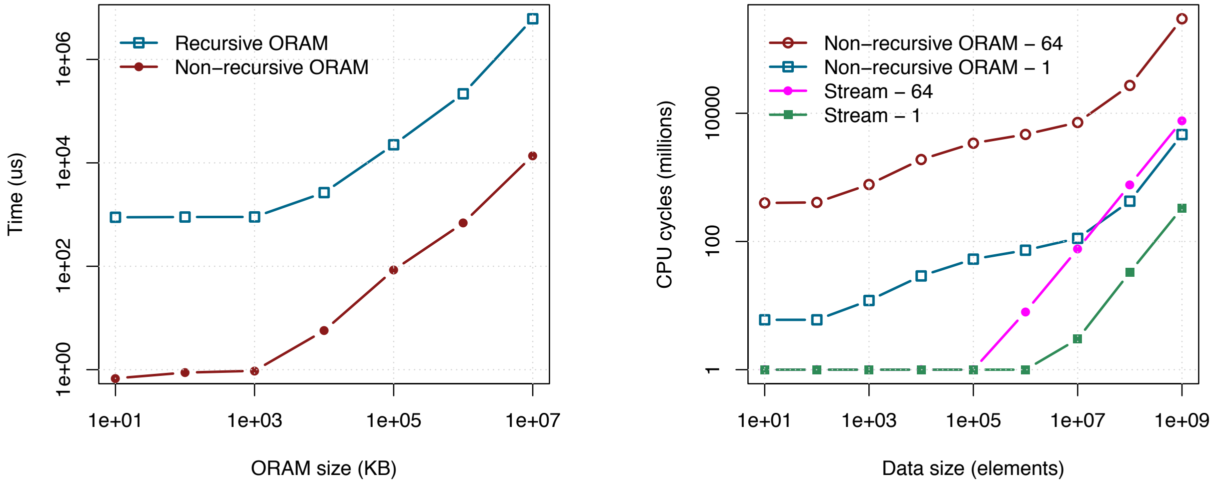
7 Discussion

Closing Other Side-Channels. The existing Raccoon implementation does not defend against kernel-space side-channel attacks. However, many of Raccoon's obfuscation principles can be applied to OS kernels as well. Memory updates in systems such as TxOS [28] can be made oblivious using Raccoon's `cmov` operation. By contrast, non-digital side-channels appear to be fundamentally beyond Raccoon's scope since physical characteristics (power, temperature, EM radiation) of hardware devices make it possible to differentiate between real values and decoy values.

Multi-threaded Programs. Raccoon's data structures are stored in thread-local storage (TLS), so Raccoon can access internal data structures without using locks. Raccoon initializes these data-structures at thread entry-points (identified by `pthread_create()`) and frees them at thread destruction-points (identified by `pthread_exit()`). Raccoon prevents race conditions on the user program's memory by using locks where necessary. Most importantly, as long as the user program is race-free, Raccoon maintains the correct data-flow dependences in both single-threaded and multi-threaded programs, as described in Section 5.1.

Taint Analysis. Raccoon's taint analysis is sound but not complete, so it over-approximates the amount of code that must be obfuscated. For large programs, this over-approximation is a significant source of overhead. Raccoon's taint analysis is flow-insensitive, path-insensitive, and context-insensitive, and Raccoon uses a rudimentary alias analysis technique that assumes two pointers alias if they have the same type. We believe that more precise static analysis techniques can be used to greatly shrink Raccoon's taint graph, thus reducing the obfuscation overhead.

Limitations Imposed by Hardware. Various x86 instructions (`DIV`, `SQRT`, etc.) consume different cycles depending on their operand values. Such operand-dependent instruction execution latency introduces the biggest hurdle in ensuring the security of Raccoon-obfuscated programs. We also believe that the performance overhead of obfuscated programs would be substantially smaller than the current overhead if processors came equipped with (small) scratchpad memory. Based on these conjectures, we plan to explore the impact of modified hardware designs in the near future.



(a) Initialization cost of recursive and non-recursive ORAM implementation (median of 10 measurements for each sample).

(b) Performance comparison of software Path ORAM and streaming over the entire array.

Figure 10: Software ORAM performance.

8 Conclusions

In this paper, we have introduced the notion of digital side-channel attacks, and we have presented a system named Raccoon to defend against such attacks. We have evaluated Raccoon's performance against 15 programs to show that its overhead is significantly less than that of the best prior work and that it has several additional benefits: it expands the threat model, it removes special-purpose hardware, it permits the release of the transformed code to the adversary, and it also expands the set of supported language features. In comparing Raccoon against GhostRider, we find that Raccoon's overhead is 8.9× lower.

Raccoon's obfuscation technique can be enhanced in several ways. First, while the performance overhead of Raccoon-obfuscated programs is high enough to preclude immediate practical deployment, we believe that this overhead can be substantially reduced by employing deterministic or special-purpose hardware. Second, Raccoon's technique of transactional execution and oblivious memory update can be applied to the operating system (OS) kernel, thus paving the way for protection against OS-based digital side-channel attacks. Finally, in addition to defending against side-channel attacks, we believe that Raccoon can be strengthened to defend against covert-channel communication.

Acknowledgments. We thank our shepherd, David Evans, and the anonymous reviewers for their helpful feedback. We also thank Casen Hunger and Akanksha Jain for their help in using machine learning techniques and microarchitectural simulators. This work was funded in part by NSF Grants DRL-1441009 and CNS-1314709 and a gift from Qualcomm.

References

[1] Aciiçmez, O., Koç, c. K., and Seifert, J.-P. On the power of simple branch prediction analysis. In *Symposium on Information, Computer and Communications Security* (2007), pp. 312–320.

[2] Aciicmez, O., and Seifert, J.-P. Cheap Hardware Parallelism Implies Cheap Security. In *Workshop on Fault Diagnosis and Tolerance in Cryptography* (2007), pp. 80–91.

[3] Bao, F., Deng, R. H., Han, Y., A.Jeng, Narasimhalu, A. D., and Ngair, T. Breaking public key cryptosystems on tamper resistant devices in the presence of transient faults. In *Workshop on Security Protocols* (1998), pp. 115–124.

[4] Blundell, C., Lewis, E. C., and Martin, M. Unrestricted transactional memory: Supporting I/O and system calls within transactions. Tech. rep., University of Pennsylvania, 2006.

[5] Brumley, D., and Boneh, D. Remote timing attacks are practical. In *USENIX Security Symposium* (2005).

[6] Carlstrom, B. D., McDonald, A., Chafi, H., Chung, J., Minh, C. C., Kozyrakis, C., and Olukotun, K. The Atomos transactional programming language. In *Conference on Programming Language Design and Implementation* (2006), pp. 1–13.

[7] Checkoway, S., and Shacham, H. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Architec-*

tural Support for Programming Languages and Operating Systems (2013), pp. 253–264.

[8] CRANE, S., HOMESCU, A., BRUNTHALER, S., LARSEN, P., AND FRANZ, M. Thwarting cache side-channel attacks through dynamic software diversity. In *Network and Distributed System Security Symposium* (2015).

[9] FLETCHER, C. W., DIJK, M. V., AND DEVADAS, S. A Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In *ACM Workshop on Scalable Trusted Computing* (2012), pp. 3–8.

[10] FLETCHER, C. W., LING, R., XIANGYAO, Y., VAN DIJK, M., KHAN, O., AND DEVADAS, S. Suppressing the oblivious RAM timing channel while making information leakage and program efficiency trade-offs. In *International Symposium on High Performance Computer Architecture* (2014), pp. 213–224.

[11] GANDOLFI, K., MOURTEL, C., AND OLIVIER, F. Electromagnetic analysis: Concrete results. In *Cryptographic Hardware and Embedded Systems* (2001), pp. 251–261.

[12] ISHII, Y., INABA, M., AND HIRAKI, K. Access map pattern matching for high performance data cache prefetch. *Journal of Instruction-Level Parallelism* (2011), 499–500.

[13] JANA, S., AND SHMATIKOV, V. Memento: Learning secrets from process footprints. In *IEEE Symposium on Security and Privacy* (2012), pp. 143–157.

[14] KIM, T., PEINADO, M., AND MAINAR-RUIZ, G. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In *USENIX Conference on Security Symposium* (2012), pp. 11–11.

[15] KOCHER, P. C. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology* (1996), pp. 104–113.

[16] KOCHER, P. C., JAFFE, J., AND JUN, B. Differential Power Analysis. In *Advances in Cryptology*. Springer Berlin Heidelberg, 1999, pp. 388–397.

[17] KONG, J., ACIICMEZ, O., SEIFERT, J.-P., AND ZHOU, H. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *High Performance Computer Architecture* (2009).

[18] KUHN, M. G. Cipher Instruction Search Attack on the Bus-Encryption Security Microcontroller DS5002FP. *IEEE Transactions on Computers 47*, 10 (1998), 1153–1157.

[19] LAMPSON, B. W. A note on the confinement problem. *Communications of the ACM* (1973), 613–615.

[20] LIU, C., HARRIS, A., MAAS, M., HICKS, M., TIWARI, M., AND SHI, E. GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation. In *Architectural Support for Programming Languages and Operating Systems* (2015), pp. 87–101.

[21] LIU, C., HICKS, M., AND SHI, E. Memory Trace Oblivious Program Execution. In *Computer Security Foundations Symposium* (2013), pp. 51–65.

[22] MAAS, M., LOVE, E., STEFANOV, E., TIWARI, M., SHI, E., ASANOVIC, K., KUBIATOWICZ, J., AND SONG, D. PHANTOM: Practical Oblivious Computation in a Secure Processor. In *Conference on Computer and Communications Security* (2013), pp. 311–324.

[23] MARTIN, R., DEMME, J., AND SETHUMADHAVAN, S. TimeWarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *International Symposium on Computer Architecture* (2012), pp. 118–129.

[24] MCKEEN, F., ALEXANDROVICH, I., BERENZON, A., ROZAS, C. V., SHAFI, H., SHANBHOGUE, V., AND SAVAGAONKAR, U. R. Innovative instructions and software models for isolated execution. In *International Workshop on Hardware and Architectural Support for Security and Privacy* (2013).

[25] MOLNAR, D., PIOTROWSKI, M., SCHULTZ, D., AND WAGNER, D. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Information Security and Cryptology* (2006), pp. 156–168.

[26] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache attacks and countermeasures: the case of AES. In *RSA conference on Topics in Cryptology* (2006), pp. 1–20.

[27] PERCIVAL, C. Cache missing for fun and profit. In *BSDCan* (2005).

[28] PORTER, D. E., HOFMANN, O. S., ROSSBACH, C. J., BENN, A., AND WITCHEL, E. Operating system transactions. In *Symposium on Operating Systems Principles* (2009), pp. 161–176.

[29] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In *Computer and Communications Security* (2009), pp. 199–212.

[30] SABELFELD, A., AND MYERS, A. C. Language-Based Information-Flow Security. *IEEE JSAC* (2003), 5–19.

[31] SCHINDLER, W. A timing attack against RSA with the chinese remainder theorem. In *Cryptographic Hardware and Embedded Systems* (2000), pp. 109–124.

[32] SHAMIR, A., AND TROMER, E. Acoustic cryptanalysis. Online at `http://www.wisdom.weizmann.ac.il/~tromer`.

[33] SHI, E., CHAN, T.-H. H., STEFANOV, E., AND LI, M. Oblivious RAM with O((log n)3) Worst-case Cost. In *International Conference on The Theory and Application of Cryptology and Information Security* (2011), pp. 197–214.

[34] STEFANOV, E., VAN DIJK, M., SHI, E., FLETCHER, C., REN, L., YU, X., AND DEVADAS, S. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *Conference on Computer and Communications Security* (2013), pp. 299–310.

[35] SUH, G. E., FLETCHER, C., CLARKE, D., GASSEND, B., VAN DIJK, M., AND DEVADAS, S. Author Retrospective AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing. In *International Conference on Supercomputing* (2014), pp. 68–70.

[36] THEKKATH, C., LIE, D., MITCHELL, M., LINCOLN, P., BONEH, D., MITCHELL, J., AND HOROWITZ, M. Architectural Support for Copy and Tamper Resistant Software. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (2000), pp. 168–177.

[37] TIWARI, M., HUNGER, C., AND KAZDAGLI, M. Understanding Microarchitectural Channels and Using Them for Defense. In *International Symposium on High Performance Computer Architecture* (2015), pp. 639–650.

[38] VATTIKONDA, B. C., DAS, S., AND SHACHAM, H. Eliminating Fine Grained Timers in Xen. In *Cloud Computing Security Workshop* (2011), pp. 41–46.

[39] WANG, Z., AND LEE, R. B. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *International Symposium on Computer Architecture* (2007), pp. 494–505.

[40] WANG, Z., AND LEE, R. B. A novel cache architecture with enhanced performance and security. In *IEEE/ACM International Symposium on Microarchitecture* (2008), pp. 83–93.

[41] YEN, S.-M., AND JOYE, M. Checking before output may not be enough against fault-based cryptanalysis. *IEEE Transactions on Computers* (2000), 967–970.

[42] Zhang, D., Askarov, A., and Myers, A. C. Predictive mitigation of timing channels in interactive systems. In *Conference on Computer and Communications Security* (2011), pp. 563–574.

[43] Zhang, Y., Juels, A., Oprea, A., and Reiter, M. K. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In *IEEE Symposium on Security and Privacy* (2011), pp. 313–328.

[44] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-VM side channels and their use to extract private keys. In *Conference on Computer and Communications Security* (2012), pp. 305–316.

[45] Zhang, Y., and Reiter, M. K. Duppel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud. In *Conference on Computer and Communications Security* (2013), pp. 827–838.

[46] Zhuang, X., Zhang, T., and Pande, S. HIDE: An Infrastructure for Efficiently Protecting Information Leakage on the Address Bus. In *Architectural Support for Programming Languages and Operating Systems* (2004), pp. 72–84.

$\mathtt{M}^2\mathtt{R}$: Enabling Stronger Privacy in MapReduce Computation

Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, Chunwang Zhang
School of Computing, National University of Singapore
ug93tad@gmail.com, prateeks@comp.nus.edu.sg, changec@comp.nus.edu.sg
ooibc@comp.nus.edu.sg, zhangchunwang@gmail.com

Abstract

New big-data analysis platforms can enable distributed computation on encrypted data by utilizing trusted computing primitives available in commodity server hardware. We study techniques for ensuring privacy-preserving computation in the popular MapReduce framework. In this paper, we first show that protecting only individual units of distributed computation (e.g. map and reduce units), as proposed in recent works, leaves several important channels of information leakage exposed to the adversary. Next, we analyze a variety of design choices in achieving a stronger notion of private execution that is the analogue of using a distributed oblivious-RAM (ORAM) across the platform. We develop a simple solution which avoids using the expensive ORAM construction, and incurs only an additive logarithmic factor of overhead to the latency. We implement our solution in a system called $\mathtt{M}^2\mathtt{R}$, which enhances an existing Hadoop implementation, and evaluate it on seven standard MapReduce benchmarks. We show that it is easy to port most existing applications to $\mathtt{M}^2\mathtt{R}$ by changing fewer than 43 lines of code. $\mathtt{M}^2\mathtt{R}$ adds fewer than 500 lines of code to the TCB, which is less than 0.16% of the Hadoop codebase. $\mathtt{M}^2\mathtt{R}$ offers a factor of 1.3× to 44.6× lower overhead than extensions of previous solutions with equivalent privacy. $\mathtt{M}^2\mathtt{R}$ adds a total of 17% to 130% overhead over the insecure baseline solution that ignores the leakage channels $\mathtt{M}^2\mathtt{R}$ addresses.

1 Introduction

The threat of data theft in public and private clouds from insiders (e.g. curious administrators) is a serious concern. Encrypting data on the cloud storage is one standard technique which allows users to protect their sensitive data from such insider threats. However, once the data is encrypted, enabling computation on it poses a significant challenge. To enable privacy-preserving computation, a range of security primitives have surfaced recently, including trusted computing support for hardware-isolated computation [2, 5, 38, 40] as well as purely cryptographic techniques [20,21,47]. These primitives show promising ways for running computation securely on a single device running an untrusted software stack. For instance, *trusted computing* primitives can isolate units of computation on an untrusted cloud server. In this approach, the hardware provides a confidential and integrity-protected execution environment to which encryption keys can be made available for decrypting the data before computing on it. Previous works have successfully demonstrated how to securely execute a unit of user-defined computation on an untrusted cloud node, using support from hardware primitives available in commodity CPUs [8, 14, 38, 39, 49] .

In this paper, we study the problem of enabling privacy-preserving *distributed computation* on an untrusted cloud. A sensitive distributed computation task comprises many units of computation which are scheduled to run on a multi-node cluster (or cloud). The input and output data between units of computation are sent over channels controlled by the cloud provisioning system, which may be compromised. We assume that each computation node in the cluster is equipped with a CPU that supports trusted computing primitives (for example, TPMs or Intel SGX). Our goal is to enable a privacy-preserving execution of a distributed computation task. Consequently, we focus on designing privacy in the popular MapReduce framework [17]. However, our techniques can be applied to other distributed dataflow frameworks such as Spark [62], Dryad [26], and epiC [27].

Problem. A MapReduce computation consists of two types of units of computation, namely *map* and *reduce*, each of which takes key-value tuples as input. The MapReduce provisioning platform, for example Hadoop [1], is responsible for scheduling the map/reduce operations for the execution in a cluster and for providing a data channel between them [31]. We aim to achieve a strong level of security in the distributed execution of a MapReduce task (or job) — that is, the adversary learns nothing beyond the execution time and the number of input and output of each computation unit. If we view each unit of computation as one atomic operation of a larger distributed program, the execution can be thought of as running a set of operations on data values passed via a data channel (or a global "RAM") under the adversary's control. That is, our definition of privacy is analogous to the strong level of privacy offered by the well-known oblivious RAM protocol in the monolithic processor case [22].

We assume that the MapReduce provisioning platform is compromised, say running malware on all nodes in the cluster. Our starting point in developing a defense is a baseline system which runs each unit of computation (map/reduce instance) in a hardware-isolated process, as proposed in recent systems [49, 59]. Inputs and outputs of each computation unit are encrypted, thus the adversary observes only encrypted data. While this baseline offers a good starting point, merely encrypting data in-transit between units of computation is not sufficient (see Section 3). For instance, the adversary can observe the pattern of data reads/writes between units. As another example, the adversary can learn the synchronization between map and reduce units due to the scheduling structure of the provisioning platform. Further, the adversary has the ability to duplicate computation, or tamper with the routing of encrypted data to observe variations in the execution of the program.

Challenges. There are several challenges in building a practical system that achieves our model of privacy. First, to execute map or reduce operations on a single computation node, one could run all computation units — including the entire MapReduce platform — in an execution environment that is protected by use of existing trusted computing primitives. However, such a solution would entail little trust given the large TCB, besides being unwieldy to implement. For instance, a standard implementation of the Hadoop stack is over 190K lines of code. The scope of exploit from vulnerabilities in such a TCB is large. Therefore, the first challenge is to enable practical privacy by minimizing the increase in platform TCB and without requiring any algorithmic changes to the original application.

The second challenge is in balancing the needs of privacy and performance. Addressing the leakage channels discussed above using generic methods easily yields a solution with poor practical efficiency. For instance, hiding data read/write patterns between specific map and reduce operations could be achieved by a generic oblivious RAM (ORAM) solution [22, 55]. However, such a solution would introduce a slowdown proportional to polylog in the size of the intermediate data exchange, which could degrade performance by over 100× when gigabytes of data are processed.

Our Approach. We make two observations that enable us to achieve our model of privacy in a MapReduce implementation. First, on a single node, most of the MapReduce codebase can stay outside of the TCB (i.e. code performing I/O and scheduling related tasks). Thus, we design four new components that integrate readily to the existing MapReduce infrastructure. These components which amount to fewer than 500 lines of code are the only pieces of trusted logic that need to be in the TCB, and are run in a protected environment on each computation node. Second, MapReduce computation (and computation in distributed dataflow frameworks in general) has a specific structure of data exchange and execution between map and reduce operations; that is, the map writes the data completely before it is consumed by the reduce. Exploiting this structure, we design a component called *secure shuffler* which achieves the desired security but is much less expensive than a generic ORAM solution, adding only a $O(\log N)$ additive term to the latency, where N is the size of the data.

Results. We have implemented a system called $\mathtt{M}^2\mathtt{R}$ based on Hadoop [1]. We ported 7 applications from a popular big-data benchmarks [25] and evaluated them on a cluster. The results confirm three findings. First, porting MapReduce jobs to $\mathtt{M}^2\mathtt{R}$ requires small development effort: changing less than 45 lines of code. Second, our solution offers a factor of $1.3\times$ to $44.6\times$ (median $11.2\times$) reduction in overhead compared to the existing solutions with equivalent privacy, and a total of $17\% - 130\%$ of overhead over the baseline solution which protects against none of the attacks we focus on in this paper. Our overhead is moderately high, but $\mathtt{M}^2\mathtt{R}$ has high compatibility and is usable with high-sensitivity big data analysis tasks (e.g. in medical, social or financial data analytics). Third, the design is scalable and adds a TCB of less than 0.16% of the original Hadoop codebase.

Contributions. In summary, our work makes three key contributions:

- *Privacy-preserving distributed computation.* We define a new pragmatic level of privacy which can be achieved in the MapReduce framework requiring no algorithmic restructuring of applications.
- *Attacks.* We show that merely encrypting data in enclaved execution (with hardware primitives) is insecure, leading to significant privacy loss.
- *Practical Design.* We design a simple, non-intrusive architectural change to MapReduce. We implement it in a real Hadoop implementation and benchmark its performance cost for privacy-sensitive applications.

2 The Problem

Our goal is to enable privacy-preserving computation for distributed dataflow frameworks. Our current design and

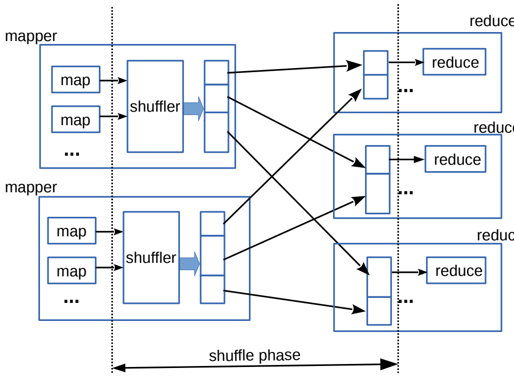


Figure 1: The MapReduce computation model.

implementation are specific to MapReduce framework, the computation structure of which is nevertheless similar to other distributed dataflow engines [26, 27, 62], differing mainly in supported operations.

Background on MapReduce. The MapReduce language enforces a strict structure: the computation task is split into map and reduce operations. Each instance of a map or reduce, called a *computation unit* (or unit), takes a list of key-value tuples[1]. A MapReduce task consists of sequential phases of map and reduce operations. Once the map step is finished, the intermediate tuples are grouped by their key-components. This process of grouping is known as *shuffling*. All tuples belonging to one group are processed by a reduce instance which expects to receive tuples sorted by their key-component. Outputs of the reduce step can be used as inputs for the map step in the next phase, creating a chained MapReduce task. Figure 1 shows the dataflow from the map to the reduce operations via the shuffling step. In the actual implementation, the provisioning of all map units on one cluster node is locally handled by a *mapper* process, and similarly, by a *reducer* process for reduce units.

2.1 Threat Model

The adversary is a malicious insider in the cloud, aiming to subvert the confidentiality of the client's computation running on the MapReduce platform. We assume that the adversary has complete access to the network and storage back-end of the infrastructure and can tamper with the persistent storage or network traffic. For each computation node in the cluster, we assume that the adversary can corrupt the entire software stack, say by installing malware.

We consider an adversary that perpetrates both passive and active attacks. A passive or *honest-but-curious* attacker passively observes the computation session, behaving honestly in relaying data between computation units, but aims to infer sensitive information from the observed data. This is a pragmatic model which includes adversaries that observe data backed up periodically on disk for archival, or have access to performance monitoring interfaces. An active or *malicious* attacker (e.g. an installed malware) can deviate arbitrarily from the expected behavior and tamper with any data under its control. Our work considers both such attacks.

There are at least two direct attacks that an adversary can mount on a MapReduce computation session. First, the adversary can observe data passing between computation units. If the data is left unencrypted, this leads to a direct breach in confidentiality. Second, the adversary can subvert the computation of each map/reduce instance by tampering with its execution. To address these basic threats, we start with a baseline system described below.

Baseline System. We consider the *baseline system* in which each computation unit is hardware-isolated and executed privately. We assume that the baseline system guarantees that the program can only be invoked on its entire input dataset, or else it aborts in its first map phase. Data blocks entering and exiting a computation unit are encrypted with authenticated encryption, and all side-channels from each computation unit are assumed to be masked [51]. Intermediate data is decrypted only in a hardware-attested computation unit, which has limited memory to securely process up to T inputs tuples. Systems achieving this baseline have been previously proposed, based on differing underlying hardware mechanisms. VC^3 is a recent system built on Intel SGX [49].

Note that in this baseline system, the MapReduce provisioning platform is responsible for invoking various trusted units of computation in hardware-isolated processes, passing encrypted data between them. In Section 3, we explain why this baseline system leaks significant information, and subsequently define a stronger privacy objective.

2.2 Problem Definition

Ideally, the distributed execution of the MapReduce program should leak nothing to the adversary, except the total size of the input, total size of the output and the running time. The aforementioned baseline system fails to achieve the ideal privacy. It leaks two types of information: (a) the input and output size, and processing time of individual computation unit, and (b) dataflow among the computation units.

We stress that the leakage of (b) is significant in many applications since it reveals relationships among the input. For instance, in the well-known example of computing Pagerank scores for an encrypted graph [44], flows from a computation unit to another correspond to edges

[1] To avoid confusion of the tuple key with cryptographic key, we refer to the first component in the tuple as *key-component*.

in the input graph. Hence, leaking the dataflow essentially reveals the whole graph edge-structure!

Techniques for hiding or reducing the leakage in (a) by padding the input/output size and introducing timing delays are known [35, 41]. Such measures can often require algorithmic redesign of the application [9] or use of specialized programming languages or hardware [33,34], and can lead to large overheads for applications where the worst case running time is significantly larger than the average case. We leave incorporating these orthogonal defenses out-of-scope.

Instead, in this work, we advocate focusing on eliminating leakage on (b), while providing a formulation that clearly captures the information that might be revealed. We formulate the admissible leakage as Ψ which captures the information (a) mentioned above, namely the input/output size and running time of each trusted computation unit invoked in the system. We formalize this intuition by defining the execution as a protocol between trusted components and the adversary, and define our privacy goal as achieving *privacy modulo-*Ψ.

Execution Protocol. Consider an honest execution of a program on input $I = \langle x_1, x_2, \ldots, x_n \rangle$. For a given map-reduce phase, let there be n map computation units. Let us label the map computation units such that the unit with label i takes x_i as input. Recall that the tuples generated by the map computation units are to be shuffled, and divided into groups according to the key-components. Let K to be the set of unique key-components and let $\pi : [n+1, n+m] \rightarrow K$ be a randomly chosen permutation, where $m = |K|$. Next, m reduce computation units are to be invoked. We label them starting from $n+1$, such that the computation unit i takes tuples with key-component $\pi(i)$ as input.

Let I_i, O_i, T_i be the respective input size (measured by number of tuples), output size, and processing time of the computation unit i, and call $\Psi_i = \langle I_i, O_i, T_i \rangle$ the *IO-profile* of computation unit i. The profile Ψ of the entire execution on input I is the sequence of Ψ_i for all computation units $i \in [1, \ldots, n+m]$ in the execution protocol. If an adversary $\widetilde{\mathscr{A}}$ can initiate the above protocol and observe Ψ, we say that the adversary has access to Ψ.

Now, let us consider the execution of the program on the same input $I = \langle x_1, x_2, \ldots, x_n \rangle$ under a MapReduce provisioning protocol by an adversary $\mathscr{A}$. A semi-honest adversary $\mathscr{A}$ can obtain information on the value of the input, output and processing time of every trusted instance, including information on trusted instances other than the map and reduce computation units. If the adversary is malicious, it can further tamper with the inputs and invocations of the instances. In the protocol, the adversary controls 6 parameters:

(C1) the start time of each computation instance,
(C2) the end time of each instance,
(C3) the encrypted tuples passed to its inputs,
(C4) the number of computation instances,
(C5) order of computation units executed,
(C6) the total number of map-reduce phases executed.

Since the adversary $\mathscr{A}$ can obtain "more" information and tamper the execution, a question to ask is, can the adversary $\mathscr{A}$ gain more knowledge compared to an adversary $\widetilde{\mathscr{A}}$ who only has access to Ψ? Using the standard notions of indistinguishability[2] and adversaries [28], we define a secure protocol as follows:

Definition 1 (**Privacy modulo-Ψ**). *A provisioning protocol for a program is modulo-Ψ private if, for any adversary $\mathscr{A}$ executing the MapReduce protocol, there is a adversary $\widetilde{\mathscr{A}}$ with access only to Ψ, such that the output of $\mathscr{A}$ and $\widetilde{\mathscr{A}}$ are indistinguishable.*

The definition states that the output of the adversaries can be directly seen as deduction made on the information available. The fact that all adversaries have output indistinguishable from the one which knows Ψ suggests that no additional information can be gained by any $\mathscr{A}$ beyond that implied by knowledge of Ψ.

Remarks. First, our definition follows the scenario proposed by Canneti [11], which facilitates universal composition. Hence, if a protocol is private module-Ψ for one map-reduce phase, then an entire sequence of phases executed is private module-Ψ. Note that our proposed M^{2}R consists of a sequence of map, shuffle, and reduce phases where each phase starts only after the previous phase has completed, and the chain of MapReduce jobs are carried out sequentially. Thus, universal composition can be applied. Second, we point out that if the developer restructures the original computation to make the IO-profile the same for all inputs, then Ψ leaks nothing about the input. Therefore, the developer can consider using orthogonal techniques to mask timing latencies [41], hiding trace paths and IO patterns [34] to achieve ideal privacy, if the performance considerations permit so.

2.3 Assumptions

In this work, we make specific assumptions about the baseline system we build upon. First, we assume that the underlying hardware sufficiently protects each computation unit from malware and snooping attacks. The range of threats that are protected against varies based on the underlying trusted computing hardware. For instance,

[2] non-negligible advantage in a distinguishing game

traditional TPMs protect against software-only attacks but not against physical access to RAM via attacks such as cold-boot [24]. More recent trusted computing primitives, such as Intel SGX [40], encrypt physical memory and therefore offer stronger protection against adversaries with direct physical access. Therefore, we do not focus on the specifics of how to protect each computation unit, as it is likely to change with varying hardware platform used in deployment. In fact, our design can be implemented in any virtualization-assisted isolation that protects user-level processes on a malicious guest OS [12, 52, 57], before Intel SGX becomes available on the market.

Second, an important assumption we make is that of information leakage via side-channels (e.g. cache latencies, power) from a computation unit is minimal. Indeed, it is a valid concern and an area of active research. Both software and hardware-based solutions are emerging, but they are orthogonal to our techniques [18, 29].

Finally, to enable arbitrary computation on encrypted data, decryption keys need to be made made available to each hardware-isolated computation unit. This provisioning of client's keys to the cloud requires a set of trusted administrator interfaces and privileged software. We assume that such trusted key provisioning exists, as is shown in recent work [49, 65].

3 Attacks

In this section, we explain why a baseline system that merely encrypts the output of each computation unit leaks significantly more than a system that achieves privacy modulo-Ψ. We explain various subtle attack channels that our solution eliminates, with an example.

Running Example. Let us consider the canonical example of the Wordcount job in MapReduce, wherein the goal is to count the number of occurrences of each word in a set of input files. The map operation takes one file as input, and for each word w in the file, outputs the tuple $\langle w, 1\rangle$. All outputs are encrypted with standard authenticated encryption. Each reduce operation takes as input all the tuples with the same tuple-key, i.e. the same word, and aggregates the values. Hence the output of reduce operations is an encrypted list of tuples $\langle w, w_c\rangle$, where w_c is the frequency of word w for all input files. For simplicity, we assume that the input is a set of files $F = \{F_1, \ldots, F_n\}$, each file has the same number of words and is small enough to be processed by a map operation[3].

What does Privacy modulo-Ψ Achieve? Here all the map computation units output same size tuples, and after grouping, each reduce unit receives tuples grouped by words. The size of map outputs and group sizes constitute Ψ, and a private modulo-Ψ execution therefore leaks some statistical information about the collection of files in aggregate, namely the frequency distribution of words in F. However, it leaks nothing about the contents of words in the individual files — for instance, the frequency of words in any given file, and the common words between any pair of files are not leaked. As we show next, the baseline system permits a lot of inference attacks as it fails to achieve privacy modulo-Ψ. In fact, eliminating the remaining leakage in this example may not be easy, as it may assume apriori knowledge about the probability distribution of words in F (e.g. using differential privacy [48]).

Passive Attacks. Consider a semi-honest adversary that executes the provisioning protocol, but aims to infer additional information. The adversary controls 6 parameters **C1-C6** (Section 2.2) in the execution protocol. The number of units (**C4**) and map-reduce phases executed (**C6**) are dependent (and implied) by Ψ in an honest execution, and do not leak any additional information about the input. However, parameters **C1,C2,C3** and **C5** may directly leak additional information, as explained below.

- *Dataflow Patterns (Channel C3).* Assume that the encrypted tuples are of the same size, and hence do not individually leak anything about the underlying plain text. However, since the adversary constitutes the data communication channel, it can correlate the tuples written out by a map unit and read by a specific reduce unit. In the Wordcount example, the i^{th} map unit processes words in the file F_i, and then the intermediate tuples are sorted before being fed to reduce units. By observing which map outputs are grouped together to the same reduce unit, the adversary can learn that the word w_i in file F_i is the same as a word w_j in file F_j. This is true if they are received by the same reduce unit as one group. Thus, data access patterns leak significant information about overlapping words in files.
- *Order of Execution (Channel C5).* A deterministic order of execution of nodes in any step can reveal information about the underlying tuples beyond what is implied by Ψ. For instance, if the provisioning protocol always sorts tuple-keys and assigns them to reduce units in sorted order, then the adversary learns significant information. In the WordCount example, if the first reduce unit always corresponds to words appearing first in the sorted order, this would leak information about specific words processed by the reduce unit. This is not directly implied by Ψ.
- *Time-of-Access (Channel C1,C2)* Even if data access patterns are eliminated, time-of-access is an-

[3] Files can be processed in fixed size blocks, so this assumption is without any loss of generality

other channel of leakage. For instance, an optimizing scheduler may start to move tuples to the reduce units even before the map step is completed (pipelining) to gain efficiency. In such cases, the adversary can correlate which blocks written by map units are read by which reduce units. If outputs of all but the i^{th} map unit are delayed, and the j^{th} reduce unit completes, then the adversary can deduce that there is no dataflow from the i^{th} map unit to j^{th} reduce unit. In general, if computation units in a subsequent step do not synchronize to obtain outputs from all units in the previous step, the time of start and completion leaks information.

Active Attacks. While we allow the adversary to abort the computation session at any time, we aim to prevent the adversary from using active attacks to gain advantage in breaking confidentiality. We remind readers that in our baseline system, the adversary can only invoke the program with its complete input set, without tampering with any original inputs. The output tuple-set of each computation unit is encrypted with an authenticated encryption scheme, so the adversary cannot tamper with individual tuples. Despite these preliminary defenses, several channels for active attacks exist:

- *Tuple Tampering*. The adversary may attempt to duplicate or eliminate an entire output tuple-set produced by a computation unit, even though it cannot forge individual tuples. As an attack illustration, suppose the adversary wants to learn how many words are unique to an input file F_i. To do this, the adversary can simply drop the output of the i^{th} map unit. If the number of tuples in the final output reduces by k, the tuples eliminated correspond to k unique words in F_i.
- *Misrouting Tuples*. The adversary can reorder intermediate tuples or route data blocks intended for one reduce unit to another. These attacks subvert our confidentiality goals. For instance, the adversary can bypass the shuffler altogether and route the output of i^{th} map unit to a reduce unit. The output of this reduce unit leaks the number of unique words in F_i. Similar inference attacks can be achieved by duplicating outputs of tuples in the reduce unit and observing the result.

4 Design

Our goal is to design a MapReduce provisioning protocol which is private modulo-Ψ and adds a small amount of the TCB to the existing MapReduce platform. We explain the design choices available and our observations that lead to an efficient and clean security design.

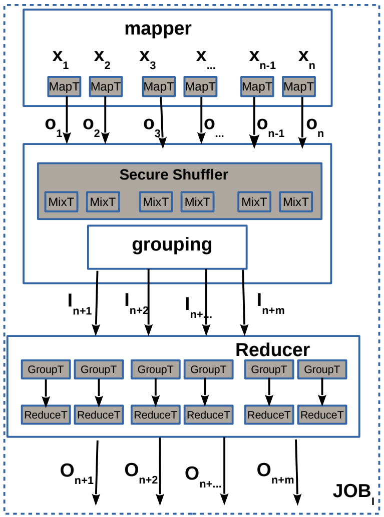


Figure 2: The data flow in $\mathtt{M}^2\mathtt{R}$. Filled components are trusted. Input, intermediate and output tuples are encrypted. The original map and reduce operations are replaced with `mapT` and `reduceT`. New components are the mixer nodes which use `mixT`, and another trusted component called `groupT`.

4.1 Architecture Overview

The computation proceeds in phases, each consisting of a map step, a shuffle step, and a reduce step. Figure 2 depicts the 4 new trusted components our design introduces into the dataflow pipeline of MapReduce. These four new TCB components are `mapT`, `reduceT`, `mixT` and `groupT`. Two of these correspond to the execution of map and reduce unit. They ensure that output tuples from the map and reduce units are encrypted and each tuple is of the same size. The other 2 components implement the critical role of *secure shuffling*. We explain our non-intrusive mechanism for secure shuffling in Section 4.2. Further, all integrity checks to defeat active attacks are designed to be distributed requiring minimal global synchronization. The shuffler in the MapReduce platform is responsible for grouping tuples, and invoking reduce units on disjoint ranges of tuple-keys. On each cluster node, the reducer checks the grouped order and the expected range of tuples received using the trusted `groupT` component. The outputs of the reduce units are then fed back into the next round of map-reduce phase.

Minimizing TCB. In our design, a major part of the MapReduce's software stack deals with job scheduling and I/O operations, hence it can be left outside of the TCB. Our design makes no change to the grouping and scheduling algorithms, and they are outside our TCB as shown in the Figure 2. Therefore, the design is conceptually simple and requires no intrusive changes to be implemented over existing MapReduce implementations. Developers need to modify their original applications to prepare them for execution in a hardware-protected process in our baseline system, as proposed in previous systems [38, 39, 49]. Beyond this modification made by the baseline system to the original MapReduce, M^{2}R requires a few additional lines of code to invoke the new privacy-enhancing TCB components. That is, MapReduce applications need modifications only to invoke components in our TCB. Next, we explain how our architecture achieves privacy and integrity in a MapReduce execution, along with the design of these four TCB components.

4.2 Privacy-Preserving Execution

For any given execution, we wish to ensure that each computation step in a phase is private modulo-Ψ. If the map step, the shuffle step, and the reduce step are individually private modulo-Ψ, by the property of serial composibility, the entire phase and a sequence of phases can be shown to be private. We discuss the design of these steps in this section, assuming a honest-but-curious adversary limited to passive attacks. The case of malicious adversaries is discussed in Section 4.3.

4.2.1 Secure Shuffling

As discussed in the previous section, the key challenge is performing secure shuffling. Consider the naive approach in which we simply move the entire shuffler into the platform TCB of each cluster node. To see why this is insecure, consider the grouping step of the shuffler, often implemented as a distributed sort or hash-based grouping algorithm. The grouping algorithm can only process a limited number of tuples locally at each mapper, so access to intermediate tuples must go to the network during the grouping process. Here, network data access patterns from the shuffler leak information. For example, if the shuffler were implemented using a standard merge sort implementation, the merge step leaks the relative position of the pointers in sorted sub-arrays as it fetches parts of each sub-array from network incrementally[4].

One generic solution to hide data access patterns is to employ an ORAM protocol when communicating with the untrusted storage backend. The grouping step will then access data obliviously, thereby hiding all correlations between grouped tuples. This solution achieves strong privacy, but with an overhead of $O(\log^k N)$ for each access when the total number of tuples is N [55]. Advanced techniques can be employed to reduce the overhead to $O(\log N)$, i.e. $k = 1$ [43]. Nevertheless, using a sorting algorithm for grouping, the total overhead becomes $O(N \log^{k+1} N)$, which translates to a factor of $30-100\times$ slowdown when processing gigabytes of shuffled data.

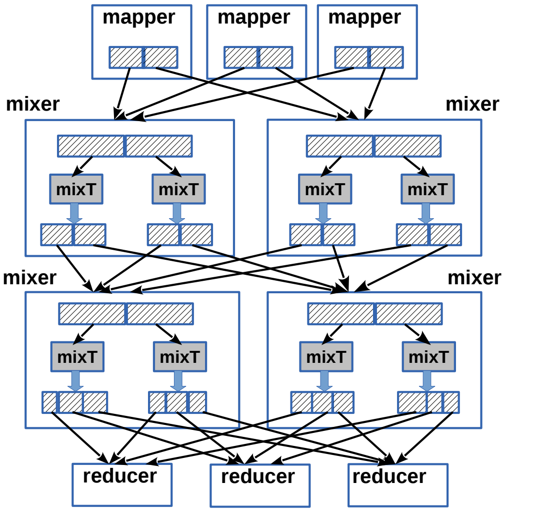


Figure 3: High level overview of the map-mix-reduce execution using a 2-round mix network.

A more advanced solution is to perform oblivious sorting using sorting networks, for example, odd-even or bitonic sorting network [23]. Such an approach hides data access patterns, but admits a $O(\log^2 N)$ latency (additive only). However, sorting networks are often designed for a fixed number of small inputs and hard to adapt to tens of gigabytes of distributed data.

We make a simple observation which yields a non-intrusive solution. Our main observation is that in MapReduce and other dataflow frameworks, the sequence of data access patterns is fixed: it consists of cycles of tuple writes followed by reads. The reduce units start reading and processing their inputs only after the map units have finished. In our solution, we rewrite intermediate encrypted tuples with re-randomized tuple keys such that there is no linkability between the re-randomized tuples and the original encrypted map output tuples. We observe that this step can be realized by secure mix networks [30]. The privacy of the computation reduces directly to the problem of secure mixing. The total latency added by our solution is an additive term of $O(\log N)$ in the worst case. Since MapReduce shuffle step is based on sorting which already admits $O(N \log N)$ overhead, our design retains the asymptotic runtime complexity of the original framework.

Our design achieves privacy using a cascaded mix network (or cascaded-mix) to securely shuffle tuples [30]. The procedure consists of a cascading of κ intermediate steps, as shown in Figure 3. It has κ identical steps (called mixing steps) each employing a number of trusted computation units called `mixT` units, the execution of which can be distributed over multiple nodes called mixers. Each `mixT` takes a fixed amount of T tuples that it can process in memory, and passes exactly the same number of encrypted tuples to all `mixT` units in the sub-

[4] This can reveal, for instance, whether the first sub-array is strictly lesser than the first element in the second sorted sub-array.

sequent step. Therefore, in each step of the cascade, the mixer utilizes N/T mixT units for mixing N tuples. At $\kappa = \log \frac{N}{T}$, the network ensures the strongest possible unlinkability, that is, the output distribution is statistically indistinguishable from a random distribution [30].

Each mixT unit decrypts the tuples it receives from the previous step, randomly permutes them using a linear-time algorithm and re-encrypts the permuted tuples with fresh randomly chosen symmetric key. These keys are known only to mixT units, and can be derived using a secure key-derivation function from a common secret. The processing time of mixT are padded to a constant. Note that the re-encryption time has low variance over different inputs, therefore such padding incurs low overhead.

Let Ω represents the number of input and output tuples of cascaded-mix with κ steps. Intuitively, when κ is sufficiently large, an semi-honest adversary who has observed the execution does not gain more knowledge than Ω. The following lemma states that indeed this is the case. We present the proof in Appendix A.

Lemma 1. *Cascaded-mix is private module-Ω under semi-honest adversary, given that the underlying encryption scheme is semantically secure.*

4.2.2 Secure Grouping

After the mixing step, the shuffler can group the randomized tuple keys using its original (unmodified) grouping algorithm, which is not in the TCB. The output of the cascaded-mix is thus fed into the existing grouping algorithm of MapReduce, which combines all tuples with the same tuple-key and forward them to reducers. Readers will notice that if the outputs of the last step of the cascaded-mix are probabilistically encrypted, this grouping step would need to be done in a trusted component. In our design, we add a last $(\kappa+1)$-th step in the cascade to accommodate the requirement for subsequent grouping. The last step in the cascade uses a deterministic symmetric encryption F_s, with a secret key s, to encrypt the key-component of the final output tuples. Specifically, the $\langle a,b \rangle$ is encrypted to a ciphertext of the form $\langle F_s(a), E(a,b) \rangle$, where $E(\cdot)$ is a probabilistic encryption scheme. This ensures that the two shuffled tuples with the same tuple-keys have the same ciphertext for the key-component of the tuple, and hence the subsequent grouping algorithm can group them without decrypting the tuples. The secret key s is randomized in each invocation of the cascaded-mix, thereby randomizing the ciphertexts across two map-reduce phases or jobs.

What the adversary gains by observing the last step of mixing is the tuples groups which are permuted using $F_s(\cdot)$. Thus, if $F_s(\cdot)$ is a pseudorandom function family, the adversary can only learn about the size of each group, which is already implied by Ψ. Putting it all together with the Lemma 1, we have:

Theorem 1. *The protocol M^{2}R is modulo-Ψ private (under semi-honest adversary), assuming that the underlying private-key encryption is semantically secure, and $F_s(\cdot)$ is a pseudorandom function family.*

4.3 Execution Integrity

So far, we have considered the privacy of the protocol against honest-but-curious adversaries. However, a malicious adversary can deviate arbitrarily from the protocol by mounting active attacks using the 6 parameters under its control. In this section, we explain the techniques necessary to prevent active attacks.

The program execution in M^{2}R can be viewed as a directed acyclic graph (DAG), where vertices denote trusted computation units and edges denote the flow of encrypted data blocks. M^{2}R has 4 kinds of trusted computation units or vertices in the DAG: mapT, mixT, groupT, and reduceT. At a high-level, our integrity-checking mechanism works by ensuring that nodes at the j^{th} level (by topologically sorted order) check the consistency of the execution at level $j-1$. If they detect that the adversary deviates or tampers with the execution or outputs from level $j-1$, then they abort the execution.

The MapReduce provisioning system is responsible for invoking trusted computation units, and is free to decide the total number of units spawned at each level j. We do not restrict the MapReduce scheduling algorithm to decide which tuples are processed by which reduce unit, and their allocation to nodes in the cluster. However, we ensure that all tuples output at level $i-1$ are processed at level i, and there is no duplicate. Note that this requirement ensures that a computation in step i starts only after outputs of previous step are passed to it, implicitly synchronizes the start of the computation units at step i. Under this constraint, it can be shown that channels **C1-C2** (start-end time of each computation node) can only allow the adversary to delay an entire step, or distinguish the outputs of units within one step, which is already implied by Ψ. We omit a detailed proof in this paper. Using these facts, we can show that the malicious adversary has no additional advantage compared to an honest-but-curious adversary, stated formally below.

Theorem 2. *The protocol M^{2}R is private modulo-Ψ under malicious adversary, assuming that the underlying authenticated-encryption is semantically secure (confidentiality) and secure under chosen message attack (integrity), and $F_s(\cdot)$ is a pseudorandom function family.*

Proof Sketch: Given a malicious adversary $\mathscr{A}$ that executes the M^{2}R protocol, we can construct an adversary

$\widetilde{\mathscr{A}}$ that simulates $\mathscr{A}$, but only has access to Ψ in the following way. To simulate $\mathscr{A}$, the adversary $\widetilde{\mathscr{A}}$ needs to fill in information not present in Ψ. For the output of a trusted unit, the simulation simply fills in random tuples, where the number of tuples is derived from Ψ. The timing information can likewise be filled-in. Whenever $\mathscr{A}$ deviates from the protocol and feeds a different input to a trusted instance, the simulation expects the instance will halt and fills in the information accordingly. Note that the input to $\mathscr{A}$ and the input constructed for the simulator $\widetilde{\mathscr{A}}$ could have the same DAG of program execution, although the encrypted tuples are different. Suppose there is a distinguisher that distinguishes $\mathscr{A}$ and $\widetilde{\mathscr{A}}$, let us consider the two cases: either the two DAG's are the same or different. If there is a non-negligible probability that they are the same, then we can construct a distinguisher to contradict the security of the encryption, or $F_s(\cdot)$. If there is a non-negligible probability that they are different, we can forge a valid authentication tag. Hence, the outputs of $\mathscr{A}$ and $\widetilde{\mathscr{A}}$ are indistinguishable. ■

Integrity Checks. Nearly all our integrity checks can be distributed across the cluster, with checking of invariants done locally at each trusted computation. Therefore, our integrity checking mechanism can largely bundle the integrity metadata with the original data. No global synchronization is necessary, except for the case of the `groupT` units as they consume data output by an untrusted grouping step. The `groupT` checks ensure that the ordering of the grouped tuples received by the designated `reduceT` is preserved. In addition, `groupT` units synchronize to ensure that each reducer processes a distinct range of tuple-keys, and that all the tuple-keys are processed by at least one of the reduce units.

4.3.1 Mechanisms

In the DAG corresponding to a program execution, the MapReduce provisioning system assigns unique *instance ids*. Let the vertex i at the level j has the designated id (i, j), and the total number of units at level j be $|V_j|$. When a computation instance is spawned, its designed instance id (i, j) and the total number of units $|V_j|$ are passed as auxiliary input parameters by the provisioning system. Each vertex with id (i, j) is an operation of type `mapT`, `groupT`, `mixT` or `reduceT`, denoted by the function $OpType(i, j)$. The basic mechanism for integrity-checking consists of each vertex emitting a *tagged-block* as output which can be checked by trusted components in the next stage. Specifically, the tagged block is 6-tuple $B = \langle O, LvlCnt, SrcID, DstID, DstLvl, DstType \rangle$, where:

O	is the encrypted output tuple-set,
$LvlCnt$	is the number of units at source level,
$SrcID$	is the instance id of the source vertex,
$DstID$	is instance id of destination vertex or `NULL`
$DstLvl$	is the level of the destination vertex,
$DstType$	is the destination operation type.

In our design, each vertex with id (i, j) fetches the tagged-blocks from all vertices at the previous level, denoted by the multiset $\mathscr{B}$, and performs the following consistency checks on $\mathscr{B}$:

1. The LvlCnt for all $b \in \mathscr{B}$ are the same (say $\ell(\mathscr{B})$).
2. The SrcID for all $b \in \mathscr{B}$ are distinct.
3. For set $S = \{SrcID(b) \mid b \in \mathscr{B}\}$, $|S| = \ell(\mathscr{B})$.
4. For all $b \in \mathscr{B}$, $DstLvl(b) = j$.
5. For all $b \in \mathscr{B}$, $DstID(b) = (i, j)$ or `NULL`.
6. For all $b \in \mathscr{B}$, $DstType(b) = OpType(i, j)$.

Conditions 1,2 and 3 ensure that tagged-blocks from *all* units in the previous level are read and that they are distinct. Thus, the adversary has not dropped or duplicated any output tuple. Condition 4 ensures that the computation nodes are ordered sequentially, that is, the adversary cannot misroute data bypassing certain levels. Condition 6 further checks that execution progresses in the expected order — for instance, the map step is followed by a mix, subsequently followed by a group step, and so on. We explain how each vertex decides the right or expected order independently later in this section. Condition 5 states that if the source vertex wishes to fix the recipient id of a tagged-block, it can verifiably enforce it by setting it to non-NULL value.

Each tagged-block is encrypted with standard authenticated encryption, protecting the integrity of all metadata in it. We explain next how each trusted computation vertex encodes the tagged-block.

Map-to-Mix DataFlow. Each `mixT` reads the output metadata of all `mapT`. Thus, each `mixT` knows the total number of tuples N generated in the entire map step, by summing up the counts of encrypted tuples received. From this, each `mixT` independently determines the total number of mixers in the system as N/T. Note that T is the pre-configured number of tuples that each `mixT` can process securely without invoking disk accesses, typically a 100M of tuples. Therefore, this step is completely decentralized and requires no co-ordination between `mixT` units. A `mapT` unit invoked with id (i, j) simply emit tagged-blocks, with the following structure: $\langle \cdot, |V_j|, (i, j), \texttt{NULL}, j+1, \texttt{mixT} \rangle$.

Mix-to-Mix DataFlow. Each `mixT` re-encrypts and permutes a fixed number (T) of tuples. In a κ-step cascaded mix network, at any step s ($s < \kappa - 1$) the `mixT` outputs T/m tuples to each one of the m `mixT` units in the step $s+1$. To ensure this, each mixT adds metadata to its tagged-block output so that it reaches only the specified mixT unit for the next stage. To do so, we use the *DstType* field, which is set to type $\mathtt{mixT}_{s+1}$ by the mixer at step s. Thus, each mixT node knows the total number of tuples being shuffled N (encoded in OpType), its step number in the cascaded mix, and the public value T. From this each mixT can determine the correct number of cascade steps to perform, and can abort the execution if the adversary tries to avoid any step of the mixing.

Mix-to-Group DataFlow. In our design, the last mix step (i,j) writes the original tuple as $\langle F_s(k),(k,v,tctr)\rangle$, where the second part of this tuple is protected with authenticated-encryption. The value $tctr$ is called a tuple-counter, which makes each tuple globally distinct in the job. Specifically, it encodes the value (i,j,ctr) where ctr is a counter unique to the instance (i,j). The assumption here is that all such output tuples will be grouped by the first component, and each group will be forwarded to reducers with no duplicates. To ensure that the outputs received are correctly ordered and untampered, the last mixT nodes send a special tagged-block to groupT nodes. This tagged-block contains the count of tuples corresponding to $F_s(k)$ generated by mixT unit with id (i,j). With this information each groupT node can locally check that:

- For each received group corresponding to $g = F_s(k)$, the count of distinct tuples $(k,\cdot,i,j,ctr)$ it receives tallies with that specified in the tagged-block received from mixT node (i,j), for all blocks in $\mathscr{B}$.

Finally, groupT units need to synchronize to check if there is any overlap between tuple-key ranges. This requires an additional exchange of tokens between groupT units containing the range of group keys and tuple-counters that each unit processes.

Group-to-Reduce Dataflow. There is a one-to-one mapping between groupT units and reduceT units, where the former checks the correctness of the tuple group before forwarding to the designated reduceT. This communication is analogous to that between mixT units, so we omit a detailed description for brevity.

5 Implementation

Baseline Setup. The design of M^{2}R can be implemented differently depending on the underlying architectural primitives available. For instance, we could implement our solution using Intel SGX, using the mechanisms of VC^3 to achieve our baseline. However, Intel SGX is not yet available in shipping CPUs, therefore we use a trusted-hypervisor approach to implement the baseline system, which minimizes the performance overheads from the baseline system. We use Intel TXT to securely boot a trusted Xen-4.4.3 hypervisor kernel, ensuring its static boot integrity[5]. The inputs ands output of map and reduce units are encrypted with AES-GCM using 256-bit keys. The original Hadoop jobs are executed as user-level processes in ring-3, attested at launch by the hypervisor, making an assumption that they are protected during subsequent execution. The MapReduce jobs are modified to call into our TCB components implemented as x86 code, which can be compiled with SFI constraints for additional safety. The hypervisor loads, verifies and executes the TCB components within its address space in ring-0. The rest of Hadoop stack runs in ring 3 and invokes the units by making hypercalls. Note that the TCB components can be isolated as user-level processes in the future, but this is only meaningful if the processes are protected by stronger solutions such as Intel SGX or other systems [12, 14, 52].

M^{2}R TCB. Our main contributions are beyond the baseline system. We add four new components to the TCB of the baseline system. We have modified a standard Hadoop implementation to invoke the mixT and groupT units before and after the grouping step. These two components add a total 90 LoC to the platform TCB. No changes are necessary to the original grouping algorithm. Each mapT and reduceT implement the trusted map and reduce operation — same as in the baseline system. They are compiled together with a static utility code which is responsible for (a) padding each tuple to a fixed size, (b) encrypting tuples with authenticated encryption, (c) adding and verifying the metadata for tagged-blocks, and (d) recording the instance id for each unit. Most of these changes are fairly straightforward to implement. To execute an application, the client encrypts and uploads all the data to M^{2}R nodes. The user then submits M^{2}R applications and finally decrypts the results.

6 Evaluation

This section describes M^{2}R performance in a small cluster under real workloads. We ported 7 data intensive jobs from the standard benchmark to M^{2}R, making less than 25% changes in number of lines of code (LoC) to the original Hadoop jobs. The applications add fewer than 500 LoC into the TCB, or less than 0.16% of the entire Hadoop software stack. M^{2}R adds $17-130\%$ overhead in running time to the baseline system. We also compare M^{2}R with another system offering the same level of privacy, in which encrypted tuples are sent back to a trusted client. We show that M^{2}R is up to 44.6× faster compared

[5] Other hypervisor solutions such as TrustVisor [39], Overshadow [14], Nova [56], SecVisor [50] could equivalently be used

Job	*LoC changed (vs. Hadoop job)*	*TCB increase (vs. Hadoop codebase)*	*Input size (vs. plaintext size)*	*Shuffled bytes*	*# App hypercalls*	*# Platform hypercall*
Wordcount	10 (15%)	370 (0.14%)	2.1G (1.06×)	4.2G	3277173	35
Index	28 (24%)	370 (0.14%)	2.5G (1.15×)	8G	3277173	59
Grep	13 (13%)	355 (0.13%)	2.1G (1.06×)	75M	3277174	10
Aggregate	16 (18%)	395 (0.15%)	2G (1.19×)	289M	18121377	12
Join	30 (22%)	478 (0.16%)	2G (1.19×)	450M	11010647	14
Pagerank	42 (20%)	429 (0.15%)	2.5G (4×)	2.6G	1750000	21
KMeans	113 (7%)	400 (0.12%)	1G (1.09×)	11K	12000064	8

Table 1: Summary of the porting effort and TCB increase for various M^{2}R applications, and the application runtime cost factors. Number of app hypercalls consists of both `mapT` and `reduceT` invocations. Number of platform hypercalls include `groupT` and `mixT` invocations.

Job	*Baseline (vs. no encryption)*	M^{2}R *(% increase vs. baseline)*	*Download-and-compute (× M^{2}R)*
Wordcount	570 (221)	1156 (100%)	1859 (1.6×)
Index	666 (423)	1549 (130%)	2061 (1.3×)
Grep	70 (48)	106 (50%)	1686 (15.9×)
Aggregate	125 (80)	205 (64%)	9140 (44.6×)
Join	422 (211)	510 (20%)	5716 (11.2×)
Pagerank	521 (334)	755 (44%)	1209 (1.6×)
KMeans	123 (71)	145 (17%)	6071 (41.9×)

Table 2: Overall running time (s) of M^{2}R applications in comparison with other systems: (1) the baseline system protecting computation only in single nodes, (2) the download-and-compute system which does not use trusted primitives but instead sends the encrypted tuples back to trusted servers when homomorphic encrypted computation is not possible [59].

to this solution.

6.1 Setup & Benchmarks

We select a standard benchmark for evaluating Hadoop under large workloads called HiBench suite [25]. The 7 benchmark applications, listed in Table 1, cover a wide range of data-intensive tasks: compute intensive (KMeans, Grep, Pagerank), shuffle intensive (Wordcount, Index), database queries (Join, Aggregate), and iterative (KMeans, Pagerank). The size of the encrypted input data is between 1 GB and 2.5 GB in these case studies. Different applications have different amount of shuffled data, ranging from small sizes (75MB in Grep, 11K in KMeans) to large sizes (4.2GB in Wordcount, 8GB in Index).

Our implementation uses the Xen-4.3.3 64-bit hypervisor compiled with trusted boot option. The rest of M^{2}R stack runs on Ubuntu 13.04 64-bit version. We conduct our experiments in a cluster of commodity servers equipped with 1 quad-core Intel CPU 1.8GHz, 1TB hard drive, 8GB RAM and 1GB Ethernet cards. We vary our setup to have between 1 to 4 compute nodes (running mappers and reducers) and between 1 to 4 mixer nodes for implementing a 2-step cascaded mix network. The results presented below are from running with 4 compute nodes and 4 mixers each reserving a 100MB buffer for mixing, averaged over 10 executions.

6.2 Results: Performance

Overheads & Cost Breakdown. We observe a linear scale-up with the number of nodes in the cluster, which confirms the scalability of M^{2}R. In our benchmarks (Table 2), we observe a total overhead of between $17\% - 130\%$ over the baseline system that simply encrypts inputs and outputs of map/reduce units, and utilizes none of our privacy-enhancing techniques. It can also be seen that in all applications except for Grep and KMeans, running time is proportional to the size of data transferred during shuffling (shuffled bytes column in Table 1). To understand the cost factors contributing to the overhead, we measure the time taken by the secure shuffler, by the `mapT` and `reduceT` units, and by the rest of the Hadoop system which comprises the time spent on I/O, scheduling and other book-keeping tasks. This relative cost breakdown is detailed in Figure 4. From the result, we observe that the cost of the secure shuffler is significant. Therefore, reducing the overheads of shuffling, by avoiding the generic ORAM solution, is well-incentivized and is critical to reducing the overall overheads. The two main benchmarks which have high overheads of over 100%, namely Wordcount and Index, incur this cost primarily due to the cost of privacy-preserving shuffling a large amount of data. In benchmarks where the shuffled data is small (Grep, KMeans), the use of `mapT`/`reduceT` adds relatively larger overheads than that from the secure shuffler. The second observation is that the total cost of the both shuffler and other trusted components is comparable to that of Hadoop, which provides evidence that M^{2}R preserves the asymptotic complexity of Hadoop.

Comparison to Previous Solutions. Apart from the baseline system, a second point of comparison are previously proposed systems that send encrypted tuples to the user for private computation. Systems such as Monomi [59] and AutoCrypt [58] employ homomorphic encryption for computing on encrypted data on the single servers. For operations that cannot be done on the server using partially homomorphic encryption, such Monomi-like systems forward the data to a trusted set of servers (or to the client's private cloud) for decryption. We refer to this approach as download-and-compute approach. We estimate the performance of a Monomi-like system extended to distributed computation tasks, for achieving privacy equivalent to ours. To compare, we assume that the system uses Paillier, ElGamal and randomized

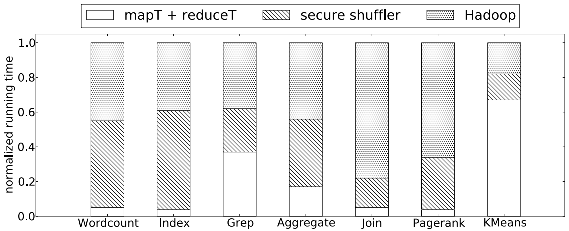


Figure 4: Normalized break-down time for M^{2}R applications. The running time consists of the time taken by mapT and reduceT, plus the time by the secure shuffler. The rest comes from the Hadoop runtime.

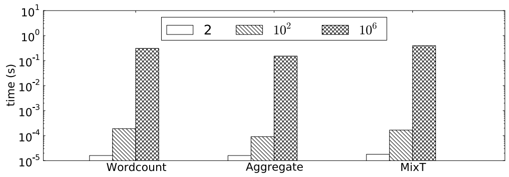


Figure 5: Cost of executing mapT instance of the Wordcount and Aggregate job, and the cost for executing mixT. Input sizes (number of ciphertexts per input) varies from 2 to 10^6.

search schemes for homomorphic computation, but not OPE or deterministic schemes (since that leaks more than M^{2}R and our baseline system do). We run operations that would fall outside such the expressiveness of the allowed homomorphic operations, including shuffling, as a separate network request to the trusted client. We batch network requests into one per MapReduce step. We assume that the network round trip latency to the client is only 1ms — an optimistic approximation since the average round trip delay in the same data center is $10-100$ms [4, 61]. We find that this download-and-compute approach is slower compared to ours by a factor of 1.3× to 44.6× (Table 2), with the median benchmark running slower by 11.2×. The overheads are low for case-studies where most of the computation can be handled by homomorphic operations, but most of the benchmarks require conversions between homomorphic schemes (thereby requiring decryption) [58, 59] or computation on plaintext values.

Platform-Specific Costs. Readers may wonder if the evaluation results are significantly affected by the choice of our implementation platform. We find that the dominant costs we report here are largely complementary to the costs incurred by the specifics of the underlying platform. We conduct a micro-benchmark to evaluate the cost of context-switches and the total time spent in the trusted components to explain this aspect. In our platform, the cost of each hypercall (switch to trusted logic) is small ($13\mu s$), and the execution of each trusted component is largely proportional to the size of its input data as shown in Figure 5. The time taken by the trusted computation grows near linearly with the input data-size, showing that the constant overheads of context-switches and other platform's specifics do not contribute to the reported results significantly. This implies that simple optimizations such as batching multiple trusted code invocations would not yield any significant improvements, since the overheads are indeed proportional to the total size of data and not the number of invocations. The total number of invocations (via hypercalls) for app-specific trusted logic (mapT, reduceT) is proportional to the total number input tuples, which amounts for less than half a second overhead even for millions of input tuples. The number of invocations to the other components (mixT and groupT) is much smaller ($8-59$) and the each invocation operates on large inputs of a few gigabytes; therefore the dominant cost is not that of context-switches, but that of the cost of multi-step shuffling operation itself and the I/O overheads.

6.3 Results: Security & Porting Effort

Porting effort. We find that the effort to adapt all benchmarks to M^{2}R is modest at best. For each benchmark, we report the number of Java LoC we changed in order to invoke the trusted components in M^{2}R, measured using the sloccount tool [6]. Table 1 shows that all applications except for KMeans need to change fewer than 43 LoC. Most changes are from data marshaling before and after invoking the mapT and reduceT units. KMeans is more complex as it is a part of the Mahout distribution and depends on many other utility classes. Despite this, the change is only 113 LoC, or merely 7% of the original KMeans implementation.

TCB increase. We define our *TCB increase* as the total size of the four trusted components. This represents the additional code running on top of a base TCB, which in our case is Xen. Note that our design can eliminate the base TCB altogether in the future by using SGX enclaves, and only retain the main trusted components we propose in M^{2}R. The TCB increase comprises the per-application trusted code and platform trusted code. The former consists of the code for loading and executing mapT, reduceT units (213 LoC) as well as the code for implementing their logic. Each map/reduce codebase itself is small, fewer than 200 LoC, and runs as trusted components in the baseline system itself. The platform trusted code includes that of mixT and groupT, which amounts to 90 LoC altogether. The entire Hadoop software stack is over 190*K* LoC and M^{2}R avoids moving all of it into the TCB. Table 1 shows that all jobs have TCB increases of fewer than 500 LoC, merely 0.16% of the Hadoop codebase.

Security. M^{2}R achieves stronger privacy than previous

[6] http://www.dwheeler.com/sloccount

Job	M^{2}R	*Baseline (additional leakage)*
Wordcount	# unique words + count	word-file relationship
Index	# unique words + count	word-file relationship
Grep	nothing	nothing
Aggregate	# groups + group size	record-group relationship
Join	# groups + group size	record-group relationship
Pagerank	node in-degree	whole input graph
KMeans	nothing	nothing

Table 3: Remaining leakage of M^{2}R applications, compared with that in the baseline system.

platforms that propose to use encrypted computation for big-data analysis. Our definition allows the adversary to observe an admissible amount of information, captured by Ψ, in the computation but hides everything else. It is possible to quantitatively analyze the increased privacy in information-theoretic terms, by assuming the probability distribution of input data [37, 53]. However, here we present a qualitative description in Table 3 highlighting how much privacy is gained by the techniques introduced in M^{2}R over the baseline system. For instance, consider the two case studies that incur most performance overhead (Wordcount, Index). In these examples, merely encrypting the map/reduce tuples leaks information about which file contains which words. This may allow adversaries to learn the specific keywords in each file in the dataset. In M^{2}R, this leakage is reduced to learning only the total number of unique words in the complete database and the counts of each, hiding information about individual files. Similarly, M^{2}R hides which records are in which group for database operations (Aggregate and Join). For Pagerank, the baseline system leaks the complete input graph edge structure, giving away which pair of nodes has an edge, whereas M^{2}R reduces this leakage to only the in-degree of graph vertices. In the two remaining case studies, M^{2}R provides no additional benefit over the baseline.

7 Related Work

Privacy-preserving data processing. One of M^{2}R's goal is to offer large-scale data processing in a privacy preserving manner on untrusted clouds. Most systems with this capability are in the database domain, i.e. supporting SQL queries processing. CryptDB [47] takes a purely cryptographic approach, showing the practicality of using partially homomorphic encryption schemes [3, 15, 45, 46, 54]. CryptDB can only work on a small set of SQL queries and therefore is unable to support arbitrary computation. Monomi [59] supports more complex queries, by adopting the download-and-compute approach for complex queries. As shown in our evaluation, such an approach incurs an order of magnitude larger overheads.

There exist alternatives supporting outsourcing of query processing to a third party via server-side trusted hardware, e.g. IBM 4764/5 cryptographic co-processors. TrustedDB [7] demonstrated that a secure outsourced database solution can be built and run at a fraction of the monetary cost of any cryptography-enabled private data processing. However, the system requires expensive hardware and a large TCB which includes the entire SQL server stack. Cipherbase improves upon TrustedDB by considering encrypting data with partially homomorphic schemes, and by introducing a trusted entity for query optimization [6]. M^{2}R differs to these systems in two fundamental aspects. First, it supports general computation on any type of data, as opposed to being restricted to SQL and structured database semantics. Second, and more importantly, M^{2}R provides confidentiality in a distributed execution environment which introduces more threats than in a single-machine environment.

VC^3 is a recent system offering privacy-preserving general-purpose data processing [49]. It considers MapReduce and utilizes Intel SGX to maintain a small TCB. This system is complementary to M^{2}R, as it focuses on techniques for isolated computation, key management, etc. which we do not consider. The privacy model in our system is stronger than that of VC^3 which does not consider traffic analysis attacks.

GraphSC offers a similar security guarantee to that of M^{2}R for specialized graph-processing tasks [42]. It provides a graph-based programming model similar to GraphLab's [36], as opposed to the dataflow model exposed by M^{2}R. GraphSC does not employ trusted primitives, but it assumes two non-colluding parties. There are two main techniques for ensuring data-oblivious and secure computation in GraphSC: sorting and garbled circuits. However, these techniques result in large performance overheads: a small Pagerank job in GraphSC is $200,000\times - 500,000\times$ slower than in GraphLab without security. M^{2}R achieves an overhead of $2\times - 5\times$ increase in running time because it leverages trusted primitives for computation on encrypted data. A direct comparison of oblivious sorting used therein instead of our secure shuffler is a promising future work.

Techniques for isolated computation. The current implementation of M^{2}R uses a trusted hypervisor based on Xen for isolated computation. Overshadow [14] and CloudVisor [63] are techniques with large TCB, whereas Flicker [38] and TrustVisor [39] reduce the TCB at the cost of performance. Recently, Minibox [32] enhances a TrustVisor-like hypervisor with two-way protection providing security for both the OS and the applications (or PALs). Advanced hardware-based techniques include Intel SGX [40] and Bastion [12] provide a hardware protected secure mode in which applications can be executed at hardware speed. All these techniques are complementary to ours.

Mix networks. The concept of mix network is first described in the design of untraceable electronic mail [13]. Since then, a body of research has concentrated on building, analyzing and attacking anonymous communication systems [16, 19]. Canetti presents the first definition of security that is preserved under composition [11], from which others have shown that the mix network is secure under Canetti's framework [10, 60]. Security properties of cascaded mix networks were studied in [30]. We use these theoretical results in our design.

8 Conclusion & Future Work

In this paper, we defined a model of privacy-preserving distributed execution of MapReduce jobs. We analyzed various attacks channels that break data confidentiality on a baseline system which employs both encryption and trusted computing primitives. Our new design realizes the defined level of security, with a significant step towards lower performance overhead while requiring a small TCB. Our experiments with M^{2}R showed that the system requires little effort to port legacy MapReduce applications, and is scalable.

Systems such as M^{2}R show evidence that specialized designs to hide data access patterns are practical alternatives to generic constructions such as ORAM. The question of how much special-purpose constructions benefit important practical systems, as compared to generic constructions, is an area of future work. A somewhat more immediate future work is to integrate our design to other distributed dataflow systems. Although having the similar structure of computation, those systems are based on different sets of computation primitives and different execution models, which presents both opportunities and challenges for reducing the performance overheads of our design. Another avenue for future work is to realize our model of privacy-preserving distributed computation in the emerging in-memory big-data platforms [64], where only very small overheads from security mechanisms can be tolerated.

9 Acknowledgements

The first author was funded by the National Research Foundation, Prime Minister's Office, Singapore, under its Competitive Research Programme (CRP Award No. NRF-CRP8-2011-08). A special thanks to Shruti Tople and Loi Luu for their help in preparing the manuscript. We thank the anonymous reviewers for their insightful comments that helped us improve the discussions in this work.

References

[1] Apache hadoop. `http://hadoop.apache.org`.

[2] Trusted computing group. `www.trustedcomputinggroup.org`.

[3] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. In *SIGMOD*, pages 563–574, 2004.

[4] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). In *SIGCOMM*, 2010.

[5] T. Alves and D. Felton. Trustzone: integrated hardware and software security. AMD white paper, 2004.

[6] A. Arasu, S. Blanas, K. Eguro, M. Joglekar, R. Kaushik, D. Kossmann, R. Ramamurthy, P. Upadhyaya, and R. Venkatesan. Secure database-as-a-service with cipherbase. In *SIGMOD*, pages 1033–1036, 2013.

[7] S. Bajaj and R. Sion. TrustedDB: a trusted hardware based database with privacy and data confidentiality. In *SIGMOD*, pages 205–216, 2011.

[8] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *OSDI*, 2014.

[9] M. Blanton, A. Steele, and M. Alisagari. Data-oblivious graph algorithms for secure computation and outsourcing. In *ASIACCS*, pages 207–218. ACM, 2013.

[10] J. Camenisch and A. Mityagin. Mix-network with stronger security. In *Privacy Enhancing Technologies*, 2006.

[11] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *IEEE Symposium on Foundations of Computer Science*, 2001.

[12] D. Champagne and R. B. Lee. Scalable architectural support for trusted software. In *HPCA*, 2010.

[13] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 1981.

[14] X. Chen, T. Garfinkel, E. Lewis, P. Subrahmanyam, C. Waldspurger, D. Boneh, J. Dwoskin, and D. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS*, pages 2–13, 2008.

[15] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *ACM CCS$'$06*, 2006.

[16] G. Danezis and C. Diaz. A survey of anonymous communication channels. Technical report, Technical Report MSR-TR-2008-35, Microsoft Research, 2008.

[17] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI*, 2014.

[18] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan. Side-channel vulnerability factor: a metric for measuring information leakage. In *ISCA*, 2012.

[19] R. Dingledine. Anonymity bibliography. http://freehaven.net/anonbib/.

[20] C. Gentry. Fully homomorphic encryption using ideal lattices. In *ACM Symposium on Theory of Computing*, May-June 2009.

[21] C. Gentry and S. Halevi. A working implementation of fully homomorphic encryption. In *EUROCRYPT*, 2010.

[22] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM*, 1996.

[23] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. In *ICALP*, 2011.

[24] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.

[25] S. Huang, J. Huang, Y. Liu, L. Yi, and J. Dai. Hibench: a representative and comprehensive hadoop benchmark suite. In *ICDE workshops*, 2010.

[26] M. Isard, M. Budiu, Y. Y. andAndrew Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Eurosys*, 2007.

[27] D. Jiang, G. Chen, B. C. Ooi, K.-L. Tan, and S. Wu. epic: an extensible and scalable system for processing big data. In *VLDB*, 2014.

[28] J. Katz and Y. Lindell. *Introduction to modern cryptography*. CRC Press, 2014.

[29] T. Kim, M. Peinado, and G. Mainar-Ruiz. Stealthmem: system-level protection against cache-based side channel attacks in the cloud. In *USENIX Security*, 2012.

[30] M. Klonowski and M. Kutyłowski. Provable anonymity for networks of mixes. In *Information Hiding*, pages 26–38. Springer, 2005.

[31] F. Li, B. C. Ooi, M. T. Ozsu, and S. Wu. Distributed data management using mapreduce. *ACM Computing Survey*, 46(6), 2014.

[32] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry. Minbox: a two-way sandbox for x86 native code. In *USENIX ATC*, 2014.

[33] C. Liu, M. Hicks, A. Harris, M. Tiwari, M. Maas, and E. Shi. Ghostrider: A hardware-software system for memory trace oblivious computation, 2015.

[34] C. Liu, M. Hicks, and E. Shi. Memory trace oblivious program execution. In *IEEE CSF*, pages 51–65. IEEE, 2013.

[35] C. Liu, Y. Huang, E. Shi, J. Katz, and M. Hicks. Automating efficient ram-model secure computation. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 623–638. IEEE, 2014.

[36] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. In *VLDB*, 2012.

[37] L. Luu, S. Shinde, P. Saxena, and B. Demsky. A model counter for constraints over unbounded strings. In *PLDI*, page 57, 2014.

[38] J. M. McCun, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for tcb minimization. In *EuroSys*, 2008.

[39] J. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. Trustvisor: Efficient tcb reduction and attestation. In *IEEE Symposium on Security and Privacy*, pages 143–158, 2010.

[40] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP*, 2013.

[41] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Information Security and Cryptology-ICISC 2005*, pages 156–168. Springer, 2006.

[42] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi. GraphSC: parallel secure computation made easy. In *IEEE Symposium on Security and Privacy*, 2015.

[43] O. Ohrimenko. *Data-oblivious algorithms for privacy-preserving access to cloud storage*. PhD thesis, Brown University, 2014.

[44] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: bringing order to the web. Technical report, Stanford InfoLab, 1999.

[45] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, May 1999.

[46] R. A. Popa, F. H. Li, and N. Zeldovich. An ideal-security protocol for order-preserving encoding. In *IEEE Symposium on Security and Privacy*, pages 463–477, 2013.

[47] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *SOSP*, 2011.

[48] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for mapreduce. In *NSDI*, 2010.

[49] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. Vc3: Trustworthy data analytics in the cloud. Technical report, Microsoft Research, 2014.

[50] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *SOSP*, pages 335–50, 2007.

[51] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing your faults from telling your secrets: Defenses against pigeonhole attacks. *CoRR*, abs/1506.04832, 2015.

[52] S. Shinde, S. Tople, D. Kathayat, and P. Saxena. Podarch: Protecting legacy applications with a purely hardware tcb. Technical report, National University of Singapore, 2015.

[53] V. Shmatikov and M.-H. Wang. Measuring relationship anonymity in mix networks. In *ACM workshop on Privacy in electronic society*, pages 59–62. ACM, 2006.

[54] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy*, May 2000.

[55] 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.

[56] U. Steinberg and B. Kauer. Nova: a microhypervisor-based secure virtualization architecture. In *Eurosys*, 2010.

[57] J. Szefer and R. B. Lee. Architectural support for hypervisor-secure virtualization. In *ASPLOS*, 2012.

[58] S. Tople, S. Shinde, Z. Chen, and P. Saxena. AUTOCRYPT: enabling homomorphic computation on servers to protect sensitive web content. In *ACM CCS*, pages 1297–1310, 2013.

[59] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. In *VLDB*, 2013.

[60] D. Wikström. A universally composable mix-net. In *Theory of Cryptography*. 2004.

[61] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better never than late: meeting deadlines in datacenter networks. In *SIGCOMM*, 2011.

[62] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed dataset: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[63] F. Zhang, J. Chen, H. Chen, and B. Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *SOSP*, pages 203–216, 2011.

[64] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang. In-memory big data management and processing: a survey. *TKDE*, 27(7):1920–1948, 2015.

[65] Z. Zhou, J. Han, Y.-H. Lin, A. Perrig, and V. Gligor. Kiss: ”key it simple and secure” corporate key management. In *TRUST*, 2013.

Appendix A Security Analysis

Proof (Lemma 1):

Consider the “ideal mixer” that takes as input a sequence $\langle x_1, \ldots, x_N \rangle$ where each $x_i \in [1,N]$, picks a permutation $p : [1,N] \to [1,N]$ randomly and then output the sequence $\langle x_{p(1)}, x_{p(2)}, \ldots, x_{p(N)} \rangle$. Klonowski et al. [30] investigated the effectiveness of the cascaded network of mixing, and showed that $O(\log \frac{N}{T})$ steps are suffice to bring the distribution of the mixed sequence statistically close to the output of the ideal mixer, where T is the number of items an instance can process in memory. Our proof relies on the above-mentioned result.

Let us assume that κ, the number of steps carried out by cascaded-mix, is sufficiently large such that the distribution of the mixed sequence is statistically close to the ideal mixer.

Consider an adversary $\mathscr{S}$ that executes the cascaded-mix. Let us construct an adversary $\mathscr{A}$ who simulates $\mathscr{S}$ but only has access to Ω. To fill in the tuple values not present in Ω, the simulation simply fills in random tuples. Note that the number of tuples can be derived from Ω.

Now, suppose that on input $x_1, \ldots, x_N$, the output of $\mathscr{A}$ and $\mathscr{S}$ can be distinguished by $\mathscr{D}$. We want to show that this contradicts the semantic security of the underlying encryption scheme, by constructing a distinguisher $\widetilde{\mathscr{D}}$ who can distinguish multiple ciphertexts from random with polynomial-time sampling (i.e. the distinguisher sends the challenger multiple messages, and receive more than one sample).

Let $z = \langle z_1, z_2, \ldots, z_N \rangle$ be the output of the mixer on input $x_1, \ldots, x_N$. The distinguisher $\widetilde{\mathscr{D}}$ asks the challenger for a sequence of ciphertexts of z. Let $c_{i,j}$'s be the ciphertexts returned by the challenger, where $c_{i,j}$ is the i-th ciphertexts of z_j. To emulate $\mathscr{S}$, likewise, $\widetilde{\mathscr{D}}$ needs to feed the simulation with the intermediate data generated by `mixT`. Let $y_{i,j}$ be the i-th intermediate ciphertext in round j the distinguisher $\widetilde{\mathscr{D}}$ generated for the emulation. The $y_{i,j}$'s are generated as follow:

1. $\widetilde{\mathscr{D}}$ simulates the cascaded-mix by randomly picking a permutation for every `mixT`. Let $p_j : [1,N] \to [1,N]$ be the overall permutation for round j. Let $\hat{p}_j$ be the permutation that moves the i-th ciphertext in the input, to its location after j rounds. That is, $\hat{p}_j(i) = p_j(\hat{p}_{j-1}(i))$, and $\hat{p}_0(i) = i$.

2. Set $y_{i,j} = c_{\hat{p}_j(i),j}$ for each i, j.

Let v be the output of $\mathscr{D}$'s simulation. Note that if $x_{i,j}$'s are random ciphertexts, then the distribution of v is the same as the output distribution of $\mathscr{A}$. On the other hand, if $x_{i,j}$'s are ciphertexts of z, then the input to the emulation is statistically close to the input of $\mathscr{S}$, and thus distribution of v is statistically close to the output distribution of S.

Since $\mathscr{D}$ can distinguish output of $\mathscr{S}$ from $\mathscr{A}$'s, $\widetilde{\mathscr{D}}$ can distinguish the ciphertexts of z from random. □

Measuring Real-World Accuracies and Biases in Modeling Password Guessability

Blase Ur, Sean M. Segreti, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor,
Saranga Komanduri, Darya Kurilova, Michelle L. Mazurek[†], William Melicher, Richard Shay
Carnegie Mellon University, [†]University of Maryland

Abstract

Parameterized password guessability—how many guesses a particular cracking algorithm with particular training data would take to guess a password—has become a common metric of password security. Unlike statistical metrics, it aims to model real-world attackers and to provide per-password strength estimates. We investigate how cracking approaches often used by researchers compare to real-world cracking by professionals, as well as how the choice of approach biases research conclusions.

We find that semi-automated cracking by professionals outperforms popular fully automated approaches, but can be approximated by combining multiple such approaches. These approaches are only effective, however, with careful configuration and tuning; in commonly used default configurations, they underestimate the real-world guessability of passwords. We find that analyses of large password sets are often robust to the algorithm used for guessing as long as it is configured effectively. However, cracking algorithms differ systematically in their effectiveness guessing passwords with certain common features (e.g., character substitutions). This has important implications for analyzing the security of specific password characteristics or of individual passwords (e.g., in a password meter or security audit). Our results highlight the danger of relying only on a single cracking algorithm as a measure of password strength and constitute the first scientific evidence that automated guessing can often approximate guessing by professionals.

1 Introduction

Despite decades of research into alternative authentication schemes, text passwords have comparative advantages—familiarity, ease of implementation, nothing for users to carry—that make a world without text passwords unlikely in the near future [5]. Two-factor authentication, single-sign-on systems, password managers, and biometrics promise to obviate remembering a distinct password for each online account, but passwords will not disappear entirely.

Text passwords have been compromised with alarming regularity through both online and offline attacks. While online attacks are mitigated through rate-limiting password-entry attempts, faulty rate limiting contributed to the iCloud photo leak [39]. In offline attacks, including recent ones on LinkedIn [7], eHarmony [62], Gawker [2], and Adobe [48], an attacker steals a database of (usually) hashed passwords and tries to recover passwords through offline guessing. Because password reuse is common [14], recovered passwords can often be used to access accounts on other systems.

A key aspect of improving password security is making passwords more computationally expensive to guess during offline attacks. Cracking tools like the GPU-based oclHashcat [57] and distributed cracking botnets [13, 17] enable attackers to make 10^{14} guesses in hours if passwords are hashed using fast hash functions like MD5 or NTLM. These advances are offset by the development of hash functions like bcrypt [52] and scrypt [47], which make attacks more difficult by requiring many iterations or consuming lots of memory.

Unfortunately, users often create predictable passwords [7, 29], which attackers can guess quickly even if the passwords are protected by a computationally expensive hash function. In some cases, predictable passwords are a rational coping strategy [54, 60]; in other cases, users are simply unsure whether a password is secure [66]. System administrators encourage strong passwords through password-composition policies and password-strength meters. The design and effectiveness of such mechanisms hinges on robust metrics to measure how difficult passwords are to guess.

In recent years, traditional entropy metrics have fallen out of favor because they do not reflect how easily a password can be cracked in practice [3, 31, 69]. It has instead become common to measure password strength by running or simulating a particular cracking algorithm, parameterized by a set of training data [4, 31, 69]. This approach has two main advantages. First, it calculates the guessability of each password individually, enabling data-driven strength estimates during password creation [10, 33]. Second, it estimates real-world security against existing, rather than idealized, adversarial techniques. A disadvantage of this approach is that the (simulated) cracking algorithm may not be configured or trained as effectively as by a real attacker, leading to inaccurate estimates of password strength.

This paper reports on the first study of how various cracking approaches used by researchers compare to real-world cracking by professionals, as well as how the choice of approach biases research conclusions. We contracted a computer security firm specializing in password recovery to crack a set of passwords chosen for their diversity in password-composition policies. We then computed the guessability of these passwords using four popular approaches. We tested many configurations of two well-known password-cracking toolkits: John the Ripper [49] and oclHashcat [57]. We also tested two approaches popular in academia: Weir et al.'s probabilistic context-free grammar (PCFG) [70] and Ma et al.'s Markov models [40].

Unsurprisingly, a professional attacker updating his strategy dynamically during cracking outperformed fully automated, "fire-and-forget" approaches (henceforth simply referred to as *automated*), yet often only once billions or trillions of guesses had been made. We found that relying on a single automated approach to calculate guessability underestimates a password's vulnerability to an experienced attacker, but using the earliest each password is guessed by any automated approach provides a realistic and conservative approximation.

We found that each approach was highly sensitive to its configuration. Using more sophisticated configurations than those traditionally used in academic research, our comparative analysis produced far more nuanced results than prior work. These prior studies found that Markov models substantially outperform the PCFG approach [18, 40], which in turn substantially outperforms tools like John the Ripper [16, 69, 72]. We found that while Markov was marginally more successful at first, it was eventually surpassed by PCFG for passwords created under typical requirements. Furthermore, the most effective configurations of John the Ripper and Hashcat were frequently comparable to, and sometimes even more effective than, the probabilistic approaches.

Both the differences across algorithms and the sensitivity to configuration choices are particularly notable because most researchers use only a single approach as a security metric [10, 12, 19, 42, 56, 65, 69]. In addition, many researchers use adversarial cracking tools in their default configuration [11, 14, 15, 20, 21, 28, 34, 71]. Such a decision is understandable since each algorithm is very resource- and time-intensive to configure and run. This raises the question of whether considering only a single approach biases research studies and security analyses. For instance, would substituting a different cracking algorithm change the conclusions of a study?

We investigate these concerns and find that for comparative analyses of large password sets (e.g., the effect of password-composition policies on guessability), choosing one cracking algorithm can reasonably be expected to yield similar results as choosing another.

However, more fine-grained analyses—e.g., examining what characteristics make a password easy to guess—prove very sensitive to the algorithm used. We find that per-password guessability results often vary by orders of magnitude, even when two approaches are similarly effective against large password sets as a whole. This has particular significance for efforts to help system administrators ban weak passwords or provide customized guidance during password creation [10, 33]. To facilitate the analysis of password guessability across many password-cracking approaches and to further systematize passwords research, we introduce a Password Guessability Service [9] for researchers.

In summary, this paper makes the following main contributions: We show that while running a single cracking algorithm or tool relatively out-of-the-box produces only a poor estimate of password guessability, using multiple well-configured algorithms or tools in parallel can approximate passwords' vulnerability to an expert, real-world attacker. Furthermore, while comparative analyses of large password sets may be able to rely on a single cracking approach, any analysis of the strength of individual passwords (e.g., a tool to reject weak ones) or the security impact of particular characteristics (e.g., the use of digits, multiple character classes, or character substitutions) must consider many approaches in parallel.

2 Related Work

In this section, we discuss commonly used metrics of password strength (Section 2.1) and describe popular categories of password-cracking attacks (Section 2.2).

2.1 Password Security Metrics

While estimated entropy was once a leading password strength metric [8], it does not reflect what portion of a set can be cracked easily [3, 31, 69]. Two main classes of metrics have emerged in its place: statistical metrics and parameterized metrics. Both classes focus on *guess-*

ability, the number of guesses needed by an adversary to guess a given password or a fraction of a set.

Statistical metrics are particularly valuable for examining password sets as a whole. For example, Bonneau introduced partial guessing metrics [3] for estimating the number of guesses required for an idealized attacker, who can perfectly order guesses, to guess a given fraction of a set. Since password distributions are heavy-tailed, very large samples are required to determine a set's guessability accurately.

Parameterized metrics instead investigate guessability under a cracking algorithm and training data [4, 31, 69]. These metrics thus model an adversary using existing tools, rather than an idealized attack, though the metric is only as good as the chosen algorithm and training data. Parameterized metrics can also be used to compare password sets without fully running the algorithm [40].

In contrast to statistical metrics, parameterized metrics have two important properties. First, they estimate the guessability of each password individually. Estimating guessability per-password is important for security audits (e.g., identifying weak passwords) and to provide feedback to a user about a password she has created. This latter promises to become more widespread as proactive feedback tools move from length-and-character-class heuristics [15] to data-driven feedback [10, 33]. Second, parameterized metrics aim to estimate security against real-world, rather than idealized, attacks. Researchers previously assumed automated techniques approximate real-world attackers [31, 69]; we are the first to test this assumption against attacks by professionals.

Parameterized metrics have been used to measure password strength in a number of previous studies [10, 14, 16, 20, 21, 31, 34, 40, 42, 53, 56, 65, 68, 69, 72]. While there are many different methods for cracking passwords, as we detail in Section 2.2, time and resource constraints lead many researchers to run only a single algorithm per study. However, it remains an open question whether this strategy accurately models real-world attackers, or whether choosing a different algorithm would change a study's results. We address this issue.

Throughout the paper, we refer to the *guess number* of a password, or how many guesses a particular parameterized algorithm took to arrive at that password. Because the algorithm must be run or simulated, there is necessarily a *guess cutoff*, or maximum guess after which remaining passwords are denoted "not guessed."

2.2 Types of Guessing Attacks

Researchers have long investigated how to guess passwords. A handful of studies [12, 16, 53] have compared the aggregate results of running different cracking approaches. Other studies have compared results of running different cracking approaches based on guess numbers [11, 18, 40]. We are the first to examine in detail the magnitude and causes of differences in these approaches' effectiveness at guessing specific passwords; we also compare approaches from academia and adversarial tools to a professional attacker. In this section, we highlight four major types of attacks.

Brute-force and mask attacks Brute-force attacks are conceptually the simplest. They are also inefficient and therefore used in practice only when targeting very short or randomly generated, system-assigned passwords.

Mask attacks are directed brute-force attacks in which password character-class structures, such as "seven lowercase letters followed by one digit" are exhausted in an attacker-defined order [58]. While this strategy may make many guesses without success, mask attacks can be effective for short passwords, as many users craft passwords matching popular structures [37, 63]. Real-world attackers also turn to mask attacks after more efficient methods exhaust their guesses. We evaluated mask attacks in our initial tests. Unsurprisingly, we found them significantly less efficient than other attacks we analyzed.

Probabilistic context-free grammar In 2009, Weir et al. proposed using a probabilistic context-free grammar (PCFG) with a large training set of passwords from major password breaches [67] to model passwords and generate guesses [70]. They use training data to create a context-free grammar in which non-terminals represent contiguous strings of a single character class. From the passwords observed in its training data, PCFG assigns probabilities to both the structure of a password (e.g., *monkey99* has the structure {*six letters*}{*two digits*}) and the component strings (e.g., "99" will be added to the list of two-digit strings it has seen). A number of research studies [11, 16, 19, 31, 40, 42, 56, 65, 69, 72] have used PCFG or a close variant to compute guessability.

Kelley et al. proposed other improvements to Weir et al.'s PCFG algorithm, like treating uppercase and lowercase letters separately and training with structures and component strings from separate sources [31]. Because they found these modifications improved guessing effectiveness, we incorporate their improvements in our tests. In addition, multiple groups of researchers have proposed using grammatical structures and semantic tokens as PCFG non-terminals [53, 68]. More recently, Komanduri proposed a series of PCFG improvements, including supporting hybrid structures and assigning probabilities to unseen terminals [32]. We incorporate his insights, which he found improves guessing efficiency.

Markov models Narayanan and Shmatikov first proposed using a Markov model of letters in natural language with finite automata representing password structures [45]. Castelluccia et al. used a similar algorithm for password meters [10]. John the Ripper and Hashcat offer simple Markov modes in their cracking toolkits as well.

Recently, Duermuth et al. [18] and Ma et al. [40] independently evaluated many variations of Markov models and types of smoothing in cracking passwords, using large sets of leaked passwords for training. Both groups compared their model with other probabilistic attacks, including Weir et al.'s original PCFG code, finding particular configurations of a Markov model to be more efficient at guessing passwords for some datasets. We use Ma et al.'s recommended model in our tests [40].

Mangled wordlist attacks Perhaps the most popular strategy in real-world password cracking is the dictionary attack. First proposed by Morris and Thompson in 1979 [43], modern-day dictionary attacks often combine *wordlists* with *mangling rules*, string transformations that modify wordlist entries to create additional guesses. Wordlists usually contain both natural language dictionaries and stolen password sets. Typical mangling rules perform transformations like appending digits and substituting characters [50, 59].

Many modern cracking tools, including John the Ripper [49], Hashcat [57], and PasswordsPro [30], support these attacks, which we term *mangled wordlist attacks*. The popularity of this category of attack is evident from these tools' wide use and success in password-cracking competitions [36,51]. Furthermore, a number of research papers have used John the Ripper, often with the default mangling rules [11,14,15,20,21,28,34,71] or additional mangling rules [16,19,72].

Expert password crackers, such as those offering forensic password-recovery services, frequently perform a variant of the mangled wordlist attack in which humans manually write, prioritize, and dynamically update rules [23]. We term these manual updates to mangling rules *freestyle rules*. As we discuss in Section 3, we evaluate guessability using off-the-shelf tools relying on publicly available wordlists and mangling rules. We also contract a password recovery industry leader to do the same using their proprietary wordlists and freestyle rules.

3 Methodology

We analyze four automated guessing algorithms and one manual cracking approach (together, our five *cracking approaches*). We first describe the password sets for which we calculated guessability, then explain the training data we used. Afterwards, we discuss our five cracking approaches. Finally, we discuss computational limitations of our analyses.

3.1 Datasets

We examine 13,345 passwords from four sets created under composition policies ranging from the typical to the currently less common to understand the success of password-guessing approaches against passwords of different characteristics. Since no major password leaks contain passwords created under strict composition policies, we leverage passwords that our group collected for prior studies of password-composition policies [31, 42, 56]. This choice of data also enables us to contract with a professional computer security firm to crack these unfamiliar passwords. Had we used any major password leak, their analysts would have already been familiar with most or all of the passwords contained in the leak, biasing results.

The passwords in these sets were collected using Amazon's Mechanical Turk crowdsourcing service. Two recent studies have demonstrated that passwords collected for research studies, while not perfect proxies for real data, are in many ways very representative of real passwords from high-value accounts [20, 42].

Despite these claims, we were also curious how real passwords would differ in our analyses from those collected on Mechanical Turk. Therefore, we repeated our analyses of Basic passwords (see below) with 15,000 plaintext passwords sampled from the RockYou gaming site leak [67] and another 15,000 sampled from a Yahoo! Voices leak [22]. As we detail in Appendix A.4, our Basic passwords and comparable passwords from these two real leaks yielded approximately the same results.

Next, we detail our datasets, summarized in Table 1. The **Basic** set comprises 3,062 passwords collected for a research study requiring a minimum length of 8 characters [31]. As we discuss in Section 4, the vast majority of 8-character passwords can be guessed using off-the-shelf, automated approaches. Hence, we give particular attention to longer and more complex passwords, which will likely represent best practices moving forward.

System administrators commonly require passwords to contain multiple character classes (lowercase letters, uppercase letters, digits, and symbols). The **Complex** set comprises passwords required to contain 8+ characters, include all 4 character classes, and not be in a cracking wordlist [46] after removing digits and symbols. They were also collected for research [42].

Recent increases in hashing speeds have made passwords of length 8 or less increasingly susceptible to offline guessing [24, 31]. We therefore examine 2,054 **LongBasic** passwords collected for research [31] that required a a minimum length of 16 characters. Finally, we

Table 1: Characteristics of passwords per set, including the percentage of characters that were lowercase (LC) or uppercase (UC) letters, digits, or symbols (Sym).

		Length	% of Characters			
Set	#	Mean (σ)	LC	UC	Digit	Sym
Basic	3,062	9.6 (2.2)	68	4	26	1
Complex	3,000	10.7 (3.2)	51	14	25	11
LongBasic	2,054	18.1 (3.1)	73	4	20	2
LongComplex	990	13.8 (2.6)	57	12	22	8

examine 990 **LongComplex** passwords, also collected for research [56], that needed to contain 12+ characters, including characters from 3 or more character classes.

3.2 Training Data

To compare cracking approaches as directly as possible, we used the same training data for each. That said, each algorithm uses training data differently, making perfectly equivalent comparisons impossible.

Our training data comprised leaked passwords and dictionaries. The passwords were from breaches of MySpace, RockYou, and Yahoo! (excluding the aforementioned 30,000 passwords analyzed in Appendix A.4). Using leaked passwords raises ethical concerns. We believe our use of such sets in this research is justifiable because the password sets are already available publicly and we exclude personally identifiable information, such as usernames. Furthermore, malicious agents use these sets in attacks [23]; failure to consider them in our analyses may give attackers an advantage over those who work in defensive security.

Prior research has found including natural-language dictionaries to work better than using just passwords [31, 69]. We used the dictionaries previously found most effective: all single words in the Google Web corpus [26], the UNIX dictionary [1], and a 250,000-word inflection dictionary [55]. The combined set of passwords and dictionaries contained 19.4 million unique entries. For cracking approaches that take only a wordlist, without frequency information, we ordered the wordlist by descending frequency and removed duplicates. We included frequency information for the other approaches.

3.3 Simulating Password Cracking

To investigate the degree to which research results can be biased by the choice of cracking algorithm, as well as how automated approaches compare to real attacks, we investigated two cracking tools and two probabilistic algorithms. We selected approaches based on their popularity in the academic literature or the password-cracking community, as well as their conceptual distinctness. We also contracted a computer security firm specializing in password cracking for the real-world attack.

Most cracking approaches do not natively provide guess numbers, and instrumenting them to calculate guessability was typically far from trivial. Because this instrumentation enabled the comparisons in this paper and can similarly support future research, we include many details in this section about this instrumentation. Furthermore, in Section 5, we introduce a Password Guessability Service so that other researchers can leverage our instrumentation and computational resources.

For each approach, we analyze as many guesses as computationally feasible, making 100 trillion (10^{14}) guesses for some approaches and ten billion (10^{10}) guesses for the most resource-intensive approach. With the exception of Hashcat, as explained below, we filter out guesses that do not comply with a password set's composition policy. For example, a LongComplex password's guess number excludes guesses with under 12 characters or fewer than 3 character classes.

We define **Min**$_{\text{auto}}$ as the minimum guess number (and therefore the most conservative security result) for a given password across our automated cracking approaches. This number approximates the best researchers can expect with well-configured automation.

In the following subsections, we detail the configuration (and terminology) of the five approaches we tested. We ran CPU-based approaches (JTR, PCFG, Markov) on a 64-core server. Each processor on this server was an AMD Opteron 6274 running at 1.4Ghz. The machine had 256 GB of RAM and 15 TB of disk. Its market value is over $10,000, yet we still faced steep resource limitations generating Markov guesses. We ran Hashcat (more precisely, oclHashcat) on a machine with six AMD R9 270 GPUs, 2 GB of RAM, and a dual-core processor.

Probabilistic context-free grammar Weir et al.'s probabilistic context-free grammar (termed **PCFG**) [70] has been widely discussed in recent years. We use Komanduri's implementation of PCFG [32], which improves upon the guessing efficiency of Weir et al.'s work [70] by assigning letter strings probabilities based on their frequency in the training data and assigning unseen strings a non-zero probability. This implementation is a newer version of Kelley et al.'s implementation of PCFG as a lookup table for quickly computing guess numbers, rather than enumerating guesses [31].

Based on our initial testing, discussed further in Section 4.1, we prepend our training data, ordered by frequency, before PCFG's first guess to improve performance. As a result, we do not use Komanduri's hybrid structures [32], which serve a similar purpose. We weight passwords 10× as heavily as dictionary entries.

We were able to simulate 10^{12} guesses for Complex passwords and 10^{14} guesses for the other three sets.

Markov model Second, we evaluated the **Markov**-model password guesser presented by Ma et al. [40], which implemented a number of variants differing by order and approaches to smoothing. We use the order-5 Markov-chain model, which they found most effective for English-language test sets. We tried using both our combined training data (dictionaries and paswords) using the same weighting as with PCFG, as well as only the passwords from our training data. The combined training data and passwords-only training data performed nearly identically. We report only on the combined training data, which was slightly more effective for Basic passwords and is most consistent with the other approaches.

We used Ma et al.'s code [40], which they shared with us, to enumerate a list of guesses in descending probability. We used a separate program to remove guesses that did not conform to the given password-composition policy. Because this approach is extremely resource-intensive, both conceptually (traversing a very large tree) and in its current implementation, we were not able to analyze as many guesses as for other approaches. As with PCFG, we found prepending the training data improved performance, albeit only marginally for Markov. Therefore, we used this tweak. We simulated over 10^{10} guesses for Basic passwords, similar to Ma et al. [40].

John the Ripper We also tested variants of a mangled wordlist attack implemented in two popular software tools. The first tool, John the Ripper (termed **JTR**), has been used in a number of prior studies as a security metric, as described in Section 2. In most cases, these prior studies used JTR with its stock mangling rules. However, pairing the stock rules with our 19.4-million-word wordlist produced only 10^8 guesses for Basic passwords. To generate more guesses, we augment the stock rules with 5,146 rules released for DEF CON's "Crack Me If You Can" (CMIYC) password-cracking contest in 2010 [35]. Specifically, we use Trustwave SpiderLabs' reordering of these rules for guessing efficiency [64]. Our JTR tests therefore use the stock mangling rules followed by the Spiderlabs rules. For completeness, Appendix A.2 presents these rules separately.

Instrumenting JTR to calculate precise guess numbers was an involved process. We used `john-1.7.9-jumbo` with the `--stdout` flag to output guesses to standard out. We piped these guesses into a program we wrote to perform a regular expression check filtering out guesses that do not conform to the given password policy. This program then does a fast hash table lookup with GNU `gperf` [27] to quickly evaluate whether a guess matches a password in our dataset. Using this method, we achieved a throughput speed of 3 million guesses per second and made more than 10^{13} guesses for Basic passwords.

Hashcat While Hashcat is conceptually similar to JTR, we chose to also include it in our tests for two reasons. First, we discovered in our testing that JTR and Hashcat iterate through guesses in a very different order, leading to significant differences in the efficacy of guessing specific passwords. JTR iterates through the entire wordlist using one mangling rule before proceeding to the subsequent mangling rule. Hashcat, in contrast, iterates over all mangling rules for the first wordlist entry before continuing to the subsequent wordlist entry.

Second, the GPU-based oclHashcat, which is often used in practice [23, 24, 36, 51], does not permit users to filter guesses that do not meet password-composition requirements except for computationally expensive hash functions. We accept this limitation both because it represents the actual behavior of a popular closed-source tool and because, for fast hashes like MD5 or NTLM, guessing without filtering cracks passwords faster in practice than applying filtering.

Unlike JTR, Hashcat does not have a default set of mangling rules, so we evaluated several. We generally report on only the most effective set, but detail our tests of four different rule sets in Appendix A.3. This most effective rule set, which we term **Hashcat** throughout the paper, resulted from our collaboration with a Hashcat user and password researcher from MWR InfoSecurity [25, 44], who shared his mangling rules for the purpose of this analysis. We believe such a configuration represents a typical expert configuration of Hashcat.

We used `oclHashcat-1.21`. While, like JTR, Hashcat provides a debugging feature that streams guesses to standard output, we found it extremely slow in practice relative to Hashcat's very efficient GPU implementation. In support of this study, Hashcat's developers generously added a feature to oclHashcat to count how many guesses it took to arrive at each password it cracked. This feature is activated using the flag `--outfile-format=11` in `oclHashcat-1.20` and above. We therefore hashed the passwords in our datasets using the NTLM hash function, which was the fastest for Hashcat to guess in our benchmarks. We then used Hashcat to actually crack these passwords while counting guesses, with throughput of roughly 10 billion guesses per second on our system. We made more than 10^{13} guesses for Basic passwords, along with nearly 10^{15} guesses for some alternate configurations reported in Appendix A.3.

Professional cracker An open question in measuring password guessability using off-the-shelf, automated tools is how these attacks compare to an experienced, real-world attacker. Such attackers manually customize and dynamically update their attacks based on a target set's characteristics and initial successful cracks.

To this end, we contracted an industry leader in professional password recovery services, KoreLogic (termed **Pros**), to attack the password sets we study. We believe KoreLogic is representative of expert password crackers because they have organized the DEF CON "Crack Me If You Can" password-cracking contest since 2010 [36] and perform password-recovery services for many Fortune-500 companies [38]. For this study, they instrumented their distributed cracking infrastructure to count guesses.

Like most experienced crackers, the KoreLogic analysts used tools including JTR and Hashcat with proprietary wordlists, mangling rules, mask lists, and Markov models optimized over 10 years of password auditing. Similarly, they dynamically update their mangling rules (termed *freestyle rules*) as additional passwords are cracked. To unpack which aspects of a professional attack (e.g., proprietary wordlists and mangling rules, freestyle rules, etc.) give experienced crackers an advantage, we first had KoreLogic attack a set of 4,239 Complex passwords (distinct from those reported in our other tests) in artificially limited configurations.

We then had the professionals attack the Complex, LongBasic, and LongComplex passwords with no artificial limitations. An experienced password analyst wrote freestyle rules for each set before cracking began, and again after 10^{13} guesses based on the passwords guessed to that point. They made more than 10^{14} guesses per set.

LongBasic and LongComplex approaches are rare in corporate environments and thus relatively unfamiliar to real-world attackers. To mitigate this unfamiliarity, we randomly split each set in two and designated half for training and half for testing. We provided analysts with the training half (in plaintext) to familiarize them with common patterns in these sets. Because we found that automated approaches can already crack most Basic passwords, rendering them insecure, we chose not to have the professionals attack Basic passwords.

3.4 Computational Limitations

As expected, the computational cost of generating guesses in each approach proved a crucial limiting factor in our tests. In three days, oclHashcat, the fastest of our approaches, produced 10^{15} guesses using a single AMD R9 290X GPU (roughly a $500 value). In contrast, the Markov approach (our slowest) required three days on a roughly $10,000 server (64 AMD Opteron 6274 CPU cores and 256 GB of RAM) to generate 10^{10} guesses without computing a single hash. In three days on the same machine as Markov, PCFG simulated 10^{13} guesses.

The inefficiency of Markov stems partially from our use of a research implementation. Even the most efficient implementation, however, would still face substantial conceptual barriers. Whereas Hashcat and JTR incur the same performance cost generating the quadrillionth guess as the first guess, Markov must maintain a tree of substring probabilities. As more guesses are desired, the tree must grow, increasing the cost of both storing and traversing it. While Markov produced a high rate of successful guesses per guess made (see Section 4.2), the cost of generating guesses makes it a poor choice for computing guessability beyond billions of guesses.

Further, our automated approaches differ significantly in how well they handle complex password-composition policies. For PCFG, non-terminal structures can be pruned before guessing starts, so only compliant passwords are ever generated. As a result, it takes about equal time for PCFG to generate Basic passwords as LongComplex passwords. In contrast, Markov must first generate all passwords in a probability range and then filter out non-compliant passwords, adding additional overhead per guess. JTR has a similar generate-then-filter mechanism, while Hashcat (as discussed above) does not allow this post-hoc filtering at all for fast hashes. This means that Markov and JTR take much longer to generate valid LongComplex guesses than Basic guesses, and Hashcat wastes guesses against the LongComplex set.

As a result of these factors, the largest guess is necessarily unequal among approaches we test, and even among test sets within each approach. To account for this, we only compare approaches directly at equivalent guess numbers. In addition, we argue that these computational limitations are important in practice, so our findings can help researchers understand these approaches and choose among them appropriately.

4 Results

We first show, in Section 4.1, that for each automated guessing approach we evaluated, different seemingly reasonable configurations produce very different cracking results, and that out-of-the-box configurations commonly used by researchers substantially underestimate password vulnerability.

Next, in Section 4.2, we examine the relative performance of the four automated approaches. We find they are similarly effective against Basic passwords. They have far less success against the other password sets, and their relative effectiveness also diverges.

For the three non-Basic sets, we also compare the automated approaches to the professional attack. Pros outperform the automated approaches, but only after a large number of guesses. As Pros crack more passwords, their manual adjustments prove quite effective; automated approaches lack this feedback mechanism. We also find that, at least through 10^{14} guesses, automated approaches can conservatively approximate human password-cracking experts, but only if a password is counted as guessed when *any* of the four automated approaches guesses it. A single approach is not enough.

In Section 4.3, we explore the degree to which different cracking approaches overlap in which particular passwords they guess. While multiple approaches successfully guess most Basic passwords, many passwords in the other classes are guessed only by a single approach. We also find that different cracking approaches provide systematically different results based on characteristics like the number of character classes in a password.

In Section 4.4, we revisit how the choice of guessing approach impacts research questions at a high level (e.g., how composition policies impact security) and lower level (e.g., if a particular password is hard to guess). While we find analyses on large, heterogeneous sets of passwords to be fairly robust, security estimates for a given password are very sensitive to the approach used.

4.1 The Importance of Configuration

We found that using any guessing approach naively performed far more poorly, sometimes by more than an order of magnitude, than more expert configurations.

Stock vs advanced configurations We experimented with several configurations each for Hashcat and JTR, including the default configurations they ship with, and observed stark differences in performance. We detail a few here; others are described in Appendices A.2 and A.3.

For example, Hashcat configured with the (default) Best64 mangling rules guessed only about 2% of the Complex passwords before running out of guesses. Using the mangling rules described in Section 3, it made far more guesses, eventually cracking 30% (Figure 1).

Similarly, JTR guessed less than 3% of Complex passwords before exhausting its stock rules. The larger set of rules described in Section 3 enabled it to guess 29% (see Appendix A.2 for details). We found similar configuration effects for LongComplex passwords, and analogous but milder effects for the Basic and LongBasic sets.

We also compared the PCFG implementation we use throughout the paper [32] with our approximation of the originally published algorithm [70], which differs in how probabilities are assigned (see Section 3). As we detail in Appendix A.1, the newer PCFG consistently outperforms the original algorithm; the details of the same conceptual approach greatly impact guessability analyses.

Choices of training data The performance of PCFG and Markov depends heavily on the quality of training data. Our group previously found that training with closely related passwords improves performance [31]. For our non-basic password sets, however, closely matched data is not available in publicly leaked sets.

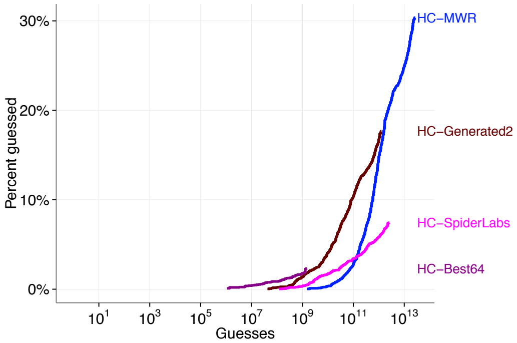


Figure 1: Results of Hashcat configured using the same wordlist, but different sets of mangling rules (described in Appendix A.3), to guess Complex passwords.

In tests reported in Appendix A.1, we thus incorporated closely matched data via cross-validation, in which we iteratively split the test set into training and testing portions. Using cross-validation improved guessing efficiency for three of the four password sets, most dramatically for LongBasic. This result demonstrates that an algorithm trained with generic training data will miss passwords that are vulnerable to an attacker who has training data that closely matches a target set. To minimize differences across approaches, however, PCFG results in the body of the paper use generic training data only.

Actionable takeaways Together, these results suggest that a researcher must carefully manage guessing configuration before calculating password guessability. In particular, tools like JTR and Hashcat will "out of the box" systematically underestimate password guessability. Unfortunately, many existing research studies rely on unoptimized configurations [11, 14, 15, 20, 21, 28, 34, 71].

While we report on the configurations we found most effective in extensive testing, we argue that the research community should establish configuration best practices, which may depend on the password sets targeted.

4.2 Comparison of Guessing Approaches

We first show that automated approaches differ in effectiveness based on the nature of the password sets being cracked and the number of guesses at which they are compared. We then compare these automated approaches to cracking by an expert attacker making dynamic updates, finding that the expert lags in initial guessing efficiency, yet becomes stronger over time. We find the minimum guess number across automated approaches can serve as a conservative proxy for guessability by an expert attacker.

4.2.1 Guessing by Automated Approaches

On some password sets and for specific numbers of guesses, the performance of all four approaches was similar (e.g., at 10^{12} guesses all but Markov had guessed 60-70% of Basic passwords). In contrast, on other sets, their performance was inconsistent at many points that would be relevant for real-world cracking (e.g., PCFG cracked 20% of Complex passwords by 10^{10} guesses, while Hashcat and JTR had cracked under 3%).

As shown in Figure 2, all four automated approaches were quite successful at guessing Basic passwords, the most widely used of the four classes. Whereas past work has found that, for password sets resembling our Basic passwords, PCFG often guesses more passwords than JTR [16] or that Markov performs significantly better than PCFG [40], good configurations of JTR, Markov, and PCFG performed somewhat similarly in our tests. Hashcat was less efficient at generating successful guesses in the millions and billions of guesses, yet it surpassed JTR by 10^{12} guesses and continued to generate successful guesses beyond 10^{13} guesses.

The four automated approaches had far less success guessing the other password sets. Figure 3 shows the guessability of the Complex passwords under each approach. Within the first ten million guesses, very few passwords were cracked by any approach. From that point until its guess cutoff, PCFG performed best, at points having guessed nearly ten times as many passwords as JTR. Although its initial guesses were often successful, the conceptual and implementation-specific performance issues we detailed in Section 3.4 prevented Markov from making over 100 million valid Complex guesses, orders of magnitude less than the other approaches we examined. A real attack using this algorithm would be similarly constrained.

Both Hashcat and JTR performed poorly compared to PCFG in early Complex guessing. By 10^9 guesses, each had each guessed under 3% of Complex passwords, compared to 20% for PCFG. Both Hashcat and JTR improve rapidly after 10^{10} guesses, however, eventually guessing around 30% of Complex passwords.

JTR required almost 10^{12} guesses and Hashcat required over 10^{13} guesses to crack 30% of Complex passwords. As we discuss in Section 4.3, there was less overlap in which passwords were guessed by multiple automated approaches for Complex passwords than for Basic passwords. As a result, the Min_{auto} curve in Figure 3, representing the smallest guess number per password across the automated approaches, shows that just under 10^{11} guesses are necessary for 30% of Complex passwords to have been guessed by at least one automated approach. Over 40% of Complex passwords were guessed by at least one automated approach in 10^{13} guesses.

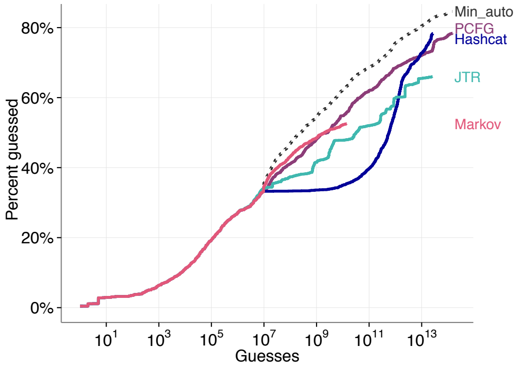


Figure 2: Automated approaches' success guessing Basic passwords. Min_{auto} represents the smallest guess number for a password by any automated approach.

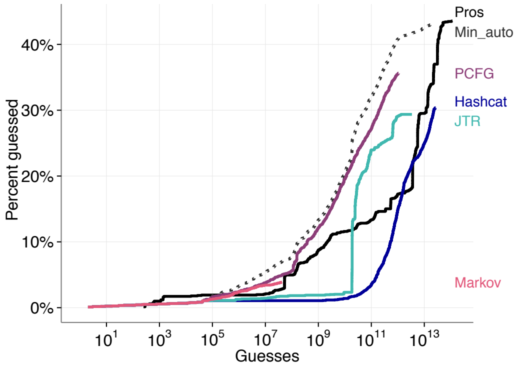


Figure 3: Success guessing Complex passwords. Pros are experts updating their guessing strategy dynamically.

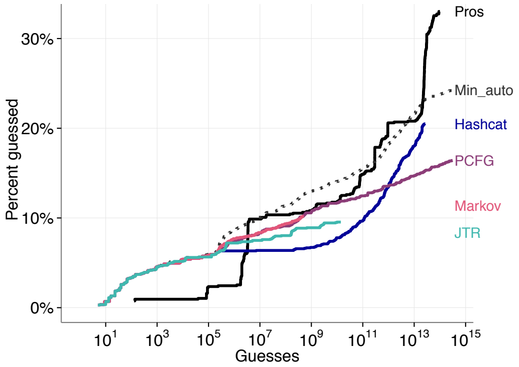


Figure 4: Success guessing LongBasic passwords.

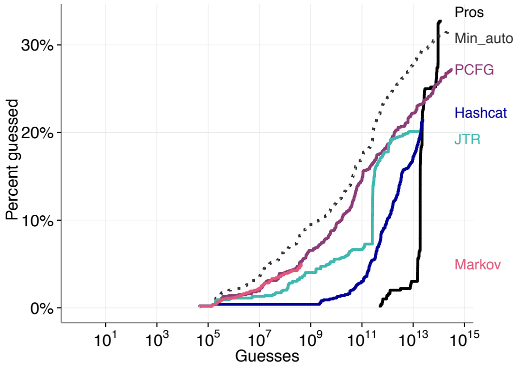


Figure 5: Success guessing LongComplex passwords.

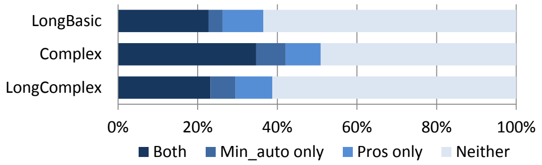


Figure 6: The proportion of passwords guessed by Min_{auto}, Pros, both, or neither within 10^{14} guesses.

LongBasic passwords were also challenging for all approaches to guess, though relative differences across approaches are not as stark as for Complex passwords. Markov was marginally more successful than other approaches at its cutoff just before 10^9 guesses. JTR and PCFG both continued to generate successful guesses through when JTR exhausted its guesses after guessing 10% of the passwords. Hashcat lagged slightly behind JTR at 10^9 guesses (7% cracked vs ∼9%), but was able to make more guesses than either, eventually guessing over 20% of the passwords, compared to 16% for PCFG and 10% for JTR at those approaches' guess cutoffs.

As with LongBasic passwords, all approaches had difficulty guessing LongComplex passwords. As shown in Figure 5, nearly 70% of LongComplex passwords were not guessed by any of the approaches we examined even after trillions of guesses. The relative performance of the four automated guessing approaches for LongComplex passwords again differed noticeably. Markov and PCFG again outperformed other approaches early. Markov guessed 5% of the passwords after 10^8 guesses, yet reached its guess cutoff soon thereafter. At 10^9 guesses PCFG and JTR had both also guessed at least 5% of the passwords, compared to almost no passwords guessed by Hashcat. PCFG's and JTR's performance diverged and then converged at higher guess numbers. Hashcat caught up at around 10^{13} guesses, cracking 20% of LongComplex passwords.

4.2.2 Guessing by Pros

As we expected, Pros guessed more passwords overall than any of the automated approaches. As we discussed in Section 3, we chose not to have Pros attack Basic passwords because those passwords could be guessed with automated approaches alone. As shown in Figures 3–5, within 10^{14} guesses Pros cracked 44% of Complex passwords, 33% of LongBasic passwords, and 33% of LongComplex passwords, improving on the guessing of the best automated approach.

Three aspects of guessing by Pros were particularly notable. First, even though Pros manually examined half of each password set and adjusted their mangling rules and wordlists before making the first guess against each set, automated approaches were often more successful at early guessing. For example, Markov surpassed Pros at guessing Complex passwords in the first 10^2 guesses and again from around 10^6 till Markov's guess cutoff at 5×10^7. Similarly, all four automated approaches guessed LongComplex passwords more successfully than Pros from the start of guessing until past 10^{13} guesses. All approaches guessed LongBasic passwords better than Pros for the first 10^6 guesses.

Second, while Pros lagged in early guessing, the freestyle rules an experienced analyst wrote at 10^{13} guesses proved rather effective and caused a large spike in successful guesses for all three password sets. Hashcat, the only automated approach that surpassed 10^{13} guesses for all sets, remained effective past 10^{13} guesses, yet did not experience nearly the same spike.

Third, while Pros were more successful across password sets once a sufficiently high number of guesses had been reached, the automated approaches we tested had guessing success that was, to a very rough approximation, surprisingly similar to Pros. As we discussed in Section 4.1 and discuss further in the appendix, this success required substantial configuration beyond each approach's performance out of the box.

We found that our Min_{auto} metric (the minimum guess number for each password across Hashcat, JTR, Markov, and PCFG) served as a conservative approximation of the success of Pros, at least through our automated guess cutoffs around 10^{13} guesses. As seen in Figures 3–6, Pros never substantially exceeded Min_{auto}, yet often performed worse than Min_{auto}.

Professional cracking with limitations To unpack why professional crackers have an advantage over novice attackers, we also had KoreLogic attack a different set of Complex passwords in artificially limited configurations. These limitations covered the wordlists they used, the mangling rules they used, and whether they were permitted to write freestyle rules. To avoid biasing subsequent tests, we provided them a comparable set of 4,239 Complex passwords [31] distinct from those examined in the rest of the paper. We call this alternate set $\textbf{Complex}_{pilot}$.

As shown in Table 2, we limited Pros in Trial 1 to use the same wordlist we used elsewhere in this paper and did not allow freestyle rules. In Trial 2, we did not limit the wordlist, but did limit mangling rules to those used in the 2010 Crack Me If You Can contest [35]. In Trial 3 and Trial 4, we did not limit the starting wordlist or mangling rules. In Trial 4, however, KoreLogic analysts dynamically adjusted their attacks through freestyle rules and wordlist tweaks after 10^{14} guesses.

We found that KoreLogic's set of proprietary mangling rules had a far greater impact on guessing efficiency than their proprietary wordlist (Figure 7). Furthermore, as evidenced by the difference between Trial 3

Table 2: The four trials of Pros guessing Complex$_{pilot}$. We artificially limited the first three trials to uncover why Pros have an advantage over more novice attackers.

Trial	Wordlist	Rules	Freestyle Rules
1	CMU wordlist	Anything	None
2	Anything	2010 CMIYC rules	None
3	Anything	Anything	None
4	Anything	Anything	Unlimited

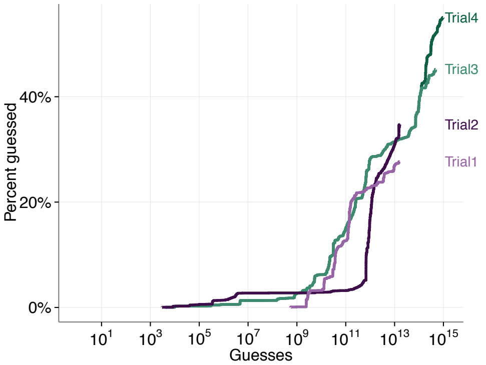


Figure 7: Complex$_{pilot}$ guessability by trial.

and Trial 4, freestyle rules also had a major impact at the point the analyst wrote them.

Actionable takeaways One conceptual advantage of parameterized metrics is that they model an attack using existing cracking approaches. However, it has long been unclear whether automated cracking approaches used by researchers effectively model the dynamically updated techniques used by expert real-world attackers. Our results demonstrate that only by considering multiple automated approaches in concert can researchers approximate professional password cracking.

One of our primary observations, both from comparing Pros to the automated approaches and from our trials artificially limiting Pros (Section 4.2.2), is that dynamically updated freestyle rules can be highly effective. This result raises the question of to what extent automated approaches can model dynamic updates. Although the adversarial cracking community has discussed techniques for automatically generating mangling rules from previous cracks [41], researchers have yet to leverage such techniques, highlighting an area ripe for future work.

Contrary to prior research (e.g., [16, 40]), we found that Hashcat, JTR, Markov, and PCFG all performed relatively effectively when configured and trained according to currently accepted best practices in the cracking and research communities. That said, our tests also highlighted a limitation of the guessability metric in not considering the performance cost of generating a guess. Despite its real-world popularity, Hashcat performed comparatively poorly until making trillions of guesses, yet generated guesses very quickly.

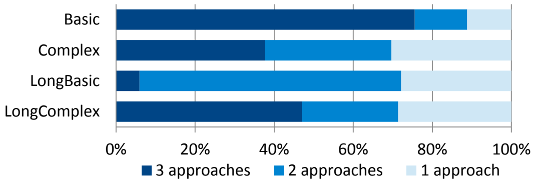


Figure 8: Number of automated approaches, excluding Markov, that cracked a particular password. We ignore passwords not guessed by any approach and use the same guess cutoff for all guessing approaches within a set.

If hashing a guess is the dominant time factor, as is the case for intentionally slow hash functions like bcrypt, PBKDF2, and scrypt, probabilistic approaches like Markov and PCFG are advantageous for an attacker. For fast hash functions like MD5 or NTLM, Hashcat's speed at generating and hashing guesses results in more passwords being guessed in the same wall-clock time. As discussed in Section 3.4, Markov proved comparatively very resource-intensive to run to a large guess number, especially for password sets with complex requirements. These practical considerations must play a role in how researchers select the best approaches for their needs.

4.3 Differences Across Approaches

Next, we focus on differences between approaches. We first examine if multiple approaches guess the same passwords. We then examine the guessability of passwords with particular characteristics, such as those containing multiple character classes or character substitutions. To examine differences across how approaches model passwords, for analyses in this section we do not prepend the training data to the guesses generated by the approach.

4.3.1 Overlap in Successful Guesses

While one would expect any two cracking approaches to guess slightly different subsets of passwords, we found larger-than-expected differences for three of the four password sets. Figure 8 shows the proportion of passwords in each class guessed by all four approaches, or only some subset of them. We exclude passwords guessed by none of the automated approaches. Within a password set, we examine all approaches only up to the minimum guess cutoff among Hashcat, JTR, and PCFG; we exclude Markov due to its low guess cutoffs.

The three approaches guessed many of the same Basic passwords: Three-fourths of Basic passwords guessed by any approach were guessed by all of them. Only 11% of Basic passwords were guessed only by a single ap-

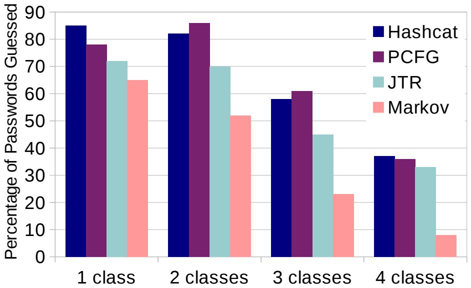


Figure 9: Percentage of Basic passwords each approach guessed, by character-class count.

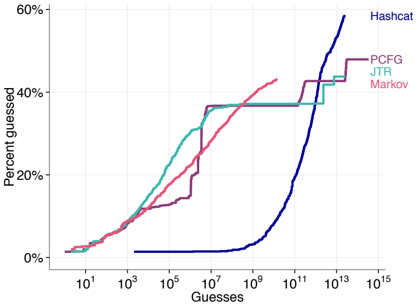


Figure 10: Approaches' effectiveness guessing passwords composed entirely of lowercase letters across sets.

proach. In contrast, only 6% of LongBasic passwords were guessed by all approaches, while 28% of Complex, LongBasic, and LongComplex passwords were guessed only by a single approach.

4.3.2 Guessing Success by Password Characteristics

While it is unsurprising that different approaches do better at guessing distinct types of passwords, we found differences that were large and difficult to predict.

Character classes and length We first considered how efficiently automated approaches guessed passwords relative to their length and character-class count. These two characteristics are of particular interest because they are frequently used in password-composition policies.

As shown in Figure 9, the impact of adding character classes is not as straightforward as one might expect. While the general trend is for passwords with more character classes to be stronger, the details vary. Markov experiences a large drop in effectiveness with each increase in character classes (63% to 52% to 23% to 8%). JTR, by contrast, finds only a minor difference between one and two classes (72% to 70%). PCFG actually increases in effectiveness between one and two classes (78% to 86%). Since changes in security and usability as a result of different policies are often incremental (e.g., [8]), the magnitude of these disagreements can easily affect research conclusions about the relative strength of passwords.

In contrast, we did not find surprising idiosyncrasies based on the length of the password. For all approaches, cracking efficiency decreased as length increased.

Character-level password characteristics As the research community seeks to understand the characteristics of good passwords, a researcher might investigate how easy it is to guess all-digit passwords, which are common [6], or examine the effect of character substitutions (e.g., *$Hplovecraft!$* → *$Hpl0v3cr@ft!$*) on guessability. Despite their sometimes similar effectiveness overall, approaches often diverged when guessing passwords that had these characteristics. As a result, researchers using different approaches could draw different conclusions about the guessability of these properties.

The guessability of the 1,490 passwords (across sets) composed entirely of lowercase letters varied starkly by guessing approach. This variation is particularly notable because such passwords made up 29% of Basic and LongBasic passwords, and were impermissible under the other two composition policies. As shown in Figure 10, Hashcat guessed few such passwords until well into the billions of guesses, whereas Markov successfully guessed passwords composed entirely of lowercase letters throughout its attack. In contrast, PCFG had a large spike in successful guesses between 1 million and 10 million guesses, but then plateaued. JTR had early success, but similarly plateaued from 10 million guesses until into the trillions of guesses.

Similarly, approaches differed in their efficiency guessing passwords containing character substitutions, which we identified using crowdsourcing on Amazon's Mechanical Turk. Passwords identified by crowdworkers as containing character substitutions included *4Everblessed*, *B1cycle_Race*, and *Ca$hmoneybr0*. PCFG performed poorly relative to JTR and Markov at guessing passwords with character substitutions. A researcher using only PCFG could mistakenly believe these passwords are much stronger than they actually are. We found similar differences with many other common characteristics, potentially skewing research conclusions.

Actionable takeaways Given the many passwords guessed by only a single cracking approach and the systematic differences in when passwords with certain characteristics are guessed, we argue that researchers must consider major cracking approaches in parallel.

Our results also show how comparative analyses uncover relative weaknesses of each approach. Upon close examination, many of these behaviors make sense. For example, PCFG abstracts passwords into structures of non-terminal characters based on character class, ignoring contextual information across these boundaries. As a result, *P@ssw0rd* would be split into "P," "@," "ssw," "0," and "rd," explaining PCFG's poor performance guessing passwords with character substitutions.

4.4 Robustness of Analyses to Approach

In this section, we examine whether differences among automated cracking approaches are likely to affect conclusions to two main types of research questions.

We first consider analyses of password sets, such as passwords created under particular password-composition policies. We find such analyses to be somewhat, but not completely, robust to the approach used.

In contrast, per-password analyses are very sensitive to the guessing approach. Currently, such analyses are mainly used in security audits [61] to detect weak passwords. In the future, however, per-password strength metrics may be used to provide detailed feedback to users during password creation, mirroring the recent trend of data-driven password meters [10, 33]. The ability to calculate a guess number per-password is a major advantage of parameterized metrics over statistical metrics, yet this advantage is lost if guess numbers change dramatically when a different approach is used. Unfortunately, we sometimes found huge differences across approaches.

4.4.1 Per Password Set

As an example of an analysis of large password sets, we consider the relative guessability of passwords created under different composition policies, as has been studied by Shay et al. [56] and Kelley et al. [31].

Figure 11 shows the relative guessability of the three password sets examined by the Pros. LongBasic passwords were most vulnerable, and LongComplex passwords least vulnerable, to early guessing (under 10^9 guesses). Between roughly 10^9 and 10^{12} guesses, LongBasic and Complex passwords followed similar curves, though Complex passwords were cracked with higher success past 10^{12} guesses. Very few LongComplex passwords were guessed before 10^{13} guesses, yet Pros quickly guessed about one-third of the LongComplex set between 10^{13} and 10^{14} guesses.

Performing the same analysis using Min_{auto} guess numbers instead (Figure 12) would lead to similar conclusions. LongBasic passwords were again more vulnerable than Complex or LongComplex under 10^8 guesses. After 10^{12} guesses, Complex passwords were easier to guess than LongBasic or LongComplex passwords. Basic passwords were easy to guess at all points. The main difference between Min_{auto} and Pros was that LongComplex passwords appear more vulnerable to the first 10^{12} guesses under Min_{auto} than Pros.

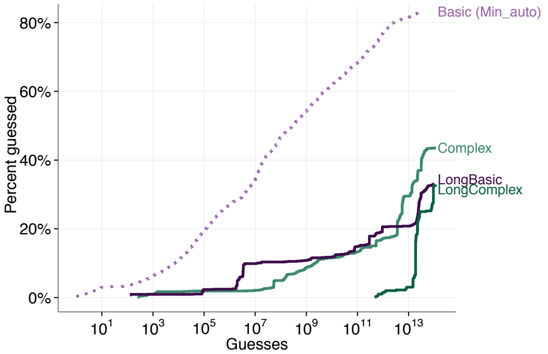


Figure 11: Pros' comparative success guessing each password. For reference, the dotted line represents the Min_{auto} guess across automated approaches for Basic passwords, which the Pros did not try to guess.

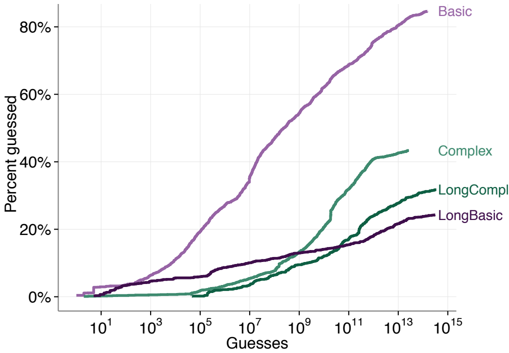


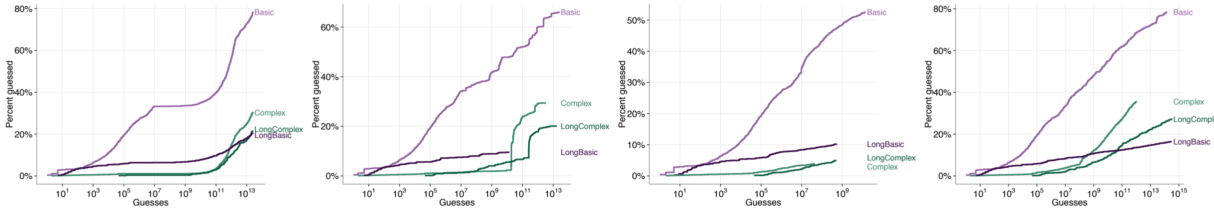
Figure 12: The guessability of all four password sets under Min_{auto}, representing the smallest guess number for each password across all four automated approaches.

Based on this data, a researcher comparing composition policies would likely reach similar conclusions using either professionals or a combination of automated approaches. As shown in Figure 13, we repeated this analysis using each of the four automated approaches in isolation. Against every approach, Basic passwords are easily guessable, and LongBasic passwords are comparatively vulnerable during early guessing. After trillions of guesses, Hashcat, PCFG, and JTR find Long Complex passwords more secure than Complex passwords. In each case, a researcher would come to similar conclusions about the relative strength of these password sets.

4.4.2 Per Individual Password

Analyses of the strength of individual passwords, in contrast, proved very sensitive to the guessing approach. Although one would expect different approaches to guess passwords at somewhat different times, many passwords' guess numbers varied by orders of magnitude across approaches. This state of affairs could cause a very weak password to be misclassified as very strong.

We examined per-password differences pairwise among JTR, Markov, and PCFG, using the same guess



(a) Hashcat guessability. (b) JTR guessability. (c) Markov guessability. (d) PCFG guessability.

Figure 13: The relative guessability of the four different password sets under each of the four automated cracking approaches considered in isolation. The research conclusions would be fairly similar in each case.

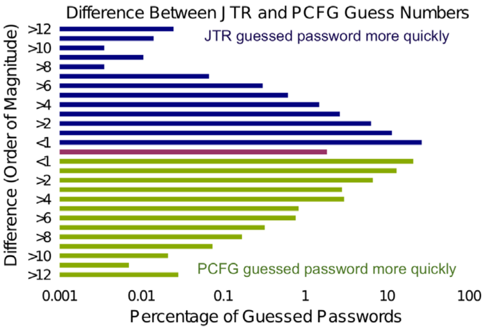


Figure 14: The % (log scale) of passwords guessed by JTR or PCFG whose guess numbers differed by a given order of magnitude. e.g., the blue > 6 bar represents passwords guessed by JTR more than 6, but no more than 7, orders of magnitude more quickly than by PCFG.

cutoff for each approach in a pair. Because Hashcat's early guesses were often unsuccessful, we exclude it from this analysis. Passwords not guessed by the guess cutoff were assigned a guess number one past the cutoff, lower-bounding differences between passwords guessed by one approach but not the other. For each password, we calculated the log_{10} of the ratio between guess numbers in the two approaches. For example, *iceman1232* was guess 595,300,840 for JTR and 61,554,045 for Markov, a 0.985 order of magnitude difference.

Among passwords guessed by JTR, PCFG, or both, 51% of passwords had guess numbers differing by more than an order of magnitude between approaches, indicating large variations in the resulting security conclusions. Alarmingly, some passwords had guess numbers differing by over 12 orders of magnitude (Figure 14). For example, *P@ssw0rd!* took JTR only 801 Complex guesses, yet PCFG never guessed it in our tests. Similarly, *1q2w3e4r5t6y7u8i* was the 29th LongBasic JTR guess, yet it was not among the 10^{14} such guesses PCFG made. In contrast, PCFG guessed *Abc@1993* after 48,670 guesses and *12345678password* after 130,555 guesses. JTR never guessed either password.

We found similar results in the two other pairwise comparisons. Among passwords guessed by Markov, PCFG, or both, 41% of guess numbers differed by at least one order of magnitude. In an extreme example, the passwords *1qaz!QAZ* and *1q2w3e4r5t6y7u8i* were among the first few hundred Markov guesses, yet not guessed by PCFG's guess cutoff. Conversely, *unitedstatesofamerica* was among PCFG's first few dozen LongBasic guesses, yet never guessed by Markov. For 37% of passwords, JTR and Markov guess numbers differed by at least one order of magnitude. Markov was particularly strong at guessing long passwords with predictable patterns. For instance, *password123456789*, *1234567890123456*, and *qwertyuiopasdfgh* were among Markov's first thirty guesses, yet JTR did not guess any of them by its cutoff.

Actionable takeaways As researchers and system administrators ask questions about password strength, they must consider whether their choice of cracking approach biases the results. When evaluating the strength of a large, heterogeneous password set, any of Hashcat, JTR, Markov, or PCFG—if configured effectively—provide fairly similar answers to research questions. Nonetheless, we recommend the more conservative strategy of calculating guessability using Min_{auto}.

In contrast, guessability results per-password can differ by many orders of magnitude between approaches even using the same training data. To mitigate these differences, we again recommend Min_{auto} for the increasingly important tasks of providing precise feedback on password strength to users and system administrators.

5 Conclusion

We report on the first broad, scientific investigation of the vulnerability of different types of passwords to guessing by an expert attacker and numerous configurations of off-the-shelf, automated approaches frequently used by researchers. We instrument these approaches, including both adversarial tools and academic research prototypes, to enable precise, guess-by-guess comparisons among automated approaches and between them and the expert.

We find that running a single guessing algorithm, particularly in its out-of-the-box configuration, often yields a very poor estimate of password strength. However, using several such algorithms, well-configured and in parallel, can be a good proxy for passwords' vulnerability to an expert attacker. We also find that while coarse-grained research results targeting heterogeneous sets of passwords are somewhat robust to the choice of (well-configured) guessing algorithm, many other analyses are not. For example, investigations of the effect on password strength of password characteristics, such as the number of character classes and the use of character substitutions, can reach different conclusions depending on the algorithm underlying the strength metric.

Finally, we hope our investigation of the effectiveness of many configurations of popular guessing approaches will help facilitate more accurate and easily reproducible research in the passwords research community. To that end, we have created a Password Guessability Service [9] that enables researchers to submit plaintext passwords and receive guessability analyses like those presented in this paper. We particularly encourage researchers investigating password-cracking algorithms to contribute to this service to improve the comparability of experiments.

Acknowledgments

We thank the authors of Hashcat and John the Ripper, Matt Marx, Jerry Ma, Weining Yang, Ninghui Li, KoreLogic, Michael Stroucken, Jonathan Bees, Chuck Cranor, and Dustin Heywood. This research was supported in part by NSF grants DGE-0903659 and CNS-1116776, and by a gift from Microsoft Research. It was also conducted with government support awarded by DoD, Air Force OSR, via the NDSEG Fellowship, 32 CFR 168a.

References

[1] The "web2" file of english words. http://www.bee-man.us/computer/grep/grep.htm#web2, 2004.

[2] BONNEAU, J. The Gawker hack: How a million passwords were lost. *Light Blue Touchpaper* Blog, December 2010. http://www.lightbluetouchpaper.org/2010/12/15/the-gawker-hack-how-a-million-passwords-were-lost/.

[3] BONNEAU, J. The science of guessing: Analyzing an anonymized corpus of 70 million passwords. In *Proc. IEEE Symp. Security & Privacy* (2012).

[4] BONNEAU, J. Statistical metrics for individual password strength. In *Proc. WPS* (2012).

[5] BONNEAU, J., HERLEY, C., VAN OORSCHOT, P. C., AND STAJANO, F. The quest to replace passwords: A framework for comparative evaluation of Web authentication schemes. In *Proc. IEEE Symp. Security & Privacy* (2012).

[6] BONNEAU, J., AND XU, R. Of contraseñas, sysmawt, and mìmǎ: Character encoding issues for web passwords. In *Proc. W2SP* (2012).

[7] BRODKIN, J. 10 (or so) of the worst passwords exposed by the LinkedIn hack. *Ars Technica*, June 2012.

[8] BURR, W. E., DODSON, D. F., AND POLK, W. T. Electronic authentication guideline. Tech. rep., NIST, 2006.

[9] CARNEGIE MELLON UNIVERSITY. Password guessability service. https://pgs.ece.cmu.edu, 2015.

[10] CASTELLUCCIA, C., DÜRMUTH, M., AND PERITO, D. Adaptive password-strength meters from Markov models. In *Proc. NDSS* (2012).

[11] CHOU, H.-C., LEE, H.-C., YU, H.-J., LAI, F.-P., HUANG, K.-H., AND HSUEH, C.-W. Password cracking based on learned patterns from disclosed passwords. *IJICIC* (2013).

[12] CHRYSANTHOU, Y. Modern password cracking: A hands-on approach to creating an optimised and versatile attack. Master's thesis, Royal Holloway, University of London, 2013.

[13] CURLYBOI. Hashtopus. http://hashtopus.nech.me/manual.html, 2009-.

[14] DAS, A., BONNEAU, J., CAESAR, M., BORISOV, N., AND WANG, X. The tangled web of password reuse. In *Proc. NDSS* (2014).

[15] DE CARNÉ DE CARNAVALET, X., AND MANNAN, M. From very weak to very strong: Analyzing password-strength meters. In *Proc. NDSS* (2014).

[16] DELL'AMICO, M., MICHIARDI, P., AND ROUDIER, Y. Password strength: An empirical analysis. In *Proc. INFOCOM* (2010).

[17] DEV, J. A. Usage of botnets for high speed md5 hash cracking. In *INTECH* (2013).

[18] DÜRMUTH, M., ANGELSTORF, F., CASTELLUCCIA, C., PERITO, D., AND CHAABANE, A. OMEN: Faster password guessing using an ordered markov enumerator. In *Proc. ESSoS* (2015).

[19] DÜRMUTH, M., CHAABANE, A., PERITO, D., AND CASTELLUCCIA, C. When privacy meets security: Leveraging personal information for password cracking. *CoRR* (2013).

[20] FAHL, S., HARBACH, M., ACAR, Y., AND SMITH, M. On The Ecological Validity of a Password Study. In *Proc. SOUPS* (2013).

[21] FORGET, A., CHIASSON, S., VAN OORSCHOT, P., AND BIDDLE, R. Improving text passwords through persuasion. In *Proc. SOUPS* (2008).

[22] GOODIN, D. Hackers expose 453,000 credentials allegedly taken from Yahoo service. *Ars Technica*, July 2012.

[23] GOODIN, D. Anatomy of a hack: How crackers ransack passwords like "qeadzcwrsfxv1331". *Ars Technica*, May 2013.

[24] GOODIN, D. "thereisnofatebutwhatwemake"-turbo-charged cracking comes to long passwords. *Ars Technica*, August 2013.

[25] GOODIN, D. Meet wordhound, the tool that puts a personal touch on password cracking. *Ars Technica*, August 2014.

[26] GOOGLE. Web 1T 5-gram version 1, 2006. http://www.ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC2006T13.

[27] HAIBLE. gperf. https://www.gnu.org/software/gperf/, 2010-.

[28] HAQUE, S. T., WRIGHT, M., AND SCIELZO, S. A study of user password strategy for multiple accounts. In *Proc. CODASPY* (2013).

[29] IMPERVA. Consumer password worst practices, 2010. http://www.imperva.com/docs/WP_Consumer_Password_Worst_Practices.pdf.

[30] InsidePro. PasswordsPro. http://www.insidepro.com/eng/passwordspro.shtml, 2003-.

[31] Kelley, P. G., Komanduri, S., Mazurek, M. L., Shay, R., Vidas, T., Bauer, L., Christin, N., Cranor, L. F., and Lopez, J. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In *Proc. IEEE Symp. Security & Privacy* (2012).

[32] Komanduri, S. *Modeling the adversary to evaluate password strengh with limited samples*. PhD thesis, Carnegie Mellon University, 2015.

[33] Komanduri, S., Shay, R., Cranor, L. F., Herley, C., and Schechter, S. Telepathwords: Preventing weak passwords by reading users' minds. In *Proc. USENIX Security* (2014).

[34] Komanduri, S., Shay, R., Kelley, P. G., Mazurek, M. L., Bauer, L., Christin, N., Cranor, L. F., and Egelman, S. Of passwords and people: Measuring the effect of password-composition policies. In *Proc. CHI* (2011).

[35] KoreLogic. "Crack Me If You Can" - DEFCON 2010. http://contest-2010.korelogic.com/rules.html, 2010-.

[36] KoreLogic. "Crack Me If You Can" - DEFCON 2013. http://contest-2013.korelogic.com, 2010-.

[37] KoreLogic. Pathwell topologies. *KoreLogic Blog*, 2014. https://blog.korelogic.com/blog/2014/04/04/pathwell_topologies.

[38] KoreLogic. "Analytical Solutions: Password Recovery Service. http://contest-2010.korelogic.com/prs.html, 2015.

[39] Love, D. Apple on iCloud breach: It's not our fault hackers guessed celebrity passwords. *International Business Times*, September 2014.

[40] Ma, J., Yang, W., Luo, M., and Li, N. A study of probabilistic password models. In *Proc. IEEE Symp. Security & Privacy* (2014).

[41] Marechal, S. Automatic wordlist mangling rule generation. *Openwall Blog*, 2012. http://www.openwall.com/presentations/Passwords12-Mangling-Rules-Generation/.

[42] Mazurek, M. L., Komanduri, S., Vidas, T., Bauer, L., Christin, N., Cranor, L. F., Kelley, P. G., Shay, R., and Ur, B. Measuring password guessability for an entire university. In *Proc. CCS* (2013).

[43] Morris, R., and Thompson, K. Password security: A case history. *CACM 22*, 11 (1979).

[44] MWR InfoSecurity. MWR InfoSecurity, 2014. https://www.mwrinfosecurity.com/.

[45] Narayanan, A., and Shmatikov, V. Fast dictionary attacks on passwords using time-space tradeoff. In *Proc. CCS* (2005).

[46] Openwall. Wordlists. http://download.openwall.net/pub/wordlists/, 2015.

[47] Percival, C. Stronger key derivation via sequential memory-hard function. In *Proc. BSD Conference* (2009).

[48] Perlroth, N. Adobe hacking attack was bigger than previously thought. *The New York Times Bits Blog*, October 2013.

[49] Peslyak, A. John the Ripper. http://www.openwall.com/john/, 1996-.

[50] Peslyak, A. John the Ripper. http://www.openwall.com/john/doc/MODES.shtml, 1996-.

[51] PHDays. "Hash Runner"– Positive Hack Days. http://2013.phdays.com/program/contests/, 2013.

[52] Provos, N., and Mazieres, D. A future-adaptable password scheme. In *Proc. USENIX* (1999).

[53] Rao, A., Jha, B., and Kini, G. Effect of grammar on security of long passwords. In *Proc. CODASPY* (2013).

[54] Schneier, B. MySpace passwords aren't so dumb. *Wired*, December 2012.

[55] SCOWL. Spell checker oriented word lists. http://wordlist.sourceforge.net, 2015.

[56] Shay, R., Komanduri, S., Durity, A. L., Huh, P. S., Mazurek, M. L., Segreti, S. M., Ur, B., Bauer, L., Cranor, L. F., and Christin, N. Can long passwords be secure and usable? In *Proc. CHI* (2014).

[57] Steube, J. Hashcat. https://hashcat.net/oclhashcat/, 2009-.

[58] Steube, J. Mask Attack. https://hashcat.net/wiki/doku.php?id=mask_attack, 2009-.

[59] Steube, J. Rule-based Attack. https://hashcat.net/wiki/doku.php?id=rule_based_attack, 2009-.

[60] Stobert, E., and Biddle, R. The password life cycle: User behaviour in managing passwords. In *Proc. SOUPS* (2014).

[61] Stricture Group. Password auditing services. http://stricture-group.com/services/password-audits.htm.

[62] Trustwave. eHarmony password dump analysis, June 2012. http://blog.spiderlabs.com/2012/06/eharmony-password-dump-analysis.html.

[63] Trustwave. 2014 business password analysis. *Password Research*, 2014.

[64] Trustwave Spiderlabs. SpiderLabs/KoreLogic-Rules. https://github.com/SpiderLabs/KoreLogic-Rules, 2012.

[65] Ur, B., Kelly, P. G., Komanduri, S., Lee, J., Maass, M., Mazurek, M., Passaro, T., Shay, R., Vidas, T., Bauer, L., Christin, N., and Cranor, L. F. How does your password measure up? The effect of strength meters on password creation. In *Proc. USENIX Security* (August 2012).

[66] Ur, B., Noma, F., Bees, J., Segreti, S. M., Shay, R., Bauer, L., Christin, N., and Cranor, L. F. "i added '!' at the end to make it secure": Observing password creation in the lab. In *Proc. SOUPS* (2015).

[67] Vance, A. If your password is 123456, just make it hackme. New York Times, 2010.

[68] Veras, R., Collins, C., and Thorpe, J. On the semantic patterns of passwords and their security impact. In *Proc. NDSS* (2014).

[69] Weir, M., Aggarwal, S., Collins, M., and Stern, H. Testing metrics for password creation policies by attacking large sets of revealed passwords. In *Proc. CCS* (2010).

[70] Weir, M., Aggarwal, S., Medeiros, B. d., and Glodek, B. Password cracking using probabilistic context-free grammars. In *Proc. IEEE Symp. Security & Privacy* (2009).

[71] Yazdi, S. H. Analyzing password strength and efficient password cracking. Master's thesis, The Florida State University, 2011.

[72] Zhang, Y., Monrose, F., and Reiter, M. K. The security of modern password expiration: An algorithmic framework and empirical analysis. In *Proc. CCS* (2010).

A Appendix

We provide additional measurements of how guessing approaches perform in different configurations. To support the ecological validity of our study, we also repeat analyses from the body of the paper on password sets leaked from RockYou and Yahoo. We provide these details in hopes of encouraging greater accuracy and reproducibility across measurements of password guessability.

A.1 Alternate PCFG Configurations

We tested four different PCFG configurations. As in the body of the paper, **PCFG** represents Komanduri's implementation of PCFG [32], which assigns letter strings probabilities based on their frequency in the training data and assigns unseen strings a non-zero probability. For consistency across approaches, we prepend all policy-compliant elements of the training data in lieu of enabling Komanduri's similar hybrid structures [32].

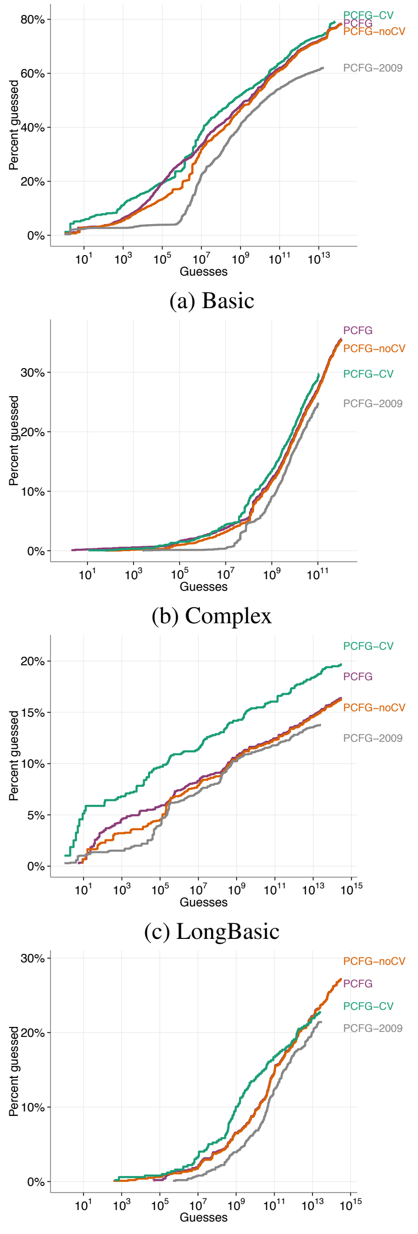
PCFG−noCV is the same as PCFG, but without the training data prepended. **PCFG−CV** is equivalent to PCFG−noCV except for using two-fold cross-validation. In each fold, we used half of the test passwords as additional training data, with a total weighting equal to the generic training data, as recommended by Kelley et al. [31]. **PCFG−2009** is our approximation of the original 2009 Weir et al. algorithm [70] in which alphabetic strings are assigned uniform probability and unseen terminals are a probability of zero.

As shown in Figure 15, prepending the training data and performing cross-validation both usually result in more efficient guessing, particularly for Long and LongBasic passwords. All three other configurations outperform the original PCFG−2009 implementation.

Figure 16 shows the guessability of the 350 passwords comprised only of digits across the Basic and LongBasic sets. Similar to the results for passwords of other common characteristics (Section 4.3.2), approaches differed. Of particular note is PCFG−2009, which plateaued at around 50% of such passwords guessed in fewer than 10 million guesses. Idiosyncratically, through 10^{14} guesses, it would never guess another password of this type because of the way it assigns probabilities.

A.2 Alternate JTR Configurations

We next separately analyze the sets of JTR mangling rules we combined in the body of the paper. **JTR_stock** represents the 23 default rules that come with JTR. **JTR_SpiderLabs** represents 5,146 rules published by KoreLogic during the 2010 DEF CON "Crack Me If You Can" password-cracking contest [35], later reordered for guessing efficiency by Trustwave Spiderlabs [64].



(a) Basic

(b) Complex

(c) LongBasic

(d) LongComplex

Figure 15: The guessing efficiency of the different PCFG configurations we tested.

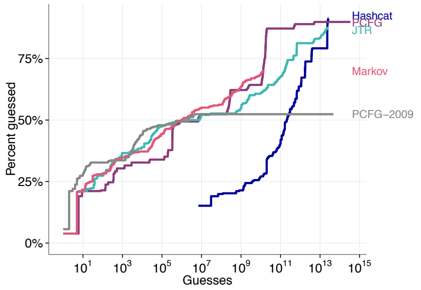


Figure 16: Guessing efficiency for the 350 Basic and LongBasic passwords composed entirely of digits.

As described in Section 3.3, our standard JTR configuration used JTR_stock followed by JTR_SpiderLabs. In isolation (Figure 17), JTR_stock rules were far more efficient on a guess-by-guess basis than JTR_SpiderLabs. Unfortunately, however, they quickly ran out of guesses. We exhausted JTR_stock in making fewer than 10^9 guesses for Basic passwords. More crucially, we made fewer than 10^5 guesses that were valid Complex passwords before exhausting these rules. Thus, any analysis of passwords that uses only the stock rules will vastly underestimate the guessability of passwords that contain (or are required to have) many different character classes.

The sharp jumps in the proportion of Complex and LongComplex passwords guessed by JTR_SpiderLabs result from one group of 13 rules. These rules capitalize the first letter, append digits, append special characters, and append both digits and special characters.

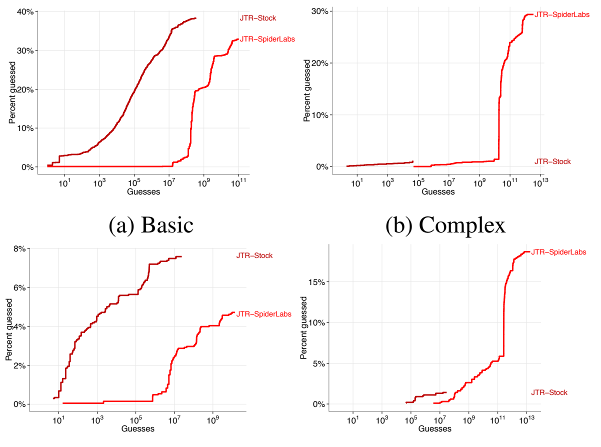
A.3 Alternate Hashcat Configurations

We tested eight Hashcat configurations and chose the one that best combined efficient early guessing with successfully continuing to guess passwords into the trillions of guesses. These configurations consist of four different sets of mangling rules, each with two different wordlists. The smaller wordlist was the same one we used in all other tests (Section 3.2). The larger wordlist augmented the same wordlist with all InsidePro wordlists[1] in descending frequency order and with duplicates removed.

Our four sets of mangling rules are the following:

Hashcat_best64: Although Hashcat does not have a default set of mangling rules, the Best64 mangling rules are often used analogously to JTR's stock rules.

Hashcat_generated2: Hashcat comes with a second set of mangling rules, "generated2." This set comprises 65,536 rules. Dustin Heywood of ATB Financial created them by randomly generating and then testing hundreds of millions of mangling rules over 6 months (2013-2014) on a 42-GPU cluster. The rules were optimized by Hashcat developers by removing semantic equivalents.



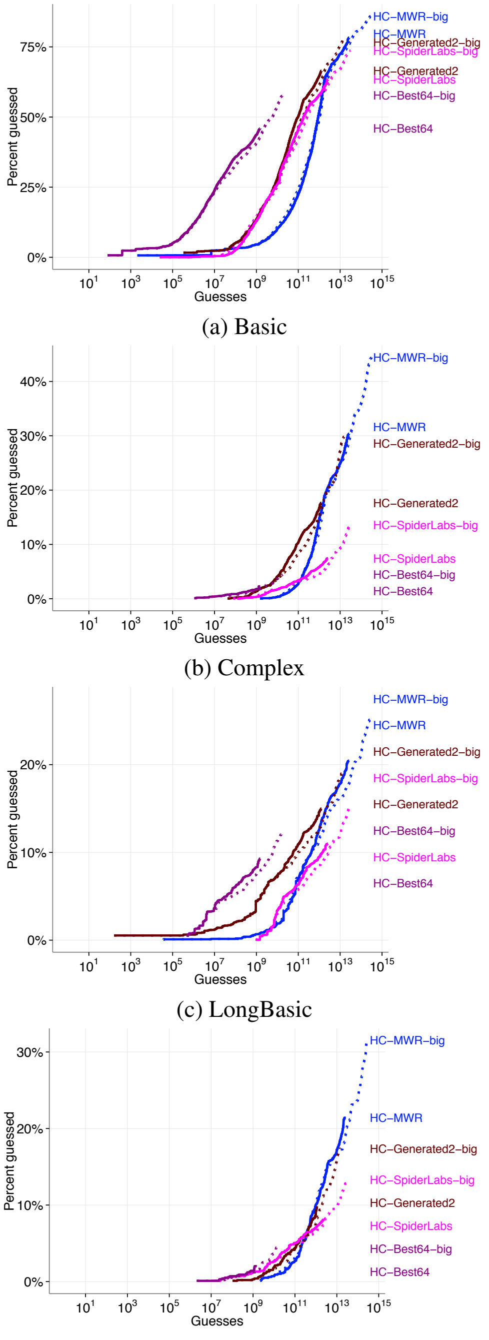
(a) Basic

(b) Complex

(c) LongBasic

(d) LongComplex

Figure 17: The guessing efficiency of JTR rules.



(a) Basic

(b) Complex

(c) LongBasic

(d) LongComplex

Figure 18: The guessing efficiency of Hashcat using four different sets of mangling rules. We tested each set with the wordlist used elsewhere in this paper, as well as a larger (**-big**) wordlist adding the InsidePro dictionaries.

[1] `http://www.insidepro.com/dictionaries.php`

Hashcat_SpiderLabs: We performed a manual translation to Hashcat of the SpiderLabs JTR rules (Section 3), which entailed removing clauses mandating minimum criteria; such rules are not permitted in oclHashcat.
Hashcat_MWR: We collaborated with with Matt Marx of MWR InfoSecurity to obtain the set of 1.4 million mangling rules he uses for password auditing [25, 44]. Following his suggestion, we augmented these rules with the aforementioned SpiderLabs rules.

Using the smaller wordlist, we exhausted all four sets of mangling rules. With the larger wordlist, we did not exhaust any set of rules. The curves in Figure 18 that use this larger dictionary have **-big** appended to the name and are graphed with dotted, rather than solid, lines.

We present the results of these eight configurations in Figure 18. True to their name, the Hashcat_best64 rules were the most efficient at guessing passwords. Unfortunately, they ran out of guesses using the smaller wordlist after only 10^9 guesses. For Complex and LongComplex passwords, Hashcat_best64 therefore guesses only a fraction of the number possible using the other sets of mangling rules, albeit in far fewer guesses. While not the most efficient guess-by-guess, the Hashcat_MWR rules eventually guessed the largest proportion of the different sets, most notably the Complex and LongComplex sets.

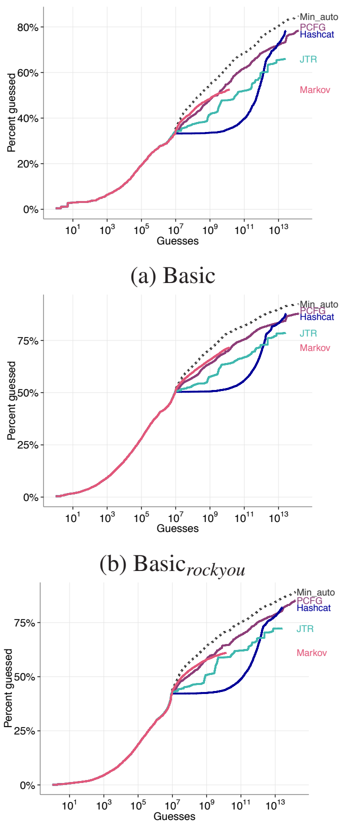
A.4 Ecological validity

To better understand how well our password sets, which we collected for research studies, compare to real plaintext passwords revealed in major password leaks, we compared the efficiency of the four automated cracking approaches in guessing Basic passwords, as well as the following two comparable sets of leaked passwords:
Basic$_{rockyou}$: 15,000 passwords randomly sampled from those containing 8+ characters in the RockYou gaming website leak of more than 32 million passwords [67]
Basic$_{yahoo}$: 15,000 passwords randomly sampled from those containing 8+ characters in the Yahoo! Voices leak of more than 450,000 passwords [22]

We found a high degree of similarity in the guessability of the Basic passwords collected for research and the leaked passwords. As shown in Figure 19, the four automated cracking approaches followed similar curves across the research passwords and the leaked passwords.

This similar guessability is notable because our analyses depend on using passwords collected by researchers for two reasons. First, no major password leak has contained passwords contained under strict composition requirements. Furthermore, in contracting experienced humans to attack the passwords, it was important to have them attack passwords they had not previously examined or tried to guess. Presumably, these experienced analysts would already have examined all major password leaks.

In the body of the paper, we reported how different approaches were impacted differently by the number of character classes contained in Basic passwords. When we repeated this analysis for Basic$_{rockyou}$ and Basic$_{yahoo}$ passwords, we found similar behavior (Figure 20). PCFG was more successful at guessing passwords containing two character classes, as opposed to only a single character class. PCFG only guesses strings that were found verbatim in its training data, which we hypothesize might be the cause of comparatively poor behavior for passwords of a single character class.



(a) Basic

(b) Basic$_{rockyou}$

(c) Basic$_{yahoo}$

Figure 19: The four automated cracking approaches targeting the Basic password set, 15,000 passwords sampled from the RockYou leak, and 15,000 passwords sampled from the Yahoo leak.

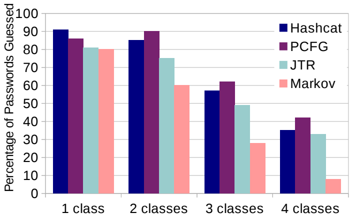


Figure 20: Combined percentage of Basic$_{rockyou}$ and Basic$_{yahoo}$ passwords each approach guessed by the number of character classes in the password.

Sound-Proof: Usable Two-Factor Authentication Based on Ambient Sound

Nikolaos Karapanos, Claudio Marforio, Claudio Soriente and Srdjan Čapkun
Institute of Information Security
ETH Zurich
{*firstname.lastname*}@*inf.ethz.ch*

Abstract

Two-factor authentication protects online accounts even if passwords are leaked. Most users, however, prefer password-only authentication. One reason why two-factor authentication is so unpopular is the extra steps that the user must complete in order to log in. Currently deployed two-factor authentication mechanisms require the user to interact with his phone to, for example, copy a verification code to the browser. Two-factor authentication schemes that eliminate user-phone interaction exist, but require additional software to be deployed.

In this paper we propose Sound-Proof, a usable and deployable two-factor authentication mechanism. Sound-Proof does not require interaction between the user and his phone. In Sound-Proof the second authentication factor is the proximity of the user's phone to the device being used to log in. The proximity of the two devices is verified by comparing the ambient noise recorded by their microphones. Audio recording and comparison are transparent to the user, so that the user experience is similar to the one of password-only authentication. Sound-Proof can be easily deployed as it works with current phones and major browsers without plugins. We build a prototype for both Android and iOS. We provide empirical evidence that ambient noise is a robust discriminant to determine the proximity of two devices both indoors and outdoors, and even if the phone is in a pocket or purse. We conduct a user study designed to compare the perceived usability of Sound-Proof with Google 2-Step Verification. Participants ranked Sound-Proof as more usable and the majority would be willing to use Sound-Proof even for scenarios in which two-factor authentication is optional.

1 Introduction

Software tokens on modern phones are replacing dedicated hardware tokens in two-factor authentication (2FA) mechanisms. Using a software token, in place of a hardware one, improves deployability and usability of 2FA. For service providers, 2FA based on software tokens results in a substantial reduction of manufacturing and shipping costs. From the user's perspective, there is no extra hardware to carry around and phones can accommodate software tokens from multiple service providers.

Despite the improvements introduced by software tokens, most users still prefer password-only authentication for services where 2FA is not mandatory [36, 12]. This is probably due to the extra burden that 2FA causes to the user [25, 51], since it typically requires the user to interact with his phone.

Recent work [14, 41] improves the usability of 2FA by eliminating the user-phone interaction. However, those proposals are not yet deployable as their requirements are not met by today's phones, computers or browsers.

In this paper, we focus on both the usability and deployability aspect of 2FA solutions. We propose Sound-Proof, a two-factor authentication mechanism that is transparent to the user and can be used with current phones and with major browsers without any plugin. In Sound-Proof the second authentication factor is the proximity of the user's phone to the computer being used to log in. When the user logs in, the two devices record the ambient noise via their microphones. The phone compares the two recordings, determines if the computer is located in the same environment, and ultimately decides whether the login attempt is legitimate or fraudulent.

Sound-Proof does not require the user to interact with his phone. The overall user experience is, therefore, close to password-only authentication. Sound-Proof works even if the phone is in the user's pocket or purse, and both indoors and outdoors. Sound-Proof can be easily deployed since it is compatible with current phones, computers and browsers. In particular, it works with any HTML5-compliant browser that implements the WebRTC API [24], which is currently being standardized by the W3C [15]. Chrome, Firefox and Opera already support WebRTC, Internet Explorer plans to support it [31], and we anticipate that other browsers will adopt it soon.

Similar to other approaches that do not require user-phone interaction nor a secure channel between the phone and the computer (e.g., [14]), Sound-Proof is not designed to protect against targeted attacks where the attacker is co-located with the victim and has the victim's login credentials. Our design choice favors usability and deployability over security and we argue that this can edge for larger user adoption.

We have implemented a prototype of Sound-Proof for both Android and iOS. Sound-Proof adds, on average, less than 5 seconds to a password-only login operation. This time is substantially shorter than the time overhead of 2FA mechanisms based on verification codes (roughly 25 seconds [50]). We also report on a user study we conducted which shows that users prefer Sound-Proof over Google 2-Step Verification [22].

In summary, we make the following contributions:

- We propose Sound-Proof, a novel 2FA mechanism that does not require user-phone interaction and is easily deployable. The second authentication factor is the proximity of the user's phone to the computer from which he is logging in. Proximity of the two devices is verified by comparing the ambient audio recorded via their microphones. Recording and comparison are transparent to the user.

- We implement a prototype of our solution for both Android and iOS. We use the prototype to evaluate the effectiveness of Sound-Proof in a number of different settings. We show that Sound-Proof works even if the phone is in the user's pocket or purse and that it fares well both indoors and outdoors.

- We conducted a user study to compare the perceived usability of Sound-Proof and Google 2-Step Verification. Participants ranked the usability of Sound-Proof higher than the one of Google 2-Step Verification, with a statistically significant difference. More importantly, we found that most participants would use Sound-Proof even if 2FA were optional.

The rest of the paper is organized as follows. Section 2 details our assumptions and goals while Section 3 reviews alternative approaches and discusses why they do not fulfill our objectives. Section 4 provides an overview on audio similarity techniques. We present Sound-Proof in Section 5 and its prototype implementation in Section 6. Section 7 evaluates Sound-Proof, while Section 8 reports on our user study. We discuss limitations and ways to further improve Sound-Proof in Section 9. Section 10 reviews related work and Section 11 concludes the paper.

2 Assumptions and Goals

System Model. We assume the general settings of browser-based web authentication. The user has a username and a password to authenticate to a web server. The server implements a 2FA mechanism that uses software tokens on phones.

The user points his browser to the server's webpage and enters his username and password. The server verifies the validity of the password and challenges the user to prove possession of the second authentication factor.

Threat Model. We assume a remote adversary who has obtained the victim's username and password via phishing, leakage of a password database, or via other means. His goal is to authenticate to the server on behalf of the user. In particular, the adversary visits the server's webpage and enters the username and password of the victim. The attack is successful if the adversary convinces the server that he also holds the second authentication factor of the victim.

We further assume that the adversary cannot compromise the victim's phone. If the adversary gains control of the platform where the software token runs, then the security of any 2FA scheme reduces to the security of password-only authentication. Also, the adversary cannot compromise the victim's computer. The compromise of the computer allows the adversary to mount a Man-In-The-Browser attack [34] and hijack the victim's session with the server, therefore defeating any 2FA mechanism.

We do not address targeted attacks where the adversary is co-located with the victim. 2FA mechanisms that do not require the user to interact with his phone cannot protect against targeted, co-located attacks. For example, if 2FA uses unauthenticated short-range communication [14], a co-located attacker can connect to the victim's phone and prove possession of the second authentication factor to the server. We argue that targeted, co-located attacks are less common than non-selective, remote attacks. Furthermore, any 2FA mechanism may not warrant protection against powerful attackers. For example, if 2FA uses verification codes, a determined attacker may gain physical access to the phone or read the code from a distance [6, 7, 37].

We do not consider Man-In-The-Middle adversaries. Client authentication is not sufficient to defeat MITM attacks in the context of web applications [29]. We also do not address active phishing attacks where the attacker lures the user into visiting a phishing website and relays the stolen credentials to the legitimate website in real-time. Such attacks can be thwarted by having the phone detect the phishing domain [14, 35]. This requires short-range communication between the phone and the browser. However, seamless short-range communication between the phone and the browser is currently not possible.

Design Goals.

- *Usability.* Users should authenticate using only their username and password as in password-only authentication. In particular, users should not be asked to interact with their phone — not even to pick up the phone or take it out of a pocket or purse.
- *Deployability.* The 2FA mechanism should work with common smartphones, computers and browsers. It should not require additional software on the computer or the installation of browser plugins. A plugin-based solution limits the usability of the system because (*i*) a different plugin may be required for each server, and (*ii*) the user must install the plugin every time he logs in from a computer for the first time. The mechanism should also work on a wide range of smartphones. We therefore discard the use of special hardware on the phone like NFC chips or biometric sensors.

3 Alternative Approaches

In this section we discuss traditional 2FA mechanisms, as well as 2FA proposals which minimize the user-phone interaction. For each solution we argue why it fails to meet our usability and deployability goals.

3.1 Traditional 2FA

Hardware Tokens. Hardware tokens range from the RSA SecurID [17] to recent dongles [53] that comply with the FIDO U2F [20] standard for universal 2FA. Such solutions require the user to carry and interact with the token and may be expensive to deploy because the service provider must ship one token per customer.

Software Tokens. Google 2-Step Verification [22] is an example of 2FA based on verification codes, that uses software tokens on phones. The verification code is retrieved either from an application running on the phone or via SMS. Such mechanisms require the user to copy the verification code from the phone to the browser.

Duo Push [16] and Encap Security [18] prompt the user with a push message on his phone with information on the current login attempt. Both solutions still require the user to interact with his phone to authorize the login.

3.2 Reduced-Interaction 2FA

Short-range Radio Communication. PhoneAuth [14] is a 2FA proposal that leverages unpaired Bluetooth communication between the browser and the phone, in order to eliminate user-phone interaction. The Bluetooth channel enables the server (through the browser) and the phone to engage in a challenge-response protocol which provides the second authentication factor. Similarly, [35] and [41] also leverage Bluetooth communication between the browser and the phone.

These schemes require the browser to expose a Bluetooth API that is currently not available on any browser. A specification to expose a Bluetooth API in browsers has been proposed by the Web Bluetooth Community Group [49]. It is unclear whether the proposed API will support the unauthenticated RFCOMM or similar functionality which is required to enable seamless connectivity between the browser and the phone. However, if the Bluetooth connection is unauthenticated, an adversary equipped with a powerful antenna may connect to the victim's phone from afar [52] and login on behalf of the user, despite 2FA.

Authy [5] is another approach that allows for seamless 2FA using Bluetooth communication between the computer and the phone. Authy, however, requires extra software on the computer.

As an alternative to Bluetooth, the browser and the phone can communicate over WiFi [41]. This approach only works when both devices are on the same network. Shirvanian et al., [41] use extra software on the computer to virtualize the wireless interface and create a software access point (AP) with which the phone needs to be associated. The user has to perform this setup procedure every time he uses a new computer to log in. Their solution also requires a phone application listening for incoming connections in the background, which is currently not possible on iOS.

Finally, the browser and the phone can communicate over NFC. NFC hardware is not commonly found in commodity computers, and current browsers do not expose APIs to access NFC. Furthermore, a solution based on NFC would not completely remove user-phone interaction because the user would still need to hold his phone close to the computer.

We acknowledge that 2FA mechanisms that employ direct communication between the browser and the phone may provide additional security against remote attackers. For example, the phone can detect if the user tries to login on a phishing website and block the attempt [14, 35]. The scheme in [41] further resists offline dictionary attacks against compromised hashed password databases. Nevertheless, none of such solutions can be deployed for the reasons we discussed above.

Near-ultrasound. SlickLogin [23] minimizes the user-phone interaction transferring the verification code from the computer to the phone using near-ultrasounds. The idea is to use spectrum frequencies that are non-audible for the majority of the population but that can be reproduced by the speakers of commodity computers ($>$ 18kHz). Using non-audible frequencies accommodates for scenarios where users may not want their devices to make audible noise. Due to their size, the speakers of commodity computers can only produce highly directional near-ultrasound frequencies [39]. Near-ultrasound signals also attenuate faster, when compared to sounds in the lower part of the spectrum (< 18kHz) [3, 28]. With SlickLogin, the user must ensure that the speaker volume is at a sufficient level during login. Also, login will fail if a headset is plugged into the laptop. Finally, this approach may not work in scenarios where there is in-band noise (e.g., when listening to music or in cafes) [28]. We also note that a solution based on near-ultrasounds may result unpleasant for young people and animals that are capable of hearing sounds above 18kHz [38].

Location Information. The server can check if the computer and the phone are co-located by comparing their GPS coordinates. GPS sensors are available on all modern phones but are rare on commodity computers. If the computer from which the user logs in has no GPS sensor, it can use the geolocation API exposed by some browsers [32]. Nevertheless, information retrieved via the geolocation API may not be accurate, for example when the device is behind a VPN or it is connected to a large managed network (such as enterprise or university networks). Furthermore, geolocation information can be easily guessed by an adversary. For example, assume the adversary knows the location of the victim's workplace and uses that location as the second authentication factor. This attack is likely to succeed during working hours since the victim is presumably at his workplace.

Other Sensors. A 2FA mechanism can combine the readings of multiple sensors that measure ambient characteristics, such as temperature, concentration of gases in the atmosphere, humidity, and altitude, as proposed in [42]. These combined sensor modalities can be used to verify the proximity between the computer through which the user is trying to login and his phone. However, today's computers and phones lack the hardware sensors that are required for such an approach to work.

4 Background on Sound Similarity

The problem of determining the similarity of two audio samples is close to the problem of audio fingerprinting and automatic media retrieval [13]. In media retrieval, a noisy recording is matched against a database of reference samples. This is done by extracting a set of relevant features from the noisy recording and comparing them against the features of the reference samples. The extracted features must be robust to, for example, background noise and attenuation. Bark Frequency Cepstrum Coefficients [26], wavelets [8] or peak frequencies [48] have been proposed as robust features for automatic media retrieval. Such techniques focus mostly on the frequency domain representation of the samples because they deal with time-misaligned samples. In our scenario, we compare two quasi-aligned samples (the offset is less than 150ms) and we therefore can also extract relevant information from their time domain representations.

In order to consider both time domain and frequency domain information of the recordings, we use one-third octave band filtering and cross-correlation.

One-third Octave Bands. Octave bands split the audible range of frequencies (roughly from 20Hz to 20kHz) in 11 non-overlapping bands where the ratio of the highest in-band frequency to the lowest in-band frequency is 2 to 1. Each octave is represented by its center frequency, where the center frequency of a particular octave is twice the center frequency of the previous octave. One-third octave bands split the first 10 octave bands in three and the last octave band in two, for a total of 32 bands. One-third octave bands are widely used in acoustics and their frequency ranges have been standardized [44]. The center frequency of the lowest band is 16Hz (covering from 14.1Hz to 17.8Hz) while the center frequency of the highest band is 20kHz (covering from 17780Hz to 22390Hz). In the following we denote with $B = [lb - hb]$ a set of contiguous one-third octave bands, from the band that has its central frequency at lbHz, to the band that has its central frequency at hbHz.

Splitting a signal in one-third octave bands provides high frequency resolution information of the original signal, while keeping its time-domain representation.

Cross-correlation. Cross-correlation is a standard measure of similarity between two time series. Let x, y denote two signals represented as n-points discrete time series,[1] the cross-correlation $c_{x,y}(l)$ measures their similarity as a function of the lag $l \in [0, n-1]$ applied to y:

$$c_{x,y}(l) = \sum_{i=0}^{n-1} x(i) \cdot y(i-l)$$

where $y(i) = 0$ if $i < 0$ or $i > n-1$.

To accommodate for different amplitudes of the two signals, the cross correlation can be normalized as:

$$c'_{x,y}(l) = \frac{c_{x,y}(l)}{\sqrt{c_{x,x}(0) \cdot c_{y,y}(0)}}$$

where $c_{x,x}(l)$ is known as auto-correlation.

The normalization maps $c'_{x,y}(l)$ in $[-1, 1]$. A value of $c'_{x,y}(l) = 1$ indicates that at lag l, the two signals have the same shape even if their amplitudes may be different; a value of $c'_{x,y}(l) = -1$ indicates that the two signals have the same shape but opposite signs. Finally, a value of $c'_{x,y}(l) = 0$ shows that the two signals are uncorrelated.

If the actual lag between the two signals is unknown, we can discard the sign information and use the absolute value of the maximum cross-correlation $\hat{c}_{x,y}(l) = \max_l(|c'_{x,y}(l)|)$ as a metric of similarity ($0 \leq \hat{c}_{x,y}(l) \leq 1$).

The computation overhead of $c_{x,y}(l)$ can be decreased by leveraging the cross-correlation theorem and computing $c_{x,y}(l) = F^{-1}(F(x)^* \cdot F(y))$, where $F()$ denotes the

[1] For simplicity we assume both series to have the same length.

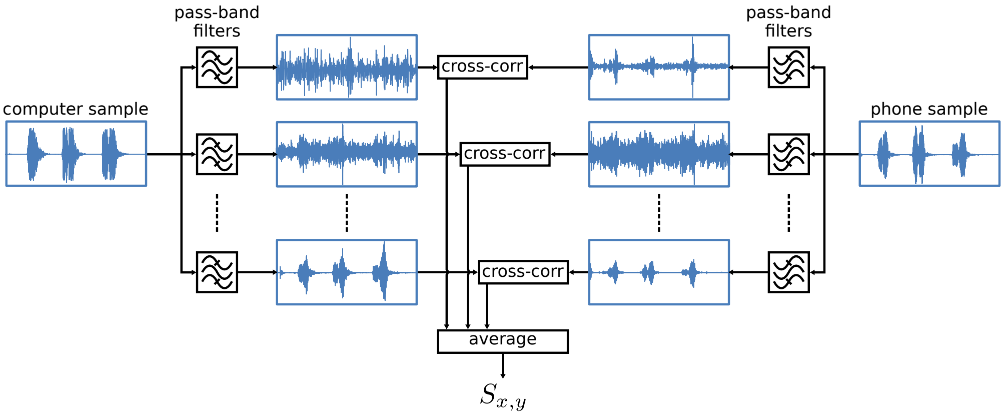


Figure 1: Block diagram of the function that computes the similarity score between two samples. The computation takes place on the phone. If $S_{x,y} > \tau_C$ and the average power of the samples is greater than τ_{dB}, the phone judges the login attempt as legitimate.

discrete Fourier transform and the asterisk denotes the complex conjugate.

5 Sound-Proof Architecture

The second authentication factor of Sound-Proof is the proximity of the user's phone to the computer being used to log in. The proximity of the two devices is determined by computing a similarity score between the ambient noise captured by their microphones. For privacy reasons we do not upload cleartext audio samples to the server. In our design, the computer encrypts its audio sample under the public key of the phone. The phone receives the encrypted sample, decrypts it, and computes the similarity score between the received sample and the one recorded locally. Finally, the phone tells the server whether the two devices are co-located or not. Note that the phone never uploads its recorded sample to the server. Communication between the computer and the phone goes through the server. We avoid short-range communication between the phone and the computer (e.g., via Bluetooth) because it requires changes to the browser or the installation of a plugin.

5.1 Similarity Score

Figure 1 shows a block diagram of the function that computes the similarity score. Each audio signal is input to a bank of pass-band filters to obtain n signal components, one per each of the one-third octave bands that we take into account. Let x_i be the signal component for the i-th one-third octave band of signal x. The similarity score is the average of the maximum cross-correlation over the pairs of signal components x_i, y_i:

$$S_{x,y} = \frac{1}{n} \sum_{i=1}^{i=n} \hat{c}_{x_i,y_i}(l)$$

where l is bounded between 0 and ℓ_{max}.

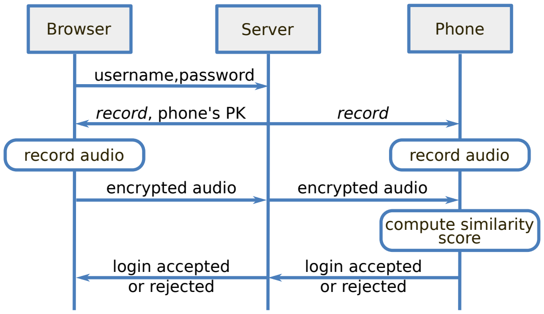


Figure 2: Sound-Proof authentication overview. At login, the phone and the computer record ambient noise with their microphones. The phone computes the similarity score between the two samples and returns the result to the server.

5.2 Enrollment and Login

Similar to other 2FA mechanisms based on software tokens, Sound-Proof requires the user to install an application on his phone and to bind the application to his account on the server. This one-time operation can be carried out using existing techniques to enroll software tokens, e.g., [22]. We assume that, at the end of the phone enrollment procedure, the server receives the unique public key of the application on the user's phone and binds that public key to the account of that user.

Figure 2 shows an overview of the login procedure. The user points the browser to the URL of the server and enters his username and password. The server retrieves the public key of the user's phone and sends it to the browser. Both the browser and the phone start recording through their local microphones for t seconds. During recording, the two devices synchronize their clocks with the server. When recording completes, each device adjusts the timestamp of its sample taking into account the clock difference with the server. The browser encrypts the audio sample under the phone's public key and sends it to the phone, using the server as a proxy. The phone decrypts the browser's sample and compares it against the one recorded locally. If the average power of both samples is above τ_{dB} and the similarity score is above τ_C, the phone concludes that it is co-located with the computer from which the user is logging in and informs the server that the login is legitimate.

The procedure is completely transparent to the user if the environment is sufficiently noisy. In case the environment is quiet, Sound-Proof requires the user to generate some noise, for example by clearing his throat.

5.3 Security Analysis

Remote Attacks. The security of Sound-Proof stems from the attacker's inability to guess the sound in the victim's environment at the time of the attack.

Let x be the sample recorded by the victim's phone and let y be the sample submitted by the attacker. A successful impersonation attack requires the average power of both signals to be above τ_{dB}, and each of the one-third octave band components of the two signals to be highly correlated. That is, the two samples must satisfy $Pwr(x) > \tau_{dB}$, $Pwr(y) > \tau_{dB}$ and $S_{x,y} > \tau_C$ with $l < \ell_{max}$.

We bound the lag l between 0 and ℓ_{max} to increase the security of the scheme against an adversary that successfully guesses the noise in the victim's environment at the time of the attack. Even if the adversary correctly guesses the noise in the victim's environment and can submit a similar audio sample, the two samples must be synchronized with an error smaller than ℓ_{max}. We also reject audio pairs where either sample has an average power below the threshold τ_{dB}. This is in order to prevent an impersonation attack when the victim's environment is quiet (e.g., while the victim is sleeping).

Quantifying the entropy of ambient noise, and hence the likelihood of the adversary guessing the signal recorded by the victim's phone, is a challenging task. Results are dependent on the environment, the language spoken by the victim, his gender or age to cite a few. In Section 7 we provide empirical evidence that Sound-Proof can discriminate between legitimate and fraudulent logins, even if the adversary correctly guesses the type of environment where the victim is located.

Co-located Attacks. Sound-Proof cannot withstand attackers who are co-located with the victim. A co-located attacker can capture the ambient sound in the victim's environment and thus successfully authenticate to the server, assuming that he also knows the victim's password. Sound-Proof shares this limitation with other 2FA mechanisms that do not require the user to interact with his phone and do not assume a secure channel between the phone and the computer (e.g., [14]). Resistance to co-located attackers requires either a secure phone-to-computer channel (as in [5, 41]) or user-phone interaction (as in [16, 22]). However, both techniques impose a significant usability burden.

6 Prototype Implementation

Our implementation works with Google Chrome (tested with version 38.0.2125.111), Mozilla Firefox (tested with version 33.0.2) and Opera (tested with version 25.0.1614.68). We anticipate the prototype to work with different versions of these browsers, as long as they implement the `navigator.getUserMedia()` API of WebRTC. We tested the phone application both on Android and on iOS. For Android, on a Samsung Galaxy S3, a Google Nexus 4 (both running Android version 4.4.4), a Sony Xperia Z3 Compact and a Motorola Nexus 6 (running Android version 5.0.2 and 5.1.1, respectively). We also tested different iPhone models (iPhone 4, 5 and 6) running iOS version 7.1.2 on the iPhone 4, and iOS version 8.1 on the newer models. The phone application should work on different phone models and with different OS versions without major modifications.

Web Server and Browser. The server component is implemented using the CherryPy [45] web framework and MySQL database. We use WebSocket [19] to push data from the server to the client. The client-side (browser) implementation is written entirely in HTML and JavaScript. Encryption of the audio recording uses AES256 with a fresh symmetric key; the symmetric key is encrypted under the public key of the phone using RSA2048. We use the HTML5 WebRTC API [15, 24]. In particular, we use the `navigator.getUserMedia()` API to access the local microphone from within the browser. Our prototype does not require browser code modifications or plugins.

Software Token. We implement the software token as an Android application as well as an iOS application. The mobile application stays idle in the background and is automatically activated when a push notification arrives. Push messages for Android and iOS use the Google GCM (Google Cloud Messaging) APIs [21] and Apple's APN (Apple Push Notifications) APIs [2] (in particular the silent push notification feature), respectively. Phone to server communication is protected with TLS.

Most of the Android code is written in Java (Android SDK), while the component that processes the audio samples is written in C (Android NDK). In particular, we use the ARM Ne10 library, based on the ARM NEON engine [4] to optimize vector operations and FFT computations. The iOS application is written in Objective-C and uses Apple's vDSP package of the Accelerate framework [1], in order to leverage the ARM NEON technology for vector operations and FFT computations. On both mobile platforms we parallelize the computation of the similarity score across the available processor cores.

Operations	Mean (ms)	Std.Dev.
Recording	3000	—
Similarity score computation	642	171
Cryptographic operations	118	15
Networking		
WiFi	978	135
Cellular	1243	209

Table 1: Overhead of the Sound-Proof prototype. On average it takes 4677ms (± 181ms) over WiFi and 4944ms (± 233ms) over Cellular to complete the 2FA verification.

Time Synchronization. Sound-Proof requires the recordings from the phone and the computer to be synchronized. For this reason, the two devices run a simple time-synchronization protocol (based on the Network Time Protocol [33]) with the server. The protocol is implemented over HTTP and allows each device to compute the difference between the local clock and the one of the server. Each device runs the time-synchronization protocol with the server while it is recording via its microphone. When recording completes, each device adjusts the timestamp of its sample taking into account the clock difference with the server.

Run-time Overhead. We compute the run-time overhead of Sound-Proof when the phone is connected either through WiFi or through the cellular network. We run 1000 login attempts with a Google Nexus 4 for each connection type, and we measure the time from the moment the user submits his username and password to the time the web server logs the user in. On average it takes 4677ms (± 181ms) over WiFi and 4944ms (± 233ms) over Cellular to complete the 2FA verification. Table 1 shows the average time and the standard deviation of each operation. The recording time is set to 3 seconds. The similarity score is computed over the set of one-third octave bands $B = [50\text{Hz} - 4\text{kHz}]$. (Section 7.1 discusses the selection of the band set.) After running the time-synchronization protocol, the resulting clock difference was, on average, 42.47ms (± 30.35ms).

7 Evaluation

Data Collection. We used our prototype to collect a large number of audio pairs. We set up a server that supported Sound-Proof. Two subjects logged in using Google Chrome[2] over 4 weeks. At each login, the phone and the computer recorded audio through their microphones for 3 seconds. We stored the two audio samples for post-processing.

Login attempts differed in the following settings. *Environment:* an office at our lab with either no ambient noise (labelled as Office) or with the computer playing music (Music); a living-room with the TV on (TV); a lecture hall while a faculty member was giving a lecture (Lecture); a train station (TrainStation); a cafe (Cafe). *User activity:* being silent, talking, coughing, or whistling. *Phone position:* on a table or a bench next to the user, in the trouser pocket, or in a purse. *Phone model:* Apple iPhone 5 or Google Nexus 4. *Computer model:* Mac Book Pro "Mid 2012" running OS X10.10 Yosemite or Dell E6510 running Windows 7.

At the end of the 4 weeks we had collected between 5 and 15 login attempts per each setting, totaling 2007 login attempts (4014 audio samples).

7.1 Analysis

We used the collected samples to find the configuration of system parameters (i.e., τ_{dB}, ℓ_{max}, B, and τ_C) that led to the best results in terms of False Rejection Rate (FRR) and the False Acceptance Rate (FAR). A false rejection occurs when a legitimate login is rejected. A false acceptance occurs when a fraudulent login is accepted. A fraudulent login is accepted if the sample submitted by the attacker and the sample recorded by the victim's phone have a similarity score greater than τ_C, and if both samples have an average power greater than τ_{dB}.

To compute the FAR, we used the following strategy. For each phone sample collected by one of the subjects (acting as the victim), we use all the computer samples collected by the other subject as the attacker's samples. We then switch the roles of the two subjects and repeat the procedure. The total number of victim–adversary sample pairs we considered was 2,045,680.

System Parameters. We set the average power threshold τ_{dB} to 40dB which, based on our measurements, is a good threshold to reject silence or very quiet recordings like the sound of a fridge buzzing or the sound of a clock ticking. Out of 2007 login attempts we found 5 attempts to have an average power of either sample below 40dB and we discard them for the rest of the evaluation.

We set ℓ_{max} to 150ms because this was the highest clock difference experienced while testing our time-synchronization protocol (see Section 6).

An important parameter of Sound-Proof is the set B of one-third octave bands to consider when computing the similarity score described in Section 5.1. The goal is to select a spectral region that (*i*) includes most common sounds and (*ii*) is robust to attenuation and directionality of audio signals. We discarded bands below 50Hz to remove very low-frequency noises. We also discarded bands above 8kHz, because these frequencies are attenuated by fabric and they are not suitable for scenarios where the phone is in a pocket or a purse. We tested all sets of one-third octave bands $B = [x - y]$ where x ranged from 50Hz to 100Hz and y ranged from 630Hz to 8kHz.

[2] We used Google Chrome since it is currently the most popular browser [43]. We have also tested Sound-Proof with other browsers and have experienced similar performance (see Section 9).

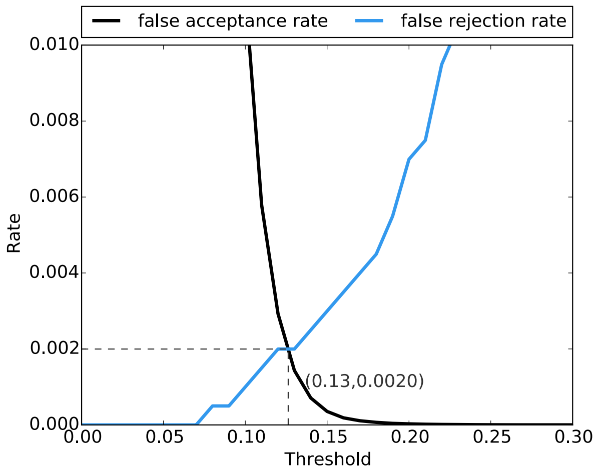
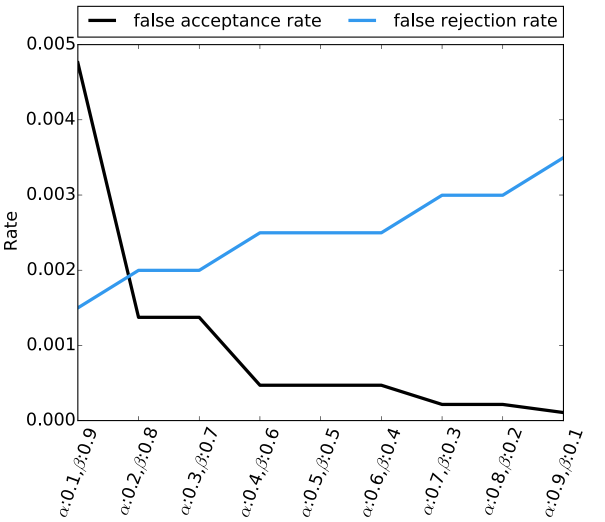


Figure 3: False Rejection Rate and False Acceptance Rate as a function of the threshold τ_C for $B = [50\text{Hz} - 4\text{kHz}]$. The Equal Error Rate is 0.0020 at $\tau_C = 0.13$.

We found the smallest Equal Error Rate (ERR, defined as the crossing point of FRR and FAR) when using $B = [50\text{Hz} - 4\text{kHz}]$. Figure 3 shows the FRR and FAR using this set of bands where the ERR is 0.0020 at $\tau_C = 0.13$. We experienced worse results with one-third octave bands above 4kHz. This was likely due to the high directionality of the microphones found on commodity devices when recoding sounds at those frequencies [47].

We also computed the best set of one-third octave bands to use in case usability and security are weighted differently by the service provider.[3] In particular, we computed the sets of bands that minimized $f = \alpha \cdot FRR + \beta \cdot FAR$, for $\alpha \in [0.1, \ldots, 0.9]$ and $\beta = 1 - \alpha$. Figure 4(b) shows the set of bands that provided the best results for each configuration of α and β. As before, we experienced better results with bands below 4kHz. Figure 4(a) plots the FRR and FAR against the possible values of α and β. We stress that the set of bands may differ across two different points on the x-axis.

Experiments in the remaining of this section were run with the configuration of the parameters that minimized the ERR to 0.0020: $\tau_{dB} = 40\text{dB}$, $\ell_{max} = 150\text{ms}$, $B = [50\text{Hz} - 4\text{kHz}]$, and $\tau_C = 0.13$.



(a) False Rejection Rate and False Acceptance Rate when usability and security have different weights.

	B	τ_c
$\alpha = 0.1,\ \beta = 0.9$	$[80\text{Hz} - 2500\text{Hz}]$	0.12
$\alpha = 0.2,\ \beta = 0.8$	$[50\text{Hz} - 2500\text{Hz}]$	0.14
$\alpha = 0.3,\ \beta = 0.7$	$[50\text{Hz} - 2500\text{Hz}]$	0.14
$\alpha = 0.4,\ \beta = 0.6$	$[50\text{Hz} - 800\text{Hz}]$	0.19
$\alpha = 0.5,\ \beta = 0.5$	$[50\text{Hz} - 800\text{Hz}]$	0.19
$\alpha = 0.6,\ \beta = 0.4$	$[50\text{Hz} - 800\text{Hz}]$	0.19
$\alpha = 0.7,\ \beta = 0.3$	$[50\text{Hz} - 1000\text{Hz}]$	0.2
$\alpha = 0.8,\ \beta = 0.2$	$[50\text{Hz} - 1000\text{Hz}]$	0.2
$\alpha = 0.9,\ \beta = 0.1$	$[50\text{Hz} - 1250\text{Hz}]$	0.21

(b) One-third octave bands and similarity score threshold.

Figure 4: Minimizing $f = \alpha \cdot FRR + \beta \cdot FAR$, for $\alpha \in [0.1, \ldots, 0.9]$ and $\beta = 1 - \alpha$.

7.2 False Rejection Rate

In the following we evaluate the impact of each setting that we consider (environment, user activity, phone position, phone model, and computer model) on the FRR. Figures 5 and 6 show a box and whisker plot for each setting. The whiskers mark the 5th and the 95th percentiles of the similarity scores. The boxes show the 25th and 75th percentiles. The line and the solid square within each box mark the median and the average, respectively. A gray line marks the similarity score threshold ($\tau_C = 0.13$) and each red dot in the plots denotes a login attempt where the similarity score was below that threshold (i.e., a false rejection).

Environment. Figure 5 shows the similarity scores for each environment. Sound-Proof fares equally well indoors and outdoors. We did not experience rejections of legitimate logins for the Music (over 432 logins), the Lecture (over 122 logins), and the TV (over 430 logins) environments. The FRR was 0.003 (1 over 310 logins) for Office, 0.003 (1 over 370 logins) for TrainStation, and 0.006 (2 over 338 logins) for Cafe.

User Activity. Figure 6(a) shows the similarity scores for different user activities. In general, if the user makes any noise the similarity score improves. We only experienced a few rejections of legitimate logins when the

[3] For example, a social network provider may value usability higher than security.

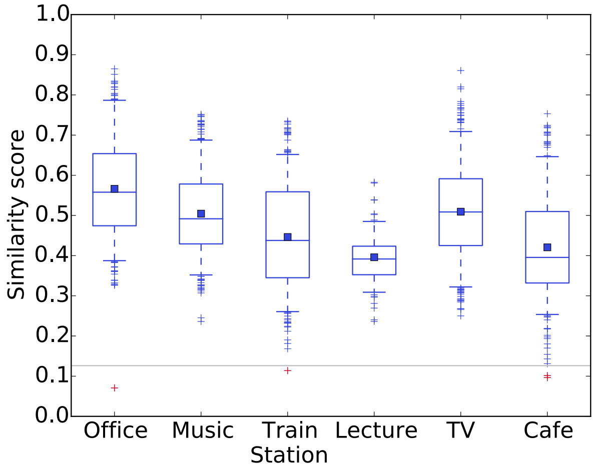


Figure 5: Impact of the environment on the False Rejection Rate.

	False Acceptance Rate		
	SC-SP	SC-DP	DC-DP
TV channel 1	1	0.1	0.1
TV channel 2	1	1	0
TV channel 3	1	0	-
TV channel 4	1	0	-
Web radio 1	1	0	0.4
Web radio 2	0.1	0.8	0.8
Web TV 1	0	0	0
Web TV 2	0	0	0

Table 2: False Acceptance Rate when the adversary and the victim devices record the same broadcast media. SC-SP stands for "same city and same Internet/cable provider", SC-DP stands for "same city but different Internet/cable providers", DC-DP stands for "different cities and different Internet/cable providers". A dash in the table means that the TV channel was not available at the victim's location.

user was silent (TrainStation and Cafe) or when he was coughing (Office). In the Lecture case the user could only be silent. We also avoided whistling in the cafe, because this may be awkward for some users. The FRR was 0.005 (3 over 579 logins) when the user was silent, 0.002 (1 over 529 logins) when the user was coughing, 0 (0 over 541 logins) when the user was speaking, and 0 (0 over 353 logins) when the user was whistling.

Phone Position. Figure 6(b) shows the similarity scores for different phone positions. Sound-Proof performs slightly better when the phone is on a table or on a bench. Worse performance when the phone is in a pocket or in a purse are likely due to the attenuation caused by the fabric around the microphone. The FRR was 0.001 (1 over 667 logins) with the phone on a table, 0.001 (1 over 675 logins) with the phone in a pocket, and 0.003 (2 over 660 logins) with the phone in a purse.

Phone Model. Figure 6(c) shows the similarity scores for the two phones. The Nexus 4 and the iPhone 5 performed equally good across all environments. The FRR was 0.002 (2 over 884 logins) with the iPhone 5 and 0.002 (2 over 1118 logins) with the Nexus 4.

Computer. Figure 6(d) shows the similarity scores for the two computers we used. We could not find significant differences between their performance. The FRR was 0.002 (3 over 1299 logins) with the MacBook Pro and 0.001 (1 over 703 logins) with the Dell.

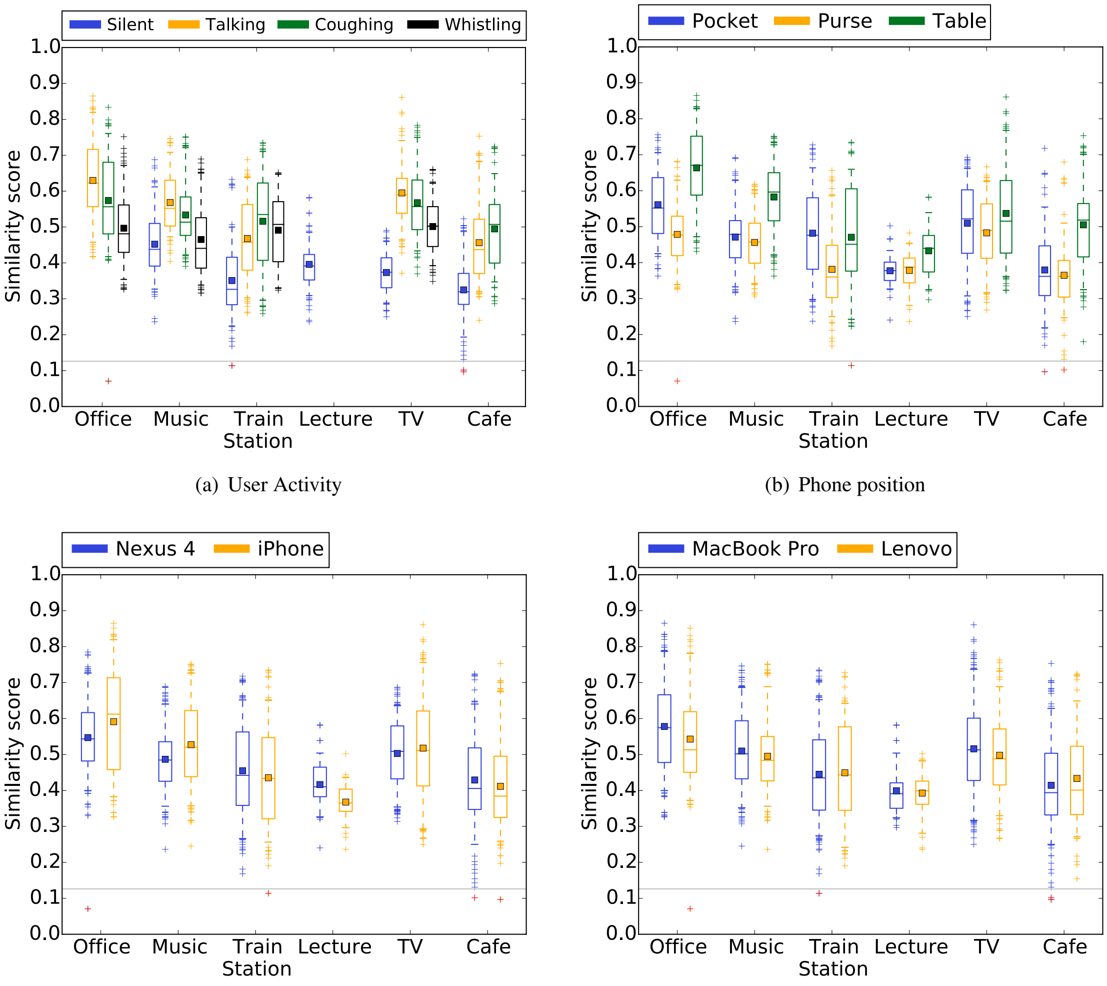
Distance Between Phone and Computer. In some settings (e.g., at home), the user's phone may be away from his computer. For instance, the user could leave the phone in his bedroom while watching TV or working in another room. We evaluated this scenario by placing the computer close to the TV in a living-room, and testing Sound-Proof while the phone was away from the computer. For this set of experiments we used the iPhone 5 and the MacBook Pro. The average noise level by the TV was measured at 50dB. We tested 3 different distances: 4, 8 and 12 meters (running 20 login attempts for each distance). All login attempts were successful (i.e., FRR=0). We also tried to log in while the phone was in another room behind a closed door, but logins were rejected.

Discussion. Based on the above results, we argue that the FRR of Sound-Proof is small enough to be practical for real-world usage. To put it in perspective, the FRR of Sound-Proof is likely to be smaller than the FRR due to mistyped passwords (0.04, as reported in [30]).

7.3 Advanced Attack Scenarios

A successful attack requires the adversary to submit a sample that is very similar to the one recoded by the victim's phone. For example, if the victim is in a cafe, the adversary should submit an audio sample that features typical sounds of that environment. In the following we assume a strong adversary that correctly guesses the victim's environment. We also evaluate the attack success rate in scenarios where the victim and the attacker access the same broadcast audio source from different locations.

Similar Environment Attack. In this experiment we assume that the victim and the adversary are located in similar environments. For each environment, we compute the FAR between each phone sample collected by one subject (the victim) and all the computer samples of the other subject (the adversary). We then switch the roles of the two subjects and repeat the procedure. The FAR for the Music and the TV environments were 0.012 (1063 over 91960 attempts) and 0.003 (311 over 90992 attempts), respectively. The FAR for the Lecture environment was 0.001 (8 over 7242 attempts). When both the victim and the attacker were located at a train station



(a) User Activity

(b) Phone position

(c) Phone model

(d) Computer

Figure 6: Impact of user activity, phone position, phone model, and computer model on the False Rejection Rate.

the FAR was 0.001 (44 over 67098 attempts). The FAR for the Office environment was 0.025 (1194 over 47250 attempts). When both the victim and the attacker were in a cafe the FAR was 0.001 (32 over 56994 attempts).

The above results show low FAR even when the attacker correctly guesses the victim's environment. This is due to the fact that ambient noise in a given environment is influenced by random events (e.g., background chatter, music, cups clinking, etc.) that cannot be controlled or predicted by the adversary.

Same Media Attack. In this experiment we assume that the victim and the adversary access the same audio source from different locations. This happens, for example, if the victim is watching TV and the adversary correctly guesses the channel to which the victim's TV is tuned. We place the victim's phone and the adversary's computer in different locations, but each of them next to a smart TV that was also capable of streaming web media. Since the devices have access to two identical audio sources, the adversary succeeds if the lag between the two audio signals is less than ℓ_{max}. We tested 4 cable TV channels, 2 web radios and 2 web TVs. For each scenario, we run the attack 100 times and report the FAR in Table 2. When the victim and the attacker were in the same city, we experienced differences based on the media provider. When the TVs reproduced content broadcasted by the same provider, the signals were closely synchronized and the similarity score was above the threshold τ_C. The FAR dropped in the case of web content. When the TVs reproduced content supplied by different providers, the lag between the signals caused the similarity score to drop below τ_C in most of the cases. The similarity score sensibly dropped when the victim and the attacker were located in different cities.

8 User Study

The goal of our user study was to evaluate the usability of Sound-Proof and to compare it with the usability of Google 2-Step Verification (2SV), since 2FA based on verification codes is arguably the most popular. (We only considered the version of Google 2SV that uses an application on the user's phone to generate verification codes.) We stress that the comparison focuses solely on the usability aspect of the two methods. In particular, we did not make the participants aware of the difference in the security guarantees, i.e., the fact that Google 2SV can better resist co-located attacks.

We ran repeated-measure experiments where each participant was asked to log in to a server using both mechanisms in random order. After using each 2FA mechanism, participants ranked its usability answering the System Usability Scale (SUS) [11]. The SUS is a widely-used scale to assess the usability of IT systems [9]. The SUS score ranges from 0 to 100, where higher scores indicate better usability.

8.1 Procedure

Recruitment. We recruited participants using a snowball sampling method. Most subjects were recruited outside our department and were not working in or studying computer science. The study was advertised as a user study to "evaluate the usability of two-factor authentication mechanisms". We informed participants that we would not collect any personal information and offered a compensation of CHF 20. Among all respondents to our email, we discarded the ones that were security experts and ended up with 32 participants.

Experiment. The experiment took place in our lab where we provided a laptop and a phone to complete the login procedures. Both devices were connected to the Internet through WiFi. We set up a Gmail account with Google 2SV enabled. We also created another website that supported Sound-Proof and mimicked the Gmail UI.

Participants saw a video where we explained the two mechanisms under evaluation. We told participants that they would need to log in using the account credentials and the hardware we provided. We also explained that we would record the keystrokes and the mouse movements (this allowed us to time the login attempts).

We then asked participants to fill in a pre-test questionnaire designed to collect demographic information. Participants logged in to our server using Sound-Proof and to Gmail using Google 2SV. We randomized the order in which each participant used the two mechanisms. After each login, participants rated the 2FA mechanism answering the SUS.

At the end of the experiment participants filled in a post-test questionnaire that covered aspects of the 2FA mechanisms under evaluation not covered by the SUS.

8.2 Results

Demographics. 58% of the participants were between 21 and 30 years old. 25% of the participants were between 31 and 40 years old. The remaining 17% of the participants were above 40 years old. 53% of the participants were female. 69% of the participants had a master or doctoral degree. 50% of the participants used 2FA for online banking and only 13% used Google 2SV to access their email accounts.

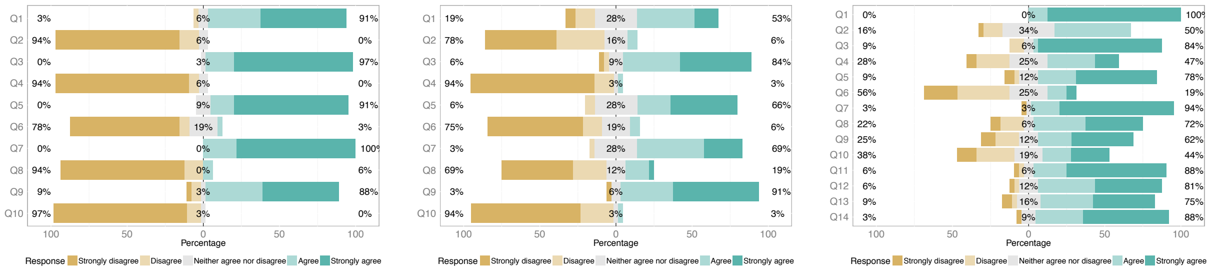
SUS Scores. The mean SUS score for Sound-Proof was 91.09 (±5.44). The mean SUS score for Google 2SV was 79.45 (±7.56). Figure 7(a) and Figure 7(b) show participant answers on 5-point Likert-scales for Sound-Proof and for Google 2SV, respectively. To analyze the statistical significance of these results, we used the following null hypothesis: "there will be no difference in perceived usability between Sound-Proof and Google 2SV". A one-way ANOVA test revealed that the difference of the SUS scores was statistically significant ($F(1,31) = 21.698,\ p < .001,\ \eta_p^2 = .412$), thus the null hypothesis can be rejected. We concluded that users perceive Sound-Proof to be more usable than Google 2SV. Appendix A reports the items of the SUS.

Login Time. We measured the login time from the moment when a participant clicked on the "login" button (right after entering the password), to the moment when that participant was logged in. We neglected the time spent entering username and password because we wanted to focus only on the time required by the 2FA mechanism. Login time for Sound-Proof was 4.7 seconds (±0.2 seconds); this time was required for the phone to receive the computer's sample and compare it with the one recorded locally. With Google 2SV, login time increased to 24.4 seconds (±7.1 seconds); this time was required for the participant to take the phone, start the application and copy the verification code from the phone to the browser.

Failure Rates. We did not witness any login failure for either of the two methods. We speculate that this may be due to the priming of the users right before the experiment, when we explained how the two methods work and that Sound-Proof may require users to make some noise in quiet environments.

Post-test Questionnaire. The post-test questionnaire was designed to collect information on the perceived quickness of the two mechanisms (Q1–Q2) and participants willingness to adopt any of the schemes (Q3–Q6). We also included items to inquire if participants would feel comfortable using the mechanisms in different environments (Q7–Q14). Figure 7(c) shows participants answers on 5-point Likert-scales. The full text of the items can be found in Appendix B.

All participants found Sound-Proof to be quick (Q1), while only 50% of the participants found Google 2SV to



(a) SUS answers for Sound-Proof

(b) SUS answers for Google 2SV

(c) Answers to the Post-test questionnaire

Figure 7: Distribution of the answers by the participants of the user study. System Usability Scale (SUS) of Sound-Proof (a) and Google 2-Step Verification (b), as well as the Post-test questionnaire (c). Percentages on the left side include participants that answered "Strongly disagree" or "Disagree". Percentages in the middle account for participants that answered "Neither agree, nor disagree". Percentages on the right side include participants that answered "Agree" or "Strongly agree".

be quick (Q2). If 2FA were mandatory, 84% of the participants would use Sound-Proof (Q3) and 47% would use Google 2SV (Q4). In case 2FA were optional the percentage of participants willing to use the two mechanisms dropped to 78% for Sound-Proof (Q5) and to 19% for Google 2SV (Q6). Similar to [36, 12], our results for Google 2SV suggest that users are likely not to use 2FA if it is optional. With Sound-Proof, the difference in user acceptance between a mandatory and an optional scenario is only 6%.

We asked participants if they would feel comfortable using either mechanism at home, at their workplace, in a cafe, and in a library. 95% of the participants would feel comfortable using Sound-Proof at home (Q7) and 77% of the participants would use it at the workplace (Q8). 68% would use it in a cafe (Q9) and 50% would use it in a library (Q10). Most participants (between 91% and 82%) would feel comfortable using Google 2SV in any of the scenario we considered (Q11–Q14).

The results of the post-test questionnaire suggest that users may be willing to adopt Sound-Proof because it is quicker and causes less burden, compared to Google 2SV. In some public places, however, users may feel more comfortable using Google 2SV. In Section 9 we discuss how to integrate the two approaches.

The post-test questionnaire allowed participants to comment on the 2FA mechanisms evaluated. Most participants found Sound-Proof to be user-friendly and appreciated the lack of interaction with the phone. Appendix C lists some of the users' comments.

9 Discussion

Software and Hardware Requirements. Similar to any other 2FA based on software tokens, Sound-Proof requires an application on the user's phone. Sound-Proof, however, does not require additional software on the computer and seamlessly works with any HTML5-compliant browser that implements the WebRTC API. Chrome, Firefox and Opera, already support WebRTC and a version of Internet Explorer supporting WebRTC will soon be released [31]. Sound-Proof needs the phone to have a data connection. Moreover, both the phone and the computer where the browser is running must be equipped with a microphone. Microphones are ubiquitous in phones, tablets and laptops. If a computer such as a desktop machine does not have an embedded microphone, Sound-Proof requires an external microphone, like the one of a webcam.

Other Browsers. Section 7 evaluates Sound-Proof using Google Chrome. We have also tested Sound-Proof with Mozilla Firefox and Opera. Each browser may use different algorithms to process the recorded audio (e.g., filtering for noise reduction), before delivering it to the web application. The WebRTC specification does not yet define how the recorded audio should be processed, leaving the specifics of the implementation to the browser vendor. When we ran our tests, Opera behaved like Chrome. Firefox audio processing was slightly different and it affected the performance our prototype. In particular, the Equal Error Rate computed over the samples collected while using Firefox was 0.012. We speculate that a better Equal Error Rate can be achieved with any browser if the software token performs the same audio processing of the browser being used to log in.

Privacy. The noise in the user's environment may leak private information to a prying server. In our design, the audio recorded by the phone is never uploaded to the server. A malicious server can also access the computer's microphone while the user is visiting the server's webpage. This is already the case for a number of websites that require access to the microphone. For example, websites for language learning, Gmail (for video-chats or phone calls), live chat-support services, or any site that uses speech-recognition require access to the microphone and may record the ambient noise any time the user visits the provider's webpage. All browsers we tested ask the user for permission before allowing a website to use `getUserMedia`. Moreover, browsers show an alert when a website triggers recording from the microphone. Providers are likely not to abuse the recording capability, since their reputation would be affected, if users detect unsolicited recording.

Quiet Environments. Sound-Proof rejects a login attempt if the power of either sample is below τ_{dB}. In case the environment is too quiet, the website can prompt the user to make any noise (by, e.g., clearing his throat, knocking on the table, etc.).

Fallback to Code-based 2FA. Sound-Proof can be combined with 2FA mechanisms based on verification codes, like Google 2SV. For example, the webpage can employ Sound-Proof as the default 2FA mechanism, but give to the user the option to log in entering a verification code. This may be useful in cases where the environment is quiet and the user feels uncomfortable making noise. Login based on verification codes is also convenient when the phone has no data connectivity (e.g., when roaming).

Failed Login Attempts and Throttling. Sound-Proof deems a login attempt as fraudulent if the similarity score between the two samples is below the threshold τ_C or if the power of either sample is below τ_{dB}. In this case, the server may request the two devices to repeat the recording and comparison phase. After a pre-defined number of failed trials, the server can fall-back to a 2FA mechanism based on verification codes. The server can also throttle login attempts in order to prevent "brute-force" attacks and to protect the user's phone battery from draining.

Login Evidence. Since audio recording and comparison is transparent to the user, he has no means to detect an ongoing attack. To mitigate this, at each login attempt the phone may vibrate, light up, or display a message to notify the user that a login attempt is taking place. The Sound-Proof application may also keep a log of the login attempts. Such techniques can help to make the user aware of fraudulent login attempts. Nevertheless, we stress that the user does not have to attend to the phone during legitimate login attempts.

Continuous Authentication. Sound-Proof can also be used as a form of continuous authentication. The server can periodically trigger Sound-Proof, while the user is logged in and interacts with the website. If the recordings of the two devices do not match, the server can forcibly log the user out. Nevertheless, such use can have a more significant impact on the user's privacy, as well as affect the battery life of the user's phone.

Alternative Devices. Our 2FA mechanism uses the phone as a software token. Another option is to use a smartwatch and we plan to develop a Sound-Proof application for smartwatches based on Android Wear and Apple Watch. We speculate that smartwatches can further lower the false rejection rate because of the proximity of the computer and the smartwatch during logins.

Logins from the Phone. If a user tries to log in from the same device where the Sound-Proof application is running, the browser and the application will capture audio through the same microphone and, therefore, the login attempt will be accepted. This requires the mobile OS to allow access to the microphone by the browser and, at the same time, by the Sound-Proof application. If the mobile OS does not allow concurrent access to the microphone, Sound-Proof can fall back to code-based 2FA.

Comparative Analysis. We use the framework of Bonneau et al. [10] to compare Sound-Proof with Google 2-Step Verification (Google 2SV), with PhoneAuth [14], and with the 2FA protocol of [41] that uses WiFi to create a channel between the phone and the computer (referred to as FBD-WF-WF in [41]). The framework of Bonneau et al. considers 25 "benefits" that an authentication scheme should provide, categorized in terms of usability, deployability, and security. Table 3 shows the overall comparison. The evaluation of Google 2SV in Table 3 matches the one reported by [10], besides the fact that we consider Google 2SV to be non-proprietary. *Usability:* No scheme is scalable nor it is effortless for the user because they all require a password as the first authentication factor. They are all "Quasi-Nothing-to-Carry" because they leverage the user's phone. Sound-Proof and PhoneAuth are more efficient to use than Google 2SV because they do not require the user to interact with his phone. They are also more efficient to use than FBD-WF-WF, because the latter requires a non-negligible setup time every time the user logs in from a new computer. All mechanisms incur some errors if the user enters the wrong password (Infrequent-Errors). All mechanisms also require similar recovery procedures if the user loses his phone. *Deployability:* Sound-Proof, PhoneAuth, and FBD-WF-WF score better than Google 2SV in the category "Accessible" because the user is asked nothing but his password. The three schemes are also better than Google 2SV in terms of cost per user, assuming users already have a phone. None of the mechanisms is server-compatible. Sound-Proof and Google 2SV are the only browser-compatible mechanisms as they require no changes to current browsers or computers. Google 2SV is more mature, and all of them are non-proprietary. *Security:* The security provided by Sound-Proof, PhoneAuth, and FBD-WF-WF is similar to the one provided by Google 2SV. However, we rate Sound-Proof and PhoneAuth as not resilient to targeted impersonation, since a targeted, co-located attacker can launch the attack from the victim's environment. FBD-WF-WF uses a paired connection between the user's computer and phone, and can better resist such attacks.

	Usability								Deployability						Security										
Scheme	Memorywise-Effortless	Scalable-for-Users	Nothing-to-Carry	Physically Effortless	Easy-to-Learn	Efficient-to-Use	Infrequent-Errors	Easy-Recovery-from-Loss	Accessible	Negligible-Cost-per-User	Server-Compatible	Browser-Compatible	Mature	Non-Proprietary	Resilient-to-Physical-Observation	Resilient-to-Targeted-Impersonation	Resilient-to-Throttled-Guessing	Resilient-to-Unthrottled-Guessing	Resilient-to-Internal-Observation	Resilient-to-Leaks-from-Other-Verifiers	Resilient-to-Phishing	Resilient-to-Theft	No-Trusted-Third-Party	Requiring-Explicit-Consent	Unlinkable
Sound-Proof			S		Y	Y	S	S	Y	Y		Y		Y	S		Y	Y		Y	Y	Y	Y	Y	
Google 2SV			S		Y	S	S	S	S	S		Y	Y	Y	S	S	Y	Y		Y	Y	Y	Y	Y	
PhoneAuth			S		Y	Y	S	S	Y	Y				Y	S		Y	Y		Y	Y	Y	Y	Y	
FBD-WF-WF			S		Y	S	S	S	Y	Y				Y	S	S	Y	Y		Y	Y	Y	Y	Y	

Table 3: Comparison of Sound-Proof against Google 2-Step Verification (Google 2SV), PhoneAuth [14], and FBD-WF-WF [41], using the framework of Bonneau et al. [10]. We use 'Y' to denote that the benefit is provided and 'S' to denote that the benefit is somewhat provided.

10 Related Work

Section 3 discusses alternative approaches to 2FA. In the following we review related work that leverages audio to verify the proximity of two devices.

Halevi et al., [27] use ambient audio to detect the proximity of two devices to thwart relay attacks in NFC payment systems. They compute the cross-correlation between the audio recorded by the two devices and employ machine-learning techniques to tell whether the two samples were recorded at the same location or not. The authors claim perfect results (0 false acceptance and false rejection rate). They, however, assume the two devices to have the same hardware (the experiment campaign used two Nokia N97 phones). Furthermore, their setup allows a maximum distance of 30 centimeters between the two devices. Our application scenario (web authentication) requires a solution that works (*i*) with heterogeneous devices, (*ii*) indoors and outdoors, and (*iii*) irrespective of the phone's position (e.g., in the user's pocket or purse). As such, we propose a different function to compute the similarity of the two samples, which we empirically found to be more robust, than what proposed in [27], in our settings.

Truong et al., [46] investigate relay attacks in zero-interaction authentication systems and use techniques similar to the ones of [27]. They propose a framework that detects co-location of two devices comparing features from multiple sensors, including GPS, Bluetooth, WiFi and audio. The authors conclude that an audio-only solution is not robust to detect co-location (20% of false rejections) and advocate for the combination of multiple sensors. Furthermore, their technique requires the two devices to sense the environment for 10 seconds. This time budget may not be available for web authentication.

The authors of [40] use ambient audio to derive a pairwise cryptographic key between two co-located devices. They use an audio fingerprinting scheme similar to the one of [26] and leverage fuzzy commitment schemes to accommodate for the difference of the two recordings. Their scheme may, in principle, be used to verify proximity of two devices in a 2FA mechanism. However, the experiments of [40] reveal that the key derivation is hardly feasible in outdoor scenarios. Our scheme takes advantage of noisy environments and, therefore, can be used in outdoor scenarios like train stations.

11 Conclusion

We proposed Sound-Proof, a two-factor authentication mechanism that does not require the user to interact with his phone and that can already be used with major browsers. We have shown that Sound-Proof works even if the phone is in the user's pocket or purse, and that it fares well both indoors and outdoors. Participants of a user study rated Sound-Proof to be more usable than Google 2-Step Verification. More importantly, most participants would use Sound-Proof for online services in which 2FA is optional. Sound-Proof improves the usability and deployability of 2FA and, as such, can foster large-scale adoption.

Acknowledgments

We thank Kurt Heutschi for the valuable discussions and insights on audio processing. We also thank our shepherd Joseph Bonneau, as well as the anonymous reviewers who helped to improve this paper with their useful feedback and comments.

References

[1] APPLE. Accelerate framework reference. https://goo.gl/WtnC0k.

[2] APPLE. Apple Push Notification Service. https://goo.gl/t8UUMf.

[3] ARENTZ, W. A., AND BANDARA, U. Near ultrasonic directional data transfer for modern smartphones. In *13th International Conference on Pervasive and Ubiquitous Computing* (2011), UbiComp '11.

[4] ARM. ARM NEON. http://www.arm.com/products/processors/technologies/neon.php.

[5] AUTHY INC. Authy. https://www.authy.com.

[6] BACKES, M., CHEN, T., DÜRMUTH, M., LENSCH, H. P. A., AND WELK, M. Tempest in a teapot: Compromising reflections revisited. In *IEEE Symposium on Security and Privacy* (2009), SP '09.

[7] BACKES, M., DÜRMUTH, M., AND UNRUH, D. Compromising reflections-or-how to read LCD monitors around the corner. In *IEEE Symposium on Security and Privacy* (2008), SP '08.

[8] BALUJA, S., AND COVELL, M. Waveprint: Efficient wavelet-based audio fingerprinting. *Pattern Recognition 41*, 11 (2008), 3467–3480.

[9] BANGOR, A., KORTUM, P. T., AND MILLER, J. T. An empirical evaluation of the system usability scale. *International Journal of Human-Computer Interaction 24*, 6 (2008).

[10] BONNEAU, J., HERLEY, C., VAN OORSCHOT, P. C., AND STAJANO, F. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *IEEE Symposium on Security and Privacy* (2012), SP '12.

[11] BROOKE, J. SUS - A quick and dirty usability scale. *Usability evaluation in industry 189*, 194 (1996), 4–7.

[12] BUSINESSWIRE. Impermium study unearths consumer attitudes toward internet security. http://goo.gl/NsUCL7, 2013.

[13] CHANDRASEKHAR, V., SHARIFI, M., AND ROSS, D. A. Survey and evaluation of audio fingerprinting schemes for mobile query-by-example applications. In *12th International Society for Music Information Retrieval Conference* (2011), ISMIR '11.

[14] CZESKIS, A., DIETZ, M., KOHNO, T., WALLACH, D. S., AND BALFANZ, D. Strengthening user authentication through opportunistic cryptographic identity assertions. In *ACM Conference on Computer and Communications Security* (2012), CCS '12.

[15] DANIEL C. BURNETT AND ADAM BERGKVIST AND CULLEN JENNINGS AND ANANT NARAYANAN. Media Capture and Streams (W3C Working Draft). http://www.w3.org/TR/mediacapture-streams/.

[16] DUO SECURITY, INC. Duo Push. https://www.duosecurity.com/product/methods/duo-mobile.

[17] EMC INC. RSA SecurID. https://www.emc.com/security/rsa-securid.htm/.

[18] ENCAP SECURITY. Encap Security. https://www.encapsecurity.com/.

[19] FETTE, I., AND MELNIKOV, A. The WebSocket protocol (RFC 6455). http://tools.ietf.org/html/rfc6455, 2011.

[20] FIDO ALLIANCE. Fido U2F specifications. https://fidoalliance.org/specifications/.

[21] GOOGLE. Google Cloud Messaging for Android. https://developer.android.com/google/gcm/index.html.

[22] GOOGLE INC. Google 2-Step Verification. https://www.google.com/landing/2step/.

[23] GOOGLE INC. SlickLogin. http://www.slicklogin.com/.

[24] GOOGLE INC. WebRTC. http://www.webrtc.org/.

[25] GUNSON, N., MARSHALL, D., MORTON, H., AND JACK, M. A. User perceptions of security and usability of single-factor and two-factor authentication in automated telephone banking. *Computers & Security 30*, 4 (2011), 208–220.

[26] HAITSMA, J., KALKER, T., AND OOSTVEEN, J. An efficient database search strategy for audio fingerprinting. In *5th Workshop on Multimedia Signal Processing* (2002), MMSP '02.

[27] HALEVI, T., MA, D., SAXENA, N., AND XIANG, T. Secure proximity detection for NFC devices based on ambient sensor data. In *17th European Symposium on Research in Computer Security* (2012), ESORICS '12.

[28] HAZAS, M., AND WARD, A. A novel broadband ultrasonic location system. In *4th International Conference on Pervasive and Ubiquitous Computing* (2002), UbiComp.

[29] KARAPANOS, N., AND CAPKUN, S. On the effective prevention of TLS man-in-the-middle attacks in web applications. In *23rd USENIX Security Symposium* (2014), USENIX Sec '14.

[30] KUMAR, M., GARFINKEL, T., BONEH, D., AND WINOGRAD, T. Reducing shoulder-surfing by using gaze-based password entry. In *3rd Symposium on Usable Privacy and Security* (2007), SOUPS '07.

[31] MICROSOFT. Bringing interoperable real-time communications to the web. http://blogs.skype.com/2014/10/27/bringing-interoperable-real-time-communications-to-the-web/.

[32] MOZZILLA. Location-Aware Browsing. https://www.mozilla.org/en-US/firefox/geolocation/.

[33] NETWORK TIME FOUNDATION. NTP: The Network Time Protocol. http://www.ntp.org/.

[34] OWASP. Man-in-the-browser attack. https://www.owasp.org/index.php/Man-in-the-browser_attack.

[35] PARNO, B., KUO, C., AND PERRIG, A. Phoolproof phishing prevention. In *10th International Conference on Financial Cryptography and Data Security* (2006), FC '06.

[36] PETSAS, T., TSIRANTONAKIS, G., ATHANASOPOULOS, E., AND IOANNIDIS, S. Two-factor authentication: Is the world ready?: Quantifying 2FA adoption. In *8th European Workshop on System Security* (2015), EuroSec '15.

[37] RAGURAM, R., WHITE, A. M., GOSWAMI, D., MONROSE, F., AND FRAHM, J. iSpy: Automatic reconstruction of typed input from compromising reflections. In *ACM Conference on Computer and Communications Security* (2011), CCS '11.

[38] RODRGUEZ VALIENTE, A., TRINIDAD, A., GARCA BERROCAL, J. R., GRRIZ, C., AND RAMREZ CAMACHO, R. Extended high-frequency (920 khz) audiometry reference thresholds in 645 healthy subjects. *International Journal of Audiology 53*, 8 (2014), 531–545.

[39] RUSSELL, D. A., TITLOW, J. P., AND BEMMEN, Y.-J. Acoustic monopoles, dipoles, and quadrupoles: An experiment revisited. *American Journal of Physics 67*, 8 (1999), 660–664.

[40] SCHÜRMANN, D., AND SIGG, S. Secure communication based on ambient audio. *IEEE Trans. Mob. Comput. 12*, 2 (2013), 358–370.

[41] SHIRVANIAN, M., JARECKI, S., SAXENA, N., AND NATHAN, N. Two-factor authentication resilient to server compromise using mix-bandwidth devices. In *The Network and Distributed System Security Symposium* (2014), NDSS '14.

[42] Shrestha, B., Saxena, N., Truong, H., and Asokan, N. Drone to the rescue: Relay-resilient authentication using ambient multi-sensing. In *Financial Cryptography and Data Security* (2014), FC '14.

[43] StatCounter. StatCounter global stats. http://gs.statcounter.com/.

[44] The American National Standards Insitute. ANSI s1.11-2004 - Specification for octave-band and fractional-octave-band analog and digital filters, 2004.

[45] The CherryPy team. CherryPy. http://www.cherrypy.org/.

[46] Truong, H. T. T., Gao, X., Shrestha, B., Saxena, N., Asokan, N., and Nurmi, P. Comparing and fusing different sensor modalities for relay attack resistance in zero-interaction authentication. In *International Conference on Pervasive Computing and Communications* (2014), PerCom '14.

[47] Vér, I., and Beranek, L. *Noise and Vibration Control Engineering*. Wiley, 2005.

[48] Wang, A. The shazam music recognition service. *Commun. ACM 49*, 8 (2006), 44–48.

[49] Web Bluetooth Community Group. Web Bluetooth. https://webbluetoothcg.github.io/web-bluetooth/.

[50] Weir, C. S., Douglas, G., Carruthers, M., and Jack, M. A. User perceptions of security, convenience and usability for ebanking authentication tokens. *Computers & Security 28*, 1-2 (2009), 47–62.

[51] Weir, C. S., Douglas, G., Richardson, T., and Jack, M. A. Usable security: User preferences for authentication methods in ebanking and the effects of experience. *Interacting with Computers 22*, 3 (2010), 153–164.

[52] Wireless Cables Inc. AIRCable. https://www.aircable.net/extend.php.

[53] Yubico. Yubikey hardware. https://www.yubico.com/.

Appendix

A System Usability Scale

We report the items of the System Usability Scale [11]. All items were answered with a 5-point Likert-scale from *Strongly Disagree* to *Strongly Agree.*

Q1 I think that I would like to use this system frequently.
Q2 I found the system unnecessarily complex.
Q3 I thought the system was easy to use.
Q4 I think that I would need the support of a technical person to be able to use this system.
Q5 I found the various functions in this system were well integrated.
Q6 I thought there was too much inconsistency in this system.
Q7 I would imagine that most people would learn to use this system very quickly.
Q8 I found the system very cumbersome to use.
Q9 I felt very confident using the system.
Q10 I needed to learn a lot of things before I could get going with this system.

B Post-test Questionnaire

We report the items of the post-test questionnaire. All items were answered with a 5-point Likert-scale from *Strongly Disagree* to *Strongly Agree.*

Q1 I thought the audio-based method was quick.
Q2 I thought the code-based method was quick.
Q3 If Second-Factor Authentication were mandatory, I would use the audio-based method to log in.
Q4 If Second-Factor Authentication were mandatory, I would use the code-based method to log in.
Q5 If Second-Factor Authentication were optional, I would use the audio-based method to log in.
Q6 If Second-Factor Authentication were optional, I would use the code-based method to log in.
Q7 I would feel comfortable using the audio-based method at home.
Q8 I would feel comfortable using the audio-based method at my workplace.
Q9 I would feel comfortable using the audio-based method in a cafe.
Q10 I would feel comfortable using the audio-based method in a library.
Q11 I would feel comfortable using the code-based method at home.
Q12 I would feel comfortable using the code-based method at my workplace.
Q13 I would feel comfortable using the code-based method in a cafe.
Q14 I would feel comfortable using the code-based method in a library.

C User Comments

This section lists some of the comments that participants added to their post-test questionnaire.

> "Sound-Proof is faster and automatic. Increased security without having to do more things"

> "I would use Sound-Proof, because it is less complicated and faster. I do not need to unlock the phone and open the application. In a public place it would feel a bit awkward unless it becomes widespread. Anyway, I am already logged in most websites that I use."

> "I like the audio idea, because what I hate the most about second-factor authentication is to have to take my phone out or find it around."

> "Sound-Proof is much easier. I am security-conscious and already use 2FA. I would be willing to switch to the audio-based method."

> "I already use Google 2SV and prefer it because I think it's more secure. However, Sound-Proof is seamless."

Android Permissions Remystified: A Field Study on Contextual Integrity

Primal Wijesekera[1], Arjun Baokar[2], Ashkan Hosseini[2], Serge Egelman[2],
David Wagner[2], and Konstantin Beznosov[1]
[1]*University of British Columbia, Vancouver, Canada,*
{primal,beznosov}@ece.ubc.ca
[2]*University of California, Berkeley, Berkeley, USA,*
{arjunbaokar,ashkan}@berkeley.edu, {egelman,daw}@cs.berkeley.edu

Abstract

We instrumented the Android platform to collect data regarding how often and under what circumstances smartphone applications access protected resources regulated by permissions. We performed a 36-person field study to explore the notion of "contextual integrity," i.e., how often applications access protected resources when users are not expecting it. Based on our collection of 27M data points and exit interviews with participants, we examine the situations in which users would like the ability to deny applications access to protected resources. At least 80% of our participants would have preferred to prevent at least one permission request, and overall, they stated a desire to block over a third of all requests. Our findings pave the way for future systems to automatically determine the situations in which users would want to be confronted with security decisions.

1 Introduction

Mobile platform permission models regulate how applications access certain resources, such as users' personal information or sensor data (e.g., camera, GPS, etc.). For instance, previous versions of Android prompt the user during application installation with a list of all the permissions that the application may use in the future; if the user is uncomfortable granting any of these requests, her only option is to discontinue installation [3]. On iOS and Android M, the user is prompted at runtime the first time an application requests any of a handful of data types, such as location, address book contacts, or photos [34].

Research has shown that few people read the Android install-time permission requests and even fewer comprehend them [16]. Another problem is habituation: on average, Android applications present the user with four permission requests during the installation process [13]. While iOS users are likely to see fewer permission requests than Android users, because there are fewer possible permissions and they are only displayed the first time the data is actually requested, it is not clear whether or not users are being prompted about access to data that they actually find concerning, or whether they would approve of subsequent requests [15].

Nissenbaum posited that the reason why most privacy models fail to predict violations is that they fail to consider contextual integrity [32]. That is, privacy violations occur when personal information is used in ways that defy users' expectations. We believe that this notion of "privacy as contextual integrity" can be applied to smartphone permission systems to yield more effective permissions by only prompting users when an application's access to sensitive data is likely to defy expectations. As a first step down this path, we examined how applications are currently accessing this data and then examined whether or not it complied with users' expectations.

We modified Android to log whenever an application accessed a permission-protected resource and then gave these modified smartphones to 36 participants who used them as their primary phones for one week. The purpose of this was to perform dynamic analysis to determine how often various applications are actually accessing protected resources under realistic circumstances. Afterwards, subjects returned the phones to our laboratory and completed exit surveys. We showed them various instances over the past week where applications had accessed certain types of data and asked whether those instances were expected, and whether they would have wanted to deny access. Participants wanted to block a third of the requests. Their decisions were governed primarily by two factors: whether they had privacy concerns surrounding the specific data type and whether they understood why the application needed it.

We contribute the following:

- To our knowledge, we performed the first field study to quantify the permission usage by third-party applications under realistic circumstances.

- We show that our participants wanted to block access to protected resources a third of the time. This suggests that some requests should be granted by runtime consent dialogs, rather than Android's previous all-or-nothing install-time approval approach.
- We show that the visibility of the requesting application and the frequency at which requests occur are two important factors which need to be taken into account in designing a runtime consent platform.

2 Related Work

While users are required to approve Android application permission requests during installation, most do not pay attention and fewer comprehend these requests [16, 26]. In fact, even developers are not fully knowledgeable about permissions [40], and are given a lot of freedom when posting an application to the Google Play Store [7]. Applications often do not follow the principle of least privilege, intentionally or unintentionally [44]. Other work has suggested improving the Android permission model with better definitions and hierarchical breakdowns [8]. Some researchers have experimented with adding fine-grained access control to the Android model [11]. Providing users with more privacy information and personal examples has been shown to help users in choosing applications with fewer permissions [21,27].

Previous work has examined the overuse of permissions by applications [13, 20], and attempted to identify malicious applications through their permission requests [36] or through natural language processing of application descriptions [35]. Researchers have also developed static analysis tools to analyze Android permission specifications [6, 9, 13]. Our work complements this static analysis by applying dynamic analysis to permission usage. Other researchers have applied dynamic analysis to native (non-Java) APIs among third-party mobile markets [39]; we apply it to the Java APIs available to developers in the Google Play Store.

Researchers examined user privacy expectations surrounding application permissions, and found that users were often surprised by the abilities of background applications to collect data [25, 42]. Their level of concern varied from annoyance to seeking retribution when presented with possible risks associated with permissions [15]. Some studies employed crowdsourcing to create a privacy model based on user expectations [30].

Researchers have designed systems to track or reduce privacy violations by recommending applications based on users' security concerns [2, 12, 19, 24, 28, 46–48]. Other tools dynamically block runtime permission requests [37]. Enck et al. found that a considerable number of applications transmitted location or other user data to third parties without requiring user consent [12]. Hornyack et al.'s AppFence system gave users the ability to deny data to applications or substitute fake data [24]. However, this broke application functionality for one-third of the applications tested.

Reducing the number of security decisions a user must make is likely to decrease habituation, and therefore, it is critical to identify *which* security decisions users should be asked to make. Based on this theory, Felt et al. created a decision tree to aid platform designers in determining the most appropriate permission-granting mechanism for a given resource (e.g., access to benign resources should be granted automatically, whereas access to dangerous resources should require approval) [14]. They concluded that the majority of Android permissions can be automatically granted, but 16% (corresponding to the 12 permissions in Table 1) should be granted via runtime dialogs.

Nissenbaum's theory of contextual integrity can help us to analyze "the appropriateness of a flow" in the context of permissions granted to Android applications [32]. There is ambiguity in defining when an application actually needs access to user data to run properly. It is quite easy to see why a location-sharing application would need access to GPS data, whereas that same request coming from a game like Angry Birds is less obvious. "Contextual integrity is preserved if information flows according to contextual norms" [32], however, the lack of thorough documentation on the Android permission model makes it easier for programmers to neglect these norms, whether intentionally or accidentally [38]. Deciding on whether an application is violating users' privacy can be quite complicated since "the scope of privacy is wide-ranging" [32]. To that end, we performed dynamic analysis to measure how often (and under what circumstances) applications were accessing protected resources, whether this complied with users' expectations, as well as how often they might be prompted if we adopt Felt et al.'s proposal to require runtime user confirmation before accessing a subset of these resources [14]. Finally, we show how it is possible to develop a classifier to automatically determine whether or not to prompt the user based on varying contextual factors.

3 Methodology

Our long-term research goal is to minimize habituation by only confronting users with *necessary* security decisions and avoiding showing them permission requests that are either expected, reversible, or unconcerning. Selecting which permissions to ask about requires understanding how often users would be confronted with each type of request (to assess the risk of habituation) and user reactions to these requests (to assess the benefit to users). In this study, we explored the problem space in two parts:

we instrumented Android so that we could collect actual usage data to understand how often access to various protected resources is requested by applications in practice, and then we surveyed our participants to understand the requests that they would not have granted, if given the option. This field study involved 36 participants over the course of one week of normal smartphone usage. In this section, we describe the log data that we collected, our recruitment procedure, and then our exit survey.

3.1 Tracking Access to Sensitive Data

In Android, when applications attempt to access protected resources (e.g., personal information, sensor data, etc.) at runtime, the operating system checks to see whether or not the requesting application was previously granted access during installation. We modified the Android platform to add a logging framework so that we could determine every time one of these resources was accessed by an application at runtime. Because our target device was a Samsung Nexus S smartphone, we modified Android 4.1.1 (Jellybean), which was the newest version of Android supported by our hardware.

3.1.1 Data Collection Architecture

Our goal was to collect as much data as possible about each applications' access to protected resources, while minimizing our impact on system performance. Our data collection framework consisted of two main components: a series of "producers" that hooked various Android API calls and a "consumer" embedded in the main Android framework service that wrote the data to a log file and uploaded it to our collection server.

We logged three kinds of permission requests. First, we logged function calls checked by `checkPermission()` in the Android `Context` implementation. Instrumenting the `Context` implementation, instead of the `ActivityManagerService` or `PackageManager`, allowed us to also log the function name invoked by the user-space application. Next, we logged access to the `ContentProvider` class, which verifies the read and write permissions of an application prior to it accessing structured data (e.g., contacts or calendars) [5]. Finally, we tracked permission checks during `Intent` transmission by instrumenting the `ActivityManagerService` and `BroadcastQueue`. `Intents` allow an application to pass messages to another application when an activity is to be performed in that other application (e.g., opening a URL in the web browser) [4].

We created a component called `Producer` that fetches the data from the above instrumented points and sends it back to the `Consumer`, which is responsible for logging everything reported. `Producers` are scattered across the Android Platform, since permission checks occur in multiple places. The `Producer` that logged the most data was in `system_server` and recorded direct function calls to Android's Java API. For a majority of privileged function calls, when a user application invokes the function, it sends the request to `system_server` via `Binder`. `Binder` is the most prominent IPC mechanism implemented to communicate with the Android Platform (whereas `Intents` communicate between applications). For requests that do not make IPC calls to the `system_server`, a `Producer` is placed in the user application context (e.g., in the case of `ContentProviders`).

The `Consumer` class is responsible for logging data produced by each `Producer`. Additionally, the `Consumer` also stores contextual information, which we describe in Section 3.1.2. The `Consumer` syncs data with the filesystem periodically to minimize impact on system performance. All log data is written to the internal storage of the device because the Android kernel is not allowed to write to external storage for security reasons. Although this protects our data from curious or careless users, it also limits our storage capacity. Thus, we compressed the log files once every two hours and upload them to our collection servers whenever the phone had an active Internet connection (the average uploaded and zipped log file was around 108KB and contained 9,000 events).

Due to the high volume of permission checks we encountered and our goal of keeping system performance at acceptable levels, we added rate-limiting logic to the `Consumer`. Specifically, if it has logged permission checks for a particular application/permission combination more than 10,000 times, it examines whether it did so while exceeding an average rate of 1 permission check every 2 seconds. If so, the `Consumer` will only record 10% of all future requests for this application/permission combination. When this rate-limiting is enabled, the `Consumer` tracks these application/permission combinations and updates all the `Producers` so that they start dropping these log entries. Finally, the `Consumer` makes a note of whenever this occurs so that we can extrapolate the true number of permission checks that occurred.

3.1.2 Data Collection

We hooked the permission-checking APIs so that every time the system checked whether an application had been granted a particular permission, we logged the name of the permission, the name of the application, and the API method that resulted in the check. In addition to timestamps, we collected the following contextual data:

- **Visibility**—We categorized whether the requesting application was visible to the user, using four categories: running (a) as a service with no user interaction; (b) as a service, but with user interaction via notifications or sounds; (c) as a foreground process, but in the background due to multitasking; or (d) as a foreground process with direct user interaction.

- **Screen Status**—Whether the screen was on/off.
- **Connectivity**—The phone's WiFi connection state.
- **Location**—The user's last known coordinates. In order to preserve battery life, we collected cached location data, rather than directly querying the GPS.
- **View**—The UI elements in the requesting application that were exposed to the user at the time that a protected resource was accessed. Specifically, since the UI is built from an XML file, we recorded the name of the screen as defined in the DOM.
- **History**—A list of applications with which the user interacted prior to the requesting application.
- **Path**—When access to a `ContentProvider` object was requested, the path to the specific content.

Felt et al. proposed granting most Android permissions without *a priori* user approval and granting 12 permissions (Table 1) at runtime so that users have contextual information to infer why the data might be needed [14]. The idea is that, if the user is asked to grant a permission while using an application, she may have some understanding of why the application needs that permission based on what she was doing. We initially wanted to perform experience sampling by probabilistically questioning participants whenever any of these 12 permissions were checked [29]. Our goal was to survey participants about whether access to these resources was expected and whether it should proceed, but we were concerned that this would prime them to the security focus of our experiment, biasing their subsequent behaviors. Instead, we instrumented the phones to probabilistically take screenshots of what participants were doing when these 12 permissions were checked so that we could ask them about it during the exit survey. We used reservoir sampling to minimize storage and performance impacts, while also ensuring that the screenshots covered a broad set of applications and permissions [43].

Figure 1 shows a screenshot captured during the study along with its corresponding log entry. The user was playing the Solitaire game while Spotify requested a WiFi scan. Since this permission was of interest (Table 1), our instrumentation took a screenshot. Since Spotify was not the application the participant was interacting with, its visibility was set to *false*. The history shows that prior to Spotify calling `getScanResults()`, the user had viewed Solitaire, the call screen, the launcher, and the list of MMS conversations.

Permission Type	**Activity**
WRITE_SYNC_SETTINGS	Change application sync settings when the user is roaming
ACCESS_WIFI_STATE	View nearby SSIDs
INTERNET	Access Internet when roaming
NFC	Communicate via NFC
READ_HISTORY_BOOKMARKS	Read users' browser history
ACCESS_FINE_LOCATION	Read GPS location
ACCESS_COARSE_LOCATION	Read network-inferred location (i.e., cell tower and/or WiFi)
LOCATION_HARDWARE	Directly access GPS data
READ_CALL_LOG	Read call history
ADD_VOICEMAIL	Read call history
READ_SMS	Read sent/received/draft SMS
SEND_SMS	Send SMS

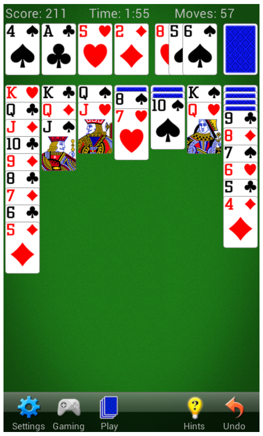
Table 1: The 12 permissions that Felt et al. recommend be granted via runtime dialogs [14]. We randomly took screenshots when these permissions were requested by applications, and we asked about them in our exit survey.

3.2 Recruitment

We placed an online recruitment advertisement on Craigslist in October of 2014, under the "et cetera jobs" section.[1] The title of the advertisement was "Research Study on Android Smartphones," and it stated that the study was about how people interact with their smartphones. We made no mention of security or privacy. Those interested in participating were directed to an online consent form. Upon agreeing to the consent form, potential participants were directed to a screening application in the Google Play store. The screening application asked for information about each potential participant's age, gender, smartphone make and model. It also collected data on their phones' internal memory size and the installed applications. We screened out applicants who were under 18 years of age or used providers other than T-Mobile, since our experimental phones could not attain 3G speeds on other providers. We collected data on participants' installed applications so that we could preinstall free applications prior to them visiting our laboratory. (We copied paid applications from their phones, since we could not download those ahead of time.)

We contacted participants who met our screening requirements to schedule a time to do the initial setup. Overall, 48 people showed up to our laboratory, and of those, 40 qualified (8 were rejected because our screening application did not distinguish some Metro PCS users

[1] Approved by the UC Berkeley IRB under protocol #2013-02-4992



(a) Screenshot

Name	Log Data
Type	API_FUNC
Permission	ACCESS_WIFI_STATE
App_Name	com.spotify.music
Timestamp	1412888326273
API Function	getScanResults()
Visibility	FALSE
Screen Status	SCREEN_ON
Connectivity	NOT_CONNECTED
Location	Lat 37.XXX Long -122.XXX - 1412538686641 (Time it was updated)
View	com.mobilityware.solitaire/.Solitaire
History	com.android.phone/.InCallScreen com.android.launcher/com.android.-launcher2.Launcher com.android.mms/ConversationList

(b) Corresponding log entry

Figure 1: Screenshot (a) and corresponding log entry (b) captured during the experiment.

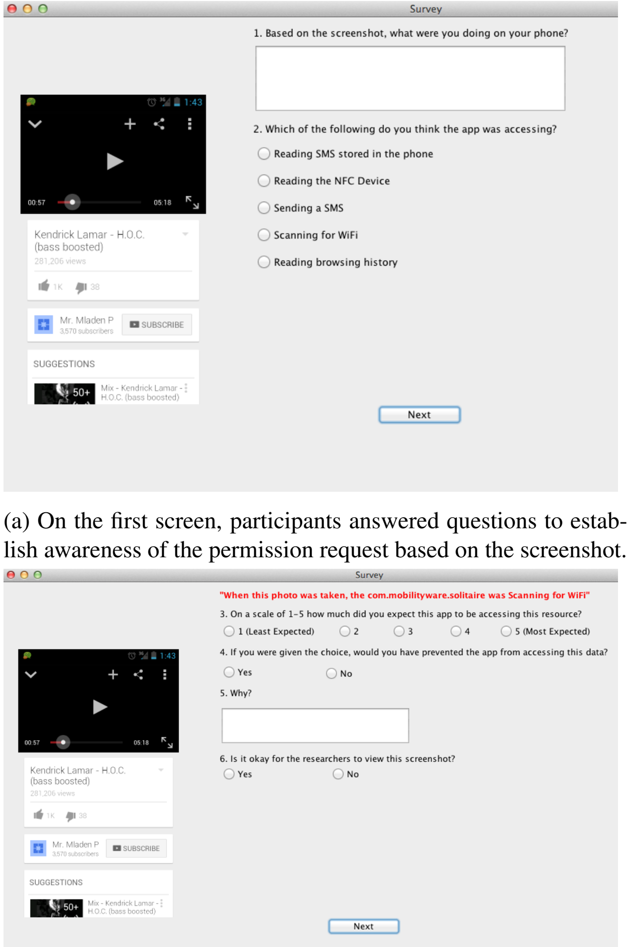
from T-Mobile users). In the email, we noted that due to the space constraints of our experimental phones, we might not be able to install all the applications on their existing phones, and therefore they needed to make a note of the ones that they planned to use that week. The initial setup took roughly 30 minutes and involved transferring their SIM cards, helping them set up their Google and other accounts, and making sure they had all the applications they needed. We compensated each participant with a $35 gift card for showing up at the setup session. Out of 40 people who were given phones, 2 did not return them, and 2 did not regularly use them during the study period. Of our 36 remaining participants who used the phones regularly, 19 were male and 17 were female; ages ranged from 20 to 63 years old (μ = 32, σ= 11).

After the initial setup session, participants used the experimental phones for one week in lieu of their normal phones. They were allowed to install and uninstall applications, and we instructed them to use these phones as they would their normal phones. Our logging framework kept track of every protected resource accessed by a user-level application along with the previously-mentioned contextual data. Due to storage constraints on the devices, our software uploaded log files to our server every two hours. However, to preserve participants' privacy, screenshots remained on the phones during the course of the week. At the end of the week, each participant returned to our laboratory, completed an exit survey, returned the phone, and then received an additional $100 gift card (i.e., slightly more than the value of the phone).

3.3 Exit Survey

When participants returned to our laboratory, they completed an exit survey. The exit survey software ran on a laptop in a private room so that it could ask questions about what they were doing on their phones during the course of the week without raising privacy concerns. We did not view their screenshots until participants gave us permission. The survey had three components:

- **Screenshots**—Our software displayed a screenshot taken after one of the 12 resources in Table 1 was accessed. Next to the screenshot (Figure 2a), we asked participants what they were doing on the phone when the screenshot was taken (open-ended). We also asked them to indicate which of several actions they believed the application was performing, chosen from a multiple-choice list of permissions presented in plain language (e.g., "reading browser history," "sending a SMS," etc.). After answering these questions, they proceeded to a second page of questions (Figure 2b). We informed participants at the top of this page of the resource that the application had accessed when the screenshot was taken, and asked them to indicate how much they expected this (5-point Likert scale). Next, we asked, "if you were given the choice, would you have prevented the app from accessing this data," and to explain why or why not. Finally, we asked for permission to view the screenshot. This phase of the exit survey was repeated for 10-15 different screenshots per participant, based on the number of screenshots saved by our reservoir sampling algorithm.
- **Locked Screens**—The second part of our survey involved questions about the same protected resources, though accessed while device screens were off (i.e., participants were not using their phones). Because there were no contextual cues (i.e., screenshots), we outright told participants which applications were accessing which resources and asked them multiple choice questions about whether they wanted to prevent this and the degree to which these



1. Based on the screenshot, what were you doing on your phone?

2. Which of the following do you think the app was accessing?

○ Reading SMS stored in the phone
○ Reading the NFC Device
○ Sending a SMS
○ Scanning for WiFi
○ Reading browsing history

(a) On the first screen, participants answered questions to establish awareness of the permission request based on the screenshot.

"When this photo was taken, the com.mobilityware.solitaire was Scanning for WiFi"

3. On a scale of 1-5 how much did you expect this app to be accessing this resource?

○ 1 (Least Expected) ○ 2 ○ 3 ○ 4 ○ 5 (Most Expected)

4. If you were given the choice, would you have prevented the app from accessing this data?

○ Yes ○ No

5. Why?

6. Is it okay for the researchers to view this screenshot?

○ Yes ○ No

(b) On the second screen, they saw the resource accessed, stated whether it was expected, and whether it should have been blocked.

Figure 2: Exit Survey Interface

behaviors were expected. They answered these questions for up to 10 requests, similarly chosen by our reservoir sampling algorithm to yield a breadth of application/permission combinations.

- **Personal Privacy Preferences**—Finally, in order to correlate survey responses with privacy preferences, participants completed two privacy scales. Because of the numerous reliability problems with the Westin index [45], we computed the average of both Buchanan et al.'s Privacy Concerns Scale (PCS) [10] and Malhotra et al.'s Internet Users' Information Privacy Concerns (IUIPC) scale [31].

After participants completed the exit survey, we re-entered the room, answered any remaining questions, and then assisted them in transferring their SIM cards back into their personal phones. Finally, we compensated each participant with a $100 gift card.

Three researchers independently coded 423 responses to the open-ended question in the screenshot portion of the survey. The number of responses per participant varied, as they were randomly selected based on the number of screenshots taken: participants who used their phones more heavily had more screenshots, and thus answered more questions. Prior to meeting to achieve consensus, the three coders disagreed on 42 responses, which resulted in an inter-rater agreement of 90%. Taking into account the 9 possible codings for each response, Fleiss' kappa yielded 0.61, indicating substantial agreement.

4 Application Behaviors

Over the week-long period, we logged 27M application requests to protected resources governed by Android permissions. This translates to over 100,000 requests per user/day. In this section, we quantify the circumstances under which these resources were accessed. We focus on the rate at which resources were accessed when participants were not actively using those applications (i.e., situations likely to defy users' expectations), access to certain resources with particularly high frequency, and the impact of replacing certain requests with runtime confirmation dialogs (as per Felt et al.'s suggestion [14]).

4.1 Invisible Permission Requests

In many cases, it is entirely expected that an application might make frequent requests to resources protected by permissions. For instance, the INTERNET permission is used every time an application needs to open a socket, ACCESS_FINE_LOCATION is used every time the user's location is checked by a mapping application, and so on. However, in these cases, one expects users to have certain contextual cues to help them understand that these applications are running and making these requests. Based on our log data, most requests occurred while participants were not actually interacting with those applications, nor did they have any cues to indicate that the applications were even running. When resources are accessed, applications can be in five different states, with regard to their visibility to users:

1. **Visible foreground application (12.04%)**: the user is using the application requesting the resource.
2. **Invisible background application (0.70%)**: due to multitasking, the application is in the background.
3. **Visible background service (12.86%)**: the application is a background service, but the user may be aware of its presence due to other cues (e.g., it is playing music or is present in the notification bar).
4. **Invisible background service (14.40%)**: the application is a background service without visibility.
5. **Screen off (60.00%)**: the application is running, but the phone screen is off because it is not in use.

Permission	Requests
ACCESS_NETWORK_STATE	31,206
WAKE_LOCK	23,816
ACCESS_FINE_LOCATION	5,652
GET_ACCOUNTS	3,411
ACCESS_WIFI_STATE	1,826
UPDATE_DEVICE_STATS	1,426
ACCESS_COARSE_LOCATION	1,277
AUTHENTICATE_ACCOUNTS	644
READ_SYNC_SETTINGS	426
INTERNET	416

Table 2: The most frequently requested permissions by applications with zero visibility to the user.

Application	Requests
Facebook	36,346
Google Location Reporting	31,747
Facebook Messenger	22,008
Taptu DJ	10,662
Google Maps	5,483
Google Gapps	4,472
Foursquare	3,527
Yahoo Weather	2,659
Devexpert Weather	2,567
Tile Game(Umoni)	2,239

Table 3: The applications making the most permission requests while running invisibly to the user.

Combining the 3.3M (12.04% of 27M) requests that were granted when the user was actively using the application (Category 1) with the 3.5M (12.86% of 27M) requests that were granted when the user had other contextual cues to indicate that the application was running (Category 3), we can see that fewer than one quarter of all permission requests (24.90% of 27M) occurred when the user had clear indications that those applications were running. This suggests that during the vast majority of the time, access to protected resources occurs opaquely to users. We focus on these 20.3M "invisible" requests (75.10% of 27M) in the remainder of this subsection.

Harbach et al. found that users' phone screens are off 94% of the time on average [22]. We observed that 60% of permission requests occurred while participants' phone screens were off, which suggests that permission requests occurred less frequently than when participants were using their phones. At the same time, certain applications made more requests when participants were not using their phones: "Brave Frontier Service," "Microsoft Sky Drive," and "Tile game by UMoni." Our study collected data on over 300 applications, and therefore it is possible that with a larger sample size, we would observe other applications engaging in this behavior. All of the aforementioned applications primarily requested ACCESS_WIFI_STATE and INTERNET. While a definitive explanation for this behavior requires examining source code or the call stacks of these applications, we hypothesize that they were continuously updating local data from remote servers. For instance, Sky Drive may have been updating documents, whereas the other two applications may have been checking the status of multiplayer games.

Table 2 shows the most frequently requested permissions from applications running invisibly to the user (i.e., Categories 2, 4, and 5); Table 3 shows the applications responsible for these requests (Appendix A lists the permissions requested by these applications). We normalized the numbers to show requests per user/day. ACCESS_NETWORK_STATE was most frequently requested, averaging 31,206 times per user/day—roughly once every 3 seconds. This is due to applications constantly checking for Internet connectivity. However, the 5,562 requests/day to ACCESS_FINE_LOCATION and 1,277 requests/day to ACCESS_COARSE_LOCATION are more concerning, as this could enable detailed tracking of the user's movement throughout the day. Similarly, a user's location can be inferred by using ACCESS_WIFI_STATE to get data on nearby WiFi SSIDs.

Contextual integrity means ensuring that information flows are appropriate, as determined by the user. Thus, users need the ability to see information flows. Current mobile platforms have done some work to let the user know about location tracking. For instance, recent versions of Android allow users to see which applications have used location data recently. While attribution is a positive step towards contextual integrity, attribution is most beneficial for actions that are reversible, whereas the disclosure of location information is not something that can be undone [14]. We observed that fewer than 1% of location requests were made when the applications were visible to the user or resulted in the displaying of a GPS notification icon. Given that Thompson et al. showed that most users do not understand that applications running in the background may have the same abilities as applications running in the foreground [42], it is likely that in the vast majority of cases, users do not know when their locations are being disclosed.

This low visibility rate is because Android only shows a notification icon when the GPS sensor is accessed, while offering alternative ways of inferring location. In 66.1% of applications' location requests, they directly queried the `TelephonyManager`, which can be used to determine location via cellular tower information. In 33.3% of the cases, applications requested the SSIDs of nearby WiFi networks. In the remaining 0.6% of cases, applications accessed location information using one of three built-in location providers: GPS, network, or passive. Applications accessed the GPS location provider only 6% of the time (which displayed a GPS notification). In the other 94% of the time, 13% queried the network provider (i.e., approximate location based on nearby cellular towers and WiFi SSIDs) and 81% queried the passive location provider. The passive location provider caches prior requests made to either the GPS or network providers. Thus, across all requests for location data, the GPS notification icon appeared 0.04% of the time.

While the alternatives to querying the GPS are less accurate, users are still surprised by their accuracy [17]. This suggests a serious violation of contextual integrity, since users likely have no idea their locations are being requested in the vast majority of cases. Thus, runtime notifications for location tracking need to be improved [18].

Apart from these invisible location requests, we also observed applications reading stored SMS messages (125 times per user/day), reading browser history (5 times per user/day), and accessing the camera (once per user/day). Though the use of these permissions does not necessarily lead to privacy violations, users have no contextual cues to understand that these requests are occurring.

4.2 High Frequency Requests

Some permission requests occurred so frequently that a few applications (i.e., Facebook, Facebook Messenger, Google Location Reporting, Google Maps, Farm Heroes Saga) had to be rate limited in our log files (see Section 3.1.1), so that the logs would not fill up users' remaining storage or incur performance overhead. Table 4 shows the complete list of application/permission combinations that exceeded the threshold. For instance, the most frequent requests came from Facebook requesting ACCESS_NETWORK_STATE with an average interval of 213.88 ms (i.e., almost 5 times per second).

With the exception of Google's applications, all rate-limited applications made excessive requests for the connectivity state. We hypothesize that once these applications lose connectivity, they continuously poll the system until it is regained. Their use of the `getActiveNetworkInfo()` method results in permission checks and returns `NetworkInfo` objects, which allow them to determine connection state (e.g., connected, disconnected, etc.) and type (e.g., WiFi, Bluetooth, cellular, etc.). Thus, these requests do not appear to be leaking sensitive information *per se*, but their frequency may have adverse effects on performance and battery life. It is possible that using the `ConnectivityManager`'s `NetworkCallback` method may be able to fulfill this need with far fewer permission checks.

Application / Permission	**Peak (ms)**	**Avg. (ms)**
com.facebook.katana ACCESS_NETWORK_STATE	213.88	956.97
com.facebook.orca ACCESS_NETWORK_STATE	334.78	1146.05
com.google.android.apps.maps ACCESS_NETWORK_STATE	247.89	624.61
com.google.process.gapps AUTHENTICATE_ACCOUNTS	315.31	315.31
com.google.process.gapps WAKE_LOCK	898.94	1400.20
com.google.process.location WAKE_LOCK	176.11	991.46
com.google.process.location ACCESS_FINE_LOCATION	1387.26	1387.26
com.google.process.location GET_ACCOUNTS	373.41	1878.88
com.google.process.location ACCESS_WIFI_STATE	1901.91	1901.91
com.king.farmheroessaga ACCESS_NETWORK_STATE	284.02	731.27
com.pandora.android ACCESS_NETWORK_STATE	541.37	541.37
com.taptu.streams ACCESS_NETWORK_STATE	1746.36	1746.36

Table 4: The application/permission combinations that needed to be rate limited during the study. The last two columns show the fastest interval recorded and the average of all the intervals recorded before rate-limiting.

4.3 Frequency of Data Exposure

Felt et al. posited that while most permissions can be granted automatically in order to not habituate users to relatively benign risks, certain requests should require runtime consent [14]. They advocated using runtime dialogs before the following actions should proceed:

1. Reading location information (e.g., using conventional location APIs, scanning WiFi SSIDs, etc.).
2. Reading the user's web browser history.
3. Reading saved SMS messages.
4. Sending SMS messages that incur charges, or inappropriately spamming the user's contact list.

These four actions are governed by the 12 Android permissions listed in Table 1. Of the 300 applications that we observed during the experiment, 91 (30.3%) performed one of these actions. On average, these permissions were requested 213 times per hour/user—roughly every 20 seconds. However, permission checks occur under a variety of circumstances, only a subset of which expose sensitive resources. As a result, platform develop-

Resource	**Visible**		**Invisible**		**Total**	
	Data Exposed	**Requests**	**Data Exposed**	**Requests**	**Data Exposed**	**Requests**
Location	758	2,205	3,881	8,755	4,639	10,960
Read SMS data	378	486	72	125	450	611
Sending SMS	7	7	1	1	8	8
Browser History	12	14	2	5	14	19
Total	1,155	2,712	3,956	8,886	5,111	11,598

Table 5: The sensitive permission requests (per user/day) when requesting applications were visible/invisible to users. "Data exposed" reflects the subset of permission-protected requests that resulted in sensitive data being accessed.

ers may decide to only show runtime warnings to users when protected data is read or modified. Thus, we attempted to quantify the frequency with which permission checks actually result in access to sensitive resources for each of these four categories. Table 5 shows the number of requests seen per user/day under each of these four categories, separating the instances in which sensitive data was exposed from the total permission requests observed. Unlike Section 4.1, we include "visible" permission requests (i.e., those occurring while the user was actively using the application or had other contextual information to indicate it was running). We didn't observe any uses of NFC, READ_CALL_LOG, ADD_VOICEMAIL, accessing WRITE_SYNC_SETTINGS or INTERNET while roaming in our dataset.

Of the location permission checks, a majority were due to requests for location provider information (e.g., `getBestProvider()` returns the best location provider based on application requirements), or checking WiFi state (e.g., `getWifiState()` only reveals whether WiFi is enabled). Only a portion of the requests actually exposed participants' locations (e.g., `getLastKnownLocation()` or `getScanResults()` exposed SSIDs of nearby WiFi networks).

Although a majority of requests for the READ_SMS permission exposed content in the SMS store (e.g., `Query()` reads the contents of the SMS store), a considerable portion simply read information about the SMS store (e.g., `renewMmsConnectivity()` resets an applications' connection to the MMS store). An exception to this is the use of SEND_SMS, which resulted in the transmission of an SMS message every time the permission was requested.

Regarding browser history, both accessing visited URLs (`getAllVisitedUrls()`) and reorganizing bookmark folders (`addFolderToCurrent()`) result in the same permission being checked. However, the latter does not expose specific URLs to the invoking application.

Our analysis of the API calls indicated that on average, only half of all permission checks granted applications access to sensitive data. For instance, across both visible and invisible requests, 5,111 of the 11,598 (44.3%) permission checks involving the 12 permissions in Table 1 resulted in the exposure of sensitive data (Table 5).

While limiting runtime permission requests to only the cases in which protected resources are exposed will greatly decrease the number of user interruptions, the frequency with which these requests occur is still too great. Prompting the user on the first request is also not appropriate (e.g., à la iOS and Android M), because our data show that in the vast majority of cases, the user has no contextual cues to understand when protected resources are being accessed. Thus, a user may grant a request the first time an application asks, because it is appropriate in that instance, but then she may be surprised to find that the application continues to access that resource in other contexts (e.g., when the application is not actively used). As a result, a more intelligent method is needed to determine when a given permission request is likely to be deemed appropriate by the user.

5 User Expectations and Reactions

To identify when users might want to be prompted about permission requests, our exit survey focused on participants' reactions to the 12 permissions in Table 1, limiting the number of requests shown to each participant based on our reservoir sampling algorithm, which was designed to ask participants about a diverse set of permission/application combinations. We collected participants' reactions to 673 permission requests ($\approx$19/participant). Of these, 423 included screenshots because participants were actively using their phones when the requests were made, whereas 250 permission requests were performed while device screens were off.[2] Of the former, 243 screenshots were taken while the requesting application was visible (Category 1 and 3 from Section 4.1), whereas 180 were taken while the application was invisible (Category 2 and 4 from Section 4.1). In this section, we describe the situations in which requests

[2] Our first 11 participants did not answer questions about permission requests occurring while not using their devices, and therefore the data only corresponds to our last 25 participants.

defied users' expectations. We present explanations for why participants wanted to block certain requests, the factors influencing those decisions, and how expectations changed when devices were not in use.

5.1 Reasons for Blocking

When viewing screenshots of what they were doing when an application requested a permission, 30 participants (80% of 36) stated that they would have preferred to block at least one request, whereas 6 stated a willingness to allow all requests, regardless of resource type or application. Across the entire study, participants wanted to block 35% of these 423 permission requests. When we asked participants to explain their rationales for these decisions, two main themes emerged: the request did not—in their minds—pertain to application functionality or it involved information they were uncomfortable sharing.

5.1.1 Relevance to Application Functionality

When prompted for the reason behind blocking a permission request, 19 (53% of 36) participants did not believe it was necessary for the application to perform its task. Of the 149 (35% of 423) requests that participants would have preferred to block, 79 (53%) were perceived as being irrelevant to the functionality of the application:

- *"It wasn't doing anything that needed my current location."* (P1)
- *"I don't understand why this app would do anything with SMS."* (P10)

Accordingly, functionality was the most common reason for wanting a permission request to proceed. Out of the 274 permissible requests, 195 (71% of 274) were perceived as necessary for the core functionality of the application, as noted by thirty-one (86% of 36) participants:

- *"Because it's a weather app and it needs to know where you are to give you weather information."*(P13)
- *"I think it needs to read the SMS to keep track of the chat conversation."*(P12)

Beyond being necessary for core functionality, participants wanted 10% (27 of 274) of requests to proceed because they offered convenience; 90% of these requests were for location data, and the majority of those applications were published under the Weather, Social, and Travel & Local categories in the Google Play store:

- *"It selects the closest stop to me so I don't have to scroll through the whole list."* (P0)
- *"This app should read my current location. I'd like for it to, so I won't have to manually enter in my zip code / area."* (P4)

Thus, requests were allowed when they were expected: when participants rated the extent to which each request was expected on a 5-point Likert scale, allowable requests averaged 3.2, whereas blocked requests averaged 2.3 (lower is less expected).

5.1.2 Privacy Concerns

Participants also wanted to deny permission requests that involved data that they considered sensitive, regardless of whether they believed the application actually needed the data to function. Nineteen (53% of 36) participants noted privacy as a concern while blocking a request, and of the 149 requests that participants wanted to block, 49 (32% of 149) requests were blocked for this reason:

- *"SMS messages are quite personal."* (P14)
- *"It is part of a personal conversation."* (P11)
- *"Pictures could be very private and I wouldn't like for anybody to have access."* (P16)

Conversely, 24 participants (66% of 36) wanted requests to proceed simply because they did not believe that the data involved was particularly sensitive; this reasoning accounted for 21% of the 274 allowable requests:

- *"I'm ok with my location being recorded, no concerns."* (P3)
- *"No personal info being shared."* (P29)

5.2 Influential Factors

Based on participants' responses to the 423 permission requests involving screenshots (i.e., requests occurring while they were actively using their phones), we quantitatively examined how various factors influenced their desire to block some of these requests.

Effects of Identifying Permissions on Blocking: In the exit survey, we asked participants to guess the permission an application was requesting, based on the screenshot of what they were doing at the time. The real answer was among four other incorrect answers. Of the 149 cases where participants wanted to block permission requests, they were only able to correctly state what permission was being requested 24% of the time; whereas when wanting a request to proceed, they correctly identified the requested permission 44% (120 of 274) of the time. However, Pearson's product-moment test on the average number of blocked requests per user and the average number of correct answers per user[3] did not yield a statistically significant correlation ($r = -0.171$, $p < 0.317$).

Effects of Visibility on Expectations: We were particularly interested in exploring if permission requests originating from foreground applications (i.e., visible to the

[3] Both measures were normally distributed.

user) were more expected than ones from background applications. Of the 243 visible permission requests that we asked about in our exit survey, participants correctly identified the requested permission 44% of the time, and their average rating on our expectation scale was 3.4. On the other hand, participants correctly identified the resources accessed by background applications only 29% of the time (52 of 180), and their average rating on our expectation scale was 3.0. A Wilcoxon Signed-Rank test with continuity correction revealed a statistically significant difference in participants' expectations between these two groups (V=441.5, p<0.001).

Effects of Visibility on Blocking: Participants wanted to block 71 (29% of 243) permission requests originating from applications running in the foreground, whereas this increased by almost 50% when the applications were in the background invisible to them (43% of 180). We calculated the percentage of denials for each participant, for both visible and invisible requests. A Wilcoxon Signed-Rank test with continuity correction revealed a statistically significant difference (V=58, p<0.001).

Effects of Privacy Preferences on Blocking: Participants completed the Privacy Concerns Scale (PCS) [10] and the Internet Users' Information Privacy Concerns (IUIPC) scale [31]. A Spearman's rank test yielded no statistically significant correlation between their privacy preferences and their desire to block permission requests ($\rho = 0.156$, p<0.364).

Effects of Expectations on Blocking: We examined whether participants' expectations surrounding requests correlated with their desire to block them. For each participant, we calculated their average Likert scores for their expectations and the percentage of requests that they wanted to block. Pearson's product-moment test showed a statistically significant correlation (r=-0.39, p<0.018). The negative correlation shows that participants were more likely to deny unexpected requests.

5.3 User Inactivity and Resource Access

In the second part of the exit survey, participants answered questions about 10 resource requests that occurred when the screen was off (not in use). Overall, they were more likely to expect resource requests to occur when using their devices ($\mu = 3.26$ versus $\mu = 2.66$). They also stated a willingness to block almost half of the permission requests (49.6% of 250) when not in use, compared to a third of the requests that occurred when using their phones (35.2% of 423). However, neither of these differences was statistically significant.

6 Feasibility of Runtime Requests

Felt et al. posited that certain sensitive permissions (Table 1) should require runtime consent [14], but in Section 4.3 we showed that the frequencies with which applications are requesting these permissions make it impractical to prompt the user each time a request occurs. Instead, the major mobile platforms have shifted towards a model of prompting the user the first time an application requests access to certain resources: iOS does this for a selected set of resources, such as location and contacts, and Android M does this for "dangerous" permissions.

How many prompts would users see, if we added runtime prompts for the first use of these 12 permissions? We analyzed a scheme where a runtime prompt is displayed at most once for each unique triplet of (application, permission, application visibility), assuming the screen is on. With a naïve scheme, our study data indicates our participants would have seen an average of 34 runtime prompts (ranging from 13 to 77, σ=11). As a refinement, we propose that the user should be prompted only if sensitive data will be exposed (Section 4.3), reducing the average number of prompts to 29.

Of these 29 prompts, 21 (72%) are related to location. Apple iOS already prompts users when an application accesses location for the first time, with no evidence of user habituation or annoyance. Focusing on the remaining prompts, we see that our policy would introduce an average of 8 new prompts per user: about 5 for reading SMS, 1 for sending SMS, and 2 for reading browser history. Our data covers only the first week of use, but as we only prompt on first use of a permission, we expect that the number of prompts would decline greatly in subsequent weeks, suggesting that this policy would likely not introduce significant risk of habituation or annoyance. Thus, our results suggest adding runtime prompts for reading SMS, sending SMS, and reading browser history would be useful given their sensitivity and low frequency.

Our data suggests that taking visibility into account is important. If we ignore visibility and prompted only once for each pair of (application, permission), users would have no way to select a different policy for when the application is visible or not visible. In contrast, "ask-on-first-use" for the triple (application, permission, visibility) gives users the option to vary their decision based on the visibility of the requesting application. We evaluated these two policies by analyzing the exit survey data (limited to situations where the screen was on) for cases where the same user was asked twice in the survey about situations with the same (application, permission) pair or the same (application, permission, visibility) triplet, to see whether the user's first decision to block or not matched their subsequent decisions. For the former policy, we saw only 51.3% agreement; for the latter, agreement increased to 83.5%. This suggests that the (application, permission, visibility) triplet captures many of the contextual factors that users care about, and thus it is reasonable to prompt only once per unique triplet.

A complicating factor is that applications can also run even when the user is not actively using the phone. In addition to the 29 prompts mentioned above, our data indicates applications would have triggered an average of 7 more prompts while the user was not actively using the phone: 6 for location and one for reading SMS. It is not clear how to handle prompts when the user is not available to respond to the prompt: attribution might be helpful, but further research is needed.

6.1 Modeling Users' Decisions

We constructed several statistical models to examine whether users' desire to block certain permission requests could be predicted using the contextual data that we collected. If such a relationship exists, a classifier could determine when to deny potentially unexpected permission requests without user intervention. Conversely, the classifier could be used to only prompt the user about questionable data requests. Thus, the response variable in our models is the user's choice of whether to block the given permission request. Our predictive variables consisted of the information that might be available at runtime: permission type (with the restriction that the invoked function exposes data), requesting application, and visibility of that application. We constructed several mixed effects binary logistic regression models to account for both inter-subject and intra-subject effects.

6.1.1 Model Selection

In our mixed effects models, permission types and the visibility of the requesting application were fixed effects, because all possible values for each variable existed in our data set. Visibility had two values: visible (the user is interacting with the application or has other contextual cues to know that it is running) and invisible. Permission types were categorized based on Table 5. The application name and the participant ID were included as random effects, because our survey data did not have an exhaustive list of all possible applications a user could run, and the participant has a non-systematic effect on the data.

Table 6 shows two goodness-of-fit metrics: the Akaike Information Criterion (AIC) and Bayesian Information Criterion (BIC). Lower values for AIC and BIC represent better fit. Table 6 shows the different parameters included in each model. We found no evidence of interaction effects and therefore did not include them. Visual inspection of residual plots of each model did not reveal obvious deviations from homoscedasticity or normality.

Predictors	**AIC**	**BIC**	**Screen State**
UserCode	490.60	498.69	Screen On
Application	545.98	554.07	Screen On
Application UserCode	491.86	503.99	Screen On
Permission Application UserCode	494.69	527.05	Screen On
Visibility **Application** **UserCode**	**481.65**	**497.83**	**Screen On**
Permission Visibility Application UserCode	484.23	520.64	Screen On
UserCode	245.13	252.25	Screen Off
Application	349.38	356.50	Screen Off
Application UserCode	238.84	249.52	Screen Off
Permission **Application** **UserCode**	**235.48**	**263.97**	**Screen Off**

Table 6: Goodness-of-fit metrics for various mixed effects logistic regression models on the exit survey data.

We initially included the phone's screen state as another variable. However, we found that creating two separate models based on the screen state resulted in better fit than using a single model that accounted for screen state as a fixed effect. When the screen was on, the best fit was a model including application visibility and application name, while controlling for subject effects. Here, fit improved once permission type was removed from the model, which shows that the decision to block a permission request was based on contextual factors: users do not categorically deny permission requests based solely on the type of resource being accessed (i.e., they also account for their trust in the application, as well as whether they happened to be actively using it). When the screen was off, however, the effect of permission type was relatively stronger. The strong subject effect in both models indicates that these decisions vary from one user to the next. As a result, any classifier developed to automatically decide whether to block a permission at runtime (or prompt the user) will need to be tailored to that particular user's needs.

6.1.2 Predicting User Reactions

Using these two models, we built two classifiers to make decisions about whether to block any of the sensitive permission requests listed in Table 5. We used our exit survey data as ground truth, and used 5-fold cross-validation to evaluate model accuracy.

We calculated the receiver operating characteristic (ROC) to capture the tradeoff between true-positive and false-positive rate. The quality of the classifier can be quantified with a single value by calculating the area under its ROC curve (AUC) [23]. The closer the AUC gets to 1.0, the better the classifier is. When screens were on, the AUC was 0.7, which is 40% better than the random baseline (0.5). When screens were off, the AUC was 0.8, which is 60% better than a random baseline.

7 Discussion

During the study, 80% of our participants deemed at least one permission request as inappropriate. This violates Nissenbaum's notion of "privacy as contextual integrity" because applications were performing actions that defied users' expectations [33]. Felt et al. posited that users may be able to better understand why permission requests are needed if some of these requests are granted via runtime consent dialogs, rather than Android's previous install-time notification approach [14]. By granting permissions at runtime, users will have additional contextual information; based on what they were doing at the time that resources are requested, they may have a better idea of why those resources are being requested.

We make two primary contributions that system designers can use to make more usable permissions systems. We show that the visibility of the requesting application and the frequency at which requests occur are two important factors in designing a runtime consent platform. Also, we show that "prompt-on-first-use" per triplet could be implemented for some sensitive permissions without risking user habituation or annoyance.

Based on the frequency with which runtime permissions are requested (Section 4), it is infeasible to prompt users every time. Doing so would overwhelm them and lead to habituation. At the same time, drawing user attention to the situations in which users are likely to be concerned will lead to greater control and awareness. Thus, the challenge is in acquiring their preferences by confronting them minimally and then automatically inferring *when* users are likely to find a permission request unexpected, and only prompting them in these cases. Our data suggests that participants' desires to block particular permissions were heavily influenced by two main factors: their understanding of the relevance of a permission request to the functionality of the requesting application and their individual privacy concerns.

Our models in Section 6.1 showed that individual characteristics greatly explain the variance between what different users deem appropriate, in terms of access to protected resources. While responses to privacy scales failed to explain these differences, this was not a surprise, as the disconnect between stated privacy preferences and behaviors is well-documented (e.g., [1]). This means that in order to accurately model user preferences, the system will need to learn what a specific user deems inappropriate over time. Thus, a feedback loop is likely needed: when devices are "new," users will be required to provide more input surrounding permission requests, and then based on their responses, they will see fewer requests in the future. Our data suggests that prompting once for each unique (application, permission, application visibility) triplet can serve as a practical mechanism in acquiring users' privacy preferences.

Beyond individual subject characteristics (i.e., personal preferences), participants based their decisions to block certain permission requests on the specific applications making the requests and whether they had contextual cues to indicate that the applications were running (and therefore needed the data to function). Future systems could take these factors into account when deciding whether or not to draw user attention to a particular request. For example, when an application that a user is not actively using requests access to a protected resource, she should be shown a runtime prompt. Our data indicates that, if the user decides to grant a request in this situation, then with probability 0.84 the same decision will hold in future situations where she is actively using that same application, and therefore a subsequent prompt may not be needed. At a minimum, platforms need to treat permission requests from background applications differently than those originating from foreground applications. Similarly, applications running in the background should use passive indicators to communicate when they are accessing particular resources. Platforms can also be designed to make decisions about whether or not access to resources should be granted based on whether contextual cues are present, or at its most basic, whether the device screen is even on.

Finally, we built our models and analyzed our data within the framework of what resources our participants *believed* were necessary for applications to correctly function. Obviously, their perceptions may have been incorrect: if they better understood why a particular resource was necessary, they may have been more permissive. Thus, it is incumbent on developers to adequately communicate why particular resources are needed, as this impacts user notions of contextual integrity. Yet, no mechanisms in Android exist for developers to do this as part of the permission-granting process. For example, one could imagine requiring metadata to be provided that explains how each requested resource will be used, and then automatically integrating this information into permission requests. Tan et al. examined a similar feature on iOS that allows developers to include free-form text in runtime permission dialogs and observed that users were more likely to grant requests that included this text [41]. Thus, we believe that including succinct explanations in these requests would help preserve contextual integrity by promoting greater transparency.

In conclusion, we believe this study was instructive in showing the circumstances in which Android permission requests are made under real-world usage. While prior work has already identified some limitations of deployed mobile permissions systems, we believe our study can benefit system designers by demonstrating several ways in which contextual integrity can be improved, thereby empowering users to make better security decisions.

Acknowledgments

This work was supported by NSF grant CNS-1318680, by Intel through the ISTC for Secure Computing, and by the AFOSR under MURI award FA9550-12-1-0040.

References

[1] ACQUISTI, A., AND GROSSKLAGS, J. Privacy and rationality in individual decision making. *IEEE Security & Privacy* (January/February 2005), 24–30. `http://www.dtc.umn.edu/weis2004/acquisti.pdf`.

[2] ALMOHRI, H. M., YAO, D. D., AND KAFURA, D. Droidbarrier: Know what is executing on your android. In *Proc. of the 4th ACM Conf. on Data and Application Security and Privacy* (New York, NY, USA, 2014), CODASPY '14, ACM, pp. 257–264.

[3] ANDROID DEVELOPERS. System permissions. `http://developer.android.com/guide/topics/security/permissions.html`. Accessed: November 11, 2014.

[4] ANDROID DEVELOPERS. Common Intents. `https://developer.android.com/guide/components/intents-common.html`, 2014. Accessed: November 12, 2014.

[5] ANDROID DEVELOPERS. Content Providers. `http://developer.android.com/guide/topics/providers/content-providers.html`, 2014. Accessed: Nov. 12, 2014.

[6] AU, K. W. Y., ZHOU, Y. F., HUANG, Z., AND LIE, D. Pscout: Analyzing the android permission specification. In *Proc. of the 2012 ACM Conf. on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 217–228.

[7] BARRERA, D., CLARK, J., MCCARNEY, D., AND VAN OORSCHOT, P. C. Understanding and improving app installation security mechanisms through empirical analysis of android. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* (New York, NY, USA, 2012), SPSM '12, ACM, pp. 81–92.

[8] BARRERA, D., KAYACIK, H. G. U. C., VAN OORSCHOT, P. C., AND SOMAYAJI, A. A methodology for empirical analysis of permission-based security models and its application to android. In *Proc. of the ACM Conf. on Comp. and Comm. Security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 73–84.

[9] BODDEN, E. Easily instrumenting android applications for security purposes. In *Proc. of the ACM Conf. on Comp. and Comm. Sec.* (NY, NY, USA, 2013), CCS '13, ACM, pp. 1499–1502.

[10] BUCHANAN, T., PAINE, C., JOINSON, A. N., AND REIPS, U.-D. Development of measures of online privacy concern and protection for use on the internet. *Journal of the American Society for Information Science and Technology 58*, 2 (2007), 157–165.

[11] BUGIEL, S., HEUSER, S., AND SADEGHI, A.-R. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *Proc. of the 22nd USENIX Security Symposium* (Berkeley, CA, USA, 2013), SEC'13, USENIX Association, pp. 131–146.

[12] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–6.

[13] FELT, A. P., CHIN, E., HANNA, S., SONG, D., AND WAGNER, D. Android permissions demystified. In *Proc. of the ACM Conf. on Comp. and Comm. Sec.* (New York, NY, USA, 2011), CCS '11, ACM, pp. 627–638.

[14] FELT, A. P., EGELMAN, S., FINIFTER, M., AKHAWE, D., AND WAGNER, D. How to ask for permission. In *Proceedings of the 7th USENIX conference on Hot Topics in Security* (Berkeley, CA, USA, 2012), HotSec'12, USENIX Association, pp. 7–7.

[15] FELT, A. P., EGELMAN, S., AND WAGNER, D. I've got 99 problems, but vibration ain't one: a survey of smartphone users' concerns. In *Proc. of the 2nd ACM workshop on Security and Privacy in Smartphones and Mobile devices* (New York, NY, USA, 2012), SPSM '12, ACM, pp. 33–44.

[16] FELT, A. P., HA, E., EGELMAN, S., HANEY, A., CHIN, E., AND WAGNER, D. Android permissions: user attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security* (New York, NY, USA, 2012), SOUPS '12, ACM, pp. 3:1–3:14.

[17] FU, H., AND LINDQVIST, J. General area or approximate location?: How people understand location permissions. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society* (2014), ACM, pp. 117–120.

[18] FU, H., YANG, Y., SHINGTE, N., LINDQVIST, J., AND GRUTESER, M. A field study of run-time location access disclosures on android smartphones. *Proc. USEC 14* (2014).

[19] GIBLER, C., CRUSSELL, J., ERICKSON, J., AND CHEN, H. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Proc. of the 5th Intl. Conf. on Trust and Trustworthy Computing* (Berlin, Heidelberg, 2012), TRUST'12, Springer-Verlag, pp. 291–307.

[20] GORLA, A., TAVECCHIA, I., GROSS, F., AND ZELLER, A. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering* (New York, NY, USA, 2014), ICSE 2014, ACM, pp. 1025–1035.

[21] HARBACH, M., HETTIG, M., WEBER, S., AND SMITH, M. Using personal examples to improve risk communication for security & privacy decisions. In *Proc. of the 32nd Annual ACM Conf. on Human Factors in Computing Systems* (New York, NY, USA, 2014), CHI '14, ACM, pp. 2647–2656.

[22] HARBACH, M., VON ZEZSCHWITZ, E., FICHTNER, A., DE LUCA, A., AND SMITH, M. It'sa hard lock life: A field study of smartphone (un) locking behavior and risk perception. In *Symposium on Usable Privacy and Security (SOUPS)* (2014).

[23] HASTIE, T., TIBSHIRANI, R., FRIEDMAN, J., AND FRANKLIN, J. The elements of statistical learning: data mining, inference and prediction. *The Mathematical Intelligencer 27*, 2 (2005), 83–85.

[24] HORNYACK, P., HAN, S., JUNG, J., SCHECHTER, S., AND WETHERALL, D. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proc. of the ACM Conf. on Comp. and Comm. Sec.* (New York, NY, USA, 2011), CCS '11, ACM, pp. 639–652.

[25] JUNG, J., HAN, S., AND WETHERALL, D. Short paper: Enhancing mobile application permissions with runtime feedback and constraints. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* (New York, NY, USA, 2012), SPSM '12, ACM, pp. 45–50.

[26] KELLEY, P. G., CONSOLVO, S., CRANOR, L. F., JUNG, J., SADEH, N., AND WETHERALL, D. A conundrum of permissions: Installing applications on an android smartphone. In *Proc. of the 16th Intl. Conf. on Financial Cryptography and Data Sec.* (Berlin, Heidelberg, 2012), FC'12, Springer-Verlag, pp. 68–79.

[27] KELLEY, P. G., CRANOR, L. F., AND SADEH, N. Privacy as part of the app decision-making process. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2013), CHI '13, ACM, pp. 3393–3402.

[28] KLIEBER, W., FLYNN, L., BHOSALE, A., JIA, L., AND BAUER, L. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis* (New York, NY, USA, 2014), SOAP '14, ACM, pp. 1–6.

[29] LARSON, R., AND CSIKSZENTMIHALYI, M. New directions for naturalistic methods in the behavioral sciences. In *The Experience Sampling Method*, H. Reis, Ed. Jossey-Bass, San Francisco, 1983, pp. 41–56.

[30] LIN, J., SADEH, N., AMINI, S., LINDQVIST, J., HONG, J. I., AND ZHANG, J. Expectation and purpose: understanding users' mental models of mobile app privacy through crowdsourcing. In *Proc. of the 2012 ACM Conf. on Ubiquitous Computing* (New York, NY, USA, 2012), UbiComp '12, ACM, pp. 501–510.

[31] MALHOTRA, N. K., KIM, S. S., AND AGARWAL, J. Internet Users' Information Privacy Concerns (IUIPC): The Construct, The Scale, and A Causal Model. *Information Systems Research 15*, 4 (December 2004), 336–355.

[32] NISSENBAUM, H. Privacy as contextual integrity. *Washington Law Review 79* (February 2004), 119.

[33] NISSENBAUM, H. *Privacy in context: Technology, policy, and the integrity of social life*. Stanford University Press, 2009.

[34] O'GRADY, J. D. New privacy enhancements coming to ios 8 in the fall. `http://www.zdnet.com/new-privacy-enhancements-coming-to-ios-8-in-the-fall-7000030903/`, June 25 2014. Accessed: Nov. 11, 2014.

[35] PANDITA, R., XIAO, X., YANG, W., ENCK, W., AND XIE, T. WHYPER: Towards Automating Risk Assessment of Mobile Applications. In *Proc. of the 22nd USENIX Sec. Symp.* (Berkeley, CA, USA, 2013), SEC'13, USENIX Association, pp. 527–542.

[36] SARMA, B. P., LI, N., GATES, C., POTHARAJU, R., NITA-ROTARU, C., AND MOLLOY, I. Android permissions: A perspective combining risks and benefits. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies* (New York, NY, USA, 2012), SACMAT '12, ACM, pp. 13–22.

[37] SHEBARO, B., OLUWATIMI, O., MIDI, D., AND BERTINO, E. Identidroid: Android can finally wear its anonymous suit. *Trans. Data Privacy 7*, 1 (Apr. 2014), 27–50.

[38] SHKLOVSKI, I., MAINWARING, S. D., SKÚLADÓTTIR, H. H., AND BORGTHORSSON, H. Leakiness and creepiness in app space: Perceptions of privacy and mobile app use. In *Proc. of the 32nd Ann. ACM Conf. on Human Factors in Computing Systems* (New York, NY, USA, 2014), CHI '14, ACM, pp. 2347–2356.

[39] SPREITZENBARTH, M., FREILING, F., ECHTLER, F., SCHRECK, T., AND HOFFMANN, J. Mobile-sandbox: Having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing* (New York, NY, USA, 2013), SAC '13, ACM, pp. 1808–1815.

[40] STEVENS, R., GANZ, J., FILKOV, V., DEVANBU, P., AND CHEN, H. Asking for (and about) permissions used by android apps. In *Proc. of the 10th Working Conf. on Mining Software Repositories* (Piscataway, NJ, USA, 2013), MSR '13, IEEE Press, pp. 31–40.

[41] TAN, J., NGUYEN, K., THEODORIDES, M., NEGRON-ARROYO, H., THOMPSON, C., EGELMAN, S., AND WAGNER, D. The effect of developer-specified explanations for permission requests on smartphone user behavior. In *Proc. of the SIGCHI Conf. on Human Factors in Computing Systems* (2014).

[42] THOMPSON, C., JOHNSON, M., EGELMAN, S., WAGNER, D., AND KING, J. When it's better to ask forgiveness than get permission: Designing usable audit mechanisms for mobile permissions. In *Proceedings of the 2013 Symposium on Usable Privacy and Security (SOUPS)* (2013).

[43] VITTER, J. S. Random sampling with a reservoir. *ACM Trans. Math. Softw. 11*, 1 (Mar. 1985), 37–57.

[44] WEI, X., GOMEZ, L., NEAMTIU, I., AND FALOUTSOS, M. Permission evolution in the android ecosystem. In *Proceedings of the 28th Annual Computer Security Applications Conference* (New York, NY, USA, 2012), ACSAC '12, ACM, pp. 31–40.

[45] WOODRUFF, A., PIHUR, V., CONSOLVO, S., BRANDIMARTE, L., AND ACQUISTI, A. Would a privacy fundamentalist sell their dna for $1000...if nothing bad happened as a result? the westin categories, behavioral intentions, and consequences. In *Proceedings of the 2014 Symposium on Usable Privacy and Security* (2014), USENIX Association, pp. 1–18.

[46] XU, R., SAÏDI, H., AND ANDERSON, R. Aurasium: Practical policy enforcement for android applications. In *Proc. of the 21st USENIX Sec. Symp.* (Berkeley, CA, USA, 2012), Security'12, USENIX Association, pp. 27–27.

[47] ZHANG, Y., YANG, M., XU, B., YANG, Z., GU, G., NING, P., WANG, X. S., AND ZANG, B. Vetting undesirable behaviors in android apps with permission use analysis. In *Proc. of the ACM Conf. on Comp. and Comm. Sec.* (New York, NY, USA, 2013), CCS '13, ACM, pp. 611–622.

[48] ZHU, H., XIONG, H., GE, Y., AND CHEN, E. Mobile app recommendations with security and privacy awareness. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2014), KDD '14, ACM, pp. 951–960.

A Invisible requests

Following list shows the set of applications that have requested the most number of permissions while executing invisibly to the user and the most requested permission types by each respective application.

- *Facebook App*— ACCESS NETWORK STATE, ACCESS FINE LOCATION, ACCESS WIFI STATE ,WAKE LOCK,
- *Google Location*—WAKE LOCK, ACCESS FINE LOCATION, GET ACCOUNTS, ACCESS COARSE LOCATION,
- *Facebook Messenger*—ACCESS NETWORK STATE, ACCESS WIFI STATE, WAKE LOCK, READ PHONE STATE,
- *Taptu DJ*—ACCESS NETWORK STATE, INTERNET, NFC
- *Google Maps*—ACCESS NETWORK STATE, GET ACCOUNTS, WAKE LOCK, ACCESS FINE LOCATION,
- *Google (Gapps)*—WAKE LOCK, ACCESS FINE LOCATION, AUTHENTICATE ACCOUNTS, ACCESS NETWORK STATE,
- *Fouraquare*—ACCESS WIFI STATE, WAKE LOCK, ACCESS FINE LOCATION, INTERNET,
- *Yahoo Weather*—ACCESS FINE LOCATION, ACCESS NETWORK STATE, INTERNET, ACCESS WIFI STATE,

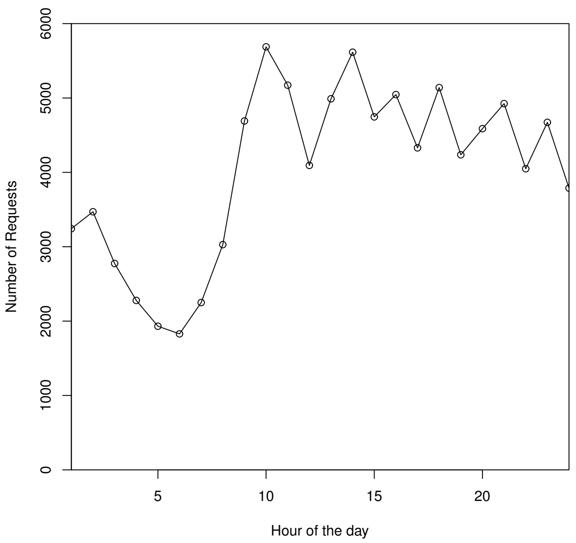
- *Devexpert Weather*—ACCESS NETWORK STATE, INTERNET, ACCESS FINE LOCATION,
- *Tile Game(Umoni)*—ACCESS NETWORK STATE, WAKE LOCK, INTERNET, ACCESS WIFI STATE,

Following is the most frequently requested permission type by applications while running invisibly to the user and the applications who requested the respective permission type most.

- *ACCESS_NETWORK_STATE*— Facebook App, Google Maps, Facebook Messenger, Google (Gapps), Taptu - DJ
- *WAKE_LOCK*—Google (Location), Google (Gapps), Google (GMS), Facebook App, GTalk.
- *ACCESS_FINE_LOCATION*—Google (Location), Google (Gapps), Facebook App, Yahoo Weather, Rhapsody (Music)
- *GET_ACCOUNTS*—Google (Location), Google (Gapps), Google (Login), Google (GM), Google (Vending)
- *ACCESS_WIFI_STATE*—Google (Location), Google (Gapps), Facebook App, Foursqaure, Facebook Messenger
- *UPDATE_DEVICE_STATS*—Google (SystemUI), Google (Location), Google (Gapps)
- *ACCESS_COARSE_LOCATION*—Google (Location), Google (Gapps), Google (News), Facebook App, Google Maps
- *AUTHENTICATE_ACCOUNTS*—Google (Gapps), Google (Login), Twitter, Yahoo Mail, Google (GMS)
- *READ_SYNC_SETTINGS*—Google (GM), Google (GMS), android.process.acore, Google (Email), Google (Gapps)
- *INTERNET*—Google (Vending), Google (Gapps), Google (GM), Facebook App, Google (Location)

B Distribution of Requests

The following graph shows the distribution of requests throughout a given day averaged across the data set.



C Permission Type Breakdown

This table lists the most frequently used permissions during the study period. (per user / per day)

Permission Type	Requests
ACCESS_NETWORK_STATE	41077
WAKE_LOCK	27030
ACCESS_FINE_LOCATION	7400
GET_ACCOUNTS	4387
UPDATE_DEVICE_STATS	2873
ACCESS_WIFI_STATE	2092
ACCESS_COARSE_LOCATION	1468
AUTHENTICATE_ACCOUNTS	1335
READ_SYNC_SETTINGS	836
VIBRATE	740
INTERNET	739
READ_SMS	611
READ_PHONE_STATE	345
STATUS_BAR	290
WRITE_SYNC_SETTINGS	206
CHANGE_COMPONENT_ENABLED_STATE	197
CHANGE_WIFI_STATE	168
READ_CALENDAR	166
ACCOUNT_MANAGER	134
ACCESS_ALL_DOWNLOADS	127
READ_EXTERNAL_STORAGE	126
USE_CREDENTIALS	101
READ_LOGS	94

D User Application Breakdown

This table shows the applications that most frequently requested access to protected resources during the study period. (per user / per day)

Application Name	Requests
facebook.katana	40041
google.process.location	32426
facebook.orca	24702
taptu.streams	15188
google.android.apps.maps	6501
google.process.gapps	5340
yahoo.mobile.client.android.weather	5505
tumblr	4251
king.farmheroessaga	3862
joelapenna.foursquared	3729
telenav.app.android.scout_us	3335
devexpert.weather	2909
ch.bitspin.timely	2549
umonistudio.tile	2478
king.candycrushsaga	2448
android.systemui	2376
bambuna.podcastaddict	2087
contapps.android	1662
handcent.nextsms	1543
foursquare.robin	1408
qisiemoji.inputmethod	1384
devian.tubemate.home	1296
lookout	1158

Phasing: Private Set Intersection using Permutation-based Hashing

Benny Pinkas
Bar-Ilan University, Israel
benny@pinkas.net

Thomas Schneider
TU Darmstadt, Germany
thomas.schneider@ec-spride.de

Gil Segev
Hebrew University, Israel
segev@cs.huji.ac.il

Michael Zohner
TU Darmstadt, Germany
michael.zohner@ec-spride.de

Abstract

Private Set Intersection (PSI) allows two parties to compute the intersection of private sets while revealing nothing more than the intersection itself. PSI needs to be applied to large data sets in scenarios such as measurement of ad conversion rates, data sharing, or contact discovery. Existing PSI protocols do not scale up well, and therefore some applications use insecure solutions instead.

We describe a new approach for designing PSI protocols based on permutation-based hashing, which enables to reduce the length of items mapped to bins while ensuring that no collisions occur. We denote this approach as Phasing, for Permutation-based Hashing Set Intersection. Phasing can dramatically improve the performance of PSI protocols whose overhead depends on the length of the representations of input items.

We apply Phasing to design a new approach for circuit-based PSI protocols. The resulting protocol is up to 5 times faster than the previously best Sort-Compare-Shuffle circuit of Huang et al. (NDSS 2012). We also apply Phasing to the OT-based PSI protocol of Pinkas et al. (USENIX Security 2014), which is the fastest PSI protocol to date. Together with additional improvements that reduce the computation complexity by a logarithmic factor, the resulting protocol improves run-time by a factor of up to 20 and can also have similar communication overhead as the previously best PSI protocol in that respect. The new protocol is only moderately less efficient than an *insecure* PSI protocol that is currently used by real-world applications, and is therefore the first secure PSI protocol that is scalable to the demands and the constraints of current real-world settings.

1 Introduction

Private set intersection (PSI) allows two parties P_1 and P_2 with respective input sets X and Y to compute the intersection $X \cap Y$ of their sets without revealing any information but the intersection itself. Although PSI has been widely studied in the literature, many real-world applications today use an insecure hash-based protocol instead of a secure PSI protocol, mainly because of the insufficient efficiency of current PSI protocols.

In this work we present *Phasing*, Permutation-based Hashing Set Intersection, which is a new approach for constructing PSI protocols based on a hashing technique that ensures that hashed elements can be represented by short strings without any collisions. The overhead of recent PSI protocols depends on the length of these representations, and this new structure of construction, together with other improvements, results in very efficient performance that is only moderately larger than that of the *insecure* protocol that is in current real-world usage.

1.1 Motivating Scenarios

The motivation for this work comes from scenarios where PSI must be applied quite frequently to large sets of data, and therefore performance becomes critical. Moreover, the communication overhead might be even more important than the computation overhead, since in large data centers it is often easier to add computing power than to improve the outgoing communication infrastructure. We describe here three scenarios which require large-scale PSI implementations.

Measuring ad conversion rates Online advertising, which is a huge business, typically measures the success of ad campaigns by measuring the success of converting viewers into customers. A popular way of measuring this value is by computing the conversion rate, which is the percentage of ad viewers who later visit the advertised site or perform a transaction there. For banner ads or services like Google Adwords it is easy to approximate this value by measuring ad click-throughs. However, measuring click-throughs is insufficient in other online advertising settings. One such setting is *mobile advertising*, which is becoming a dominating part of online advertising. Even though mobile ads have a great effect, click-throughs are an insufficient measure of their utility, since it is unlikely, due to small displays and the casual nature of mobile browsing, that a user will click on an ad and, say, purchase a car using his mobile device. Another setting where click rate measurement is unsatisfactory is advertising of offline goods, like groceries, where the purchase itself is done offline.[1]

An alternative method of measuring ad performance is to compare the list of people who have seen an ad with those who have completed a transaction. These lists are held by the advertiser (say, Google or Facebook), and by merchants, respectively. It is often possible to identify users on both ends, using identifiers such as credit card numbers, email addresses, etc. A simple solution, which ignores privacy, is for one side to disclose its list of customers to the other side, which then computes the necessary statistics. Another option is to run a PSI protocol between the two parties. (The protocol should probably be a variant of PSI, e.g. compute total revenues from customers who have seen an ad. Such protocols can be derived from basic PSI protocols.) In fact, Facebook is running a service of this type with Datalogix, Epsilon and Acxiom, companies which have transaction records for a large part of loyalty card holders in the US. According to reports[2], the computation is done using a variant of the *insecure* naive hashing PSI protocol that we describe in §3.1. Our results show that it can be computed using secure protocols even for large data sets.

Security incident information sharing Security incident handlers can benefit from information sharing since it provides them with a global view during incidents. However, incident data is often sensitive and potentially embarrassing. The shared information might reveal information about the business of the company that provided it, or of its customers. Therefore, information is typically shared rather sparsely and protected using legal agreements. Automated large scale sharing will improve security, and there is in fact work to that end, such as the IETF Managed Incident Lightweight Exchange (MILE) effort. Many computations that are applied to the shared data compute the intersection and its variants. Applying PSI to perform these computations can simplify the legal issues of information sharing. Efficient PSI protocols will enable it to be run often and in large scale.

Private contact discovery When a new user registers to a service it is often essential to identify current registered users who are also contacts of the new user. This operation can be done by simply revealing the user's contact list to the service, but can also be done in a privacy preserving manner by running a PSI protocol between the user's contact list and the registered users of the service. This latter approach is used by the TextSecure and Secret applications, but for performance reasons they use the insecure naive hashing PSI protocol described in §3.1.[3]

In these cases each user has a small number of records n_2, e.g., $n_2 = 256$, whereas the service has millions of registered users (in our experiments we use $n_1 = 2^{24}$). It therefore holds that $n_2 \ll n_1$. In our best PSI protocol, the client needs only $\mathsf{O}(n_2 \log n_1)$ memory, $\mathsf{O}(n_2)$ symmetric cryptographic operations and $\mathsf{O}(n_1)$ cheap hash table lookups, and the communication is $\mathsf{O}(n_1 \log n_1)$. (The communication overhead is indeed high as it depends on n_1, but this seems inevitable if brute force searches are to be prevented.)

1.2 Our Contributions

Our goal in this work is to enable PSI computations for large scale sets that were previously beyond the capabilities of state-of-the-art protocols. The constructions that we design in this work improve performance by more than an order of magnitude. We obtain these improvements by generalizing the hashing approach of [22] and applying it to generic secure computation-based PSI protocols. We replace the hash function in [22] by a permutation which enables us to reduce the bit-length of internal representations. Moreover, we suggest several improvements to the OT-based PSI protocol of [22]. We explain our contributions in more detail next:

Phasing: Using permutation-based hashing to reduce the bit-length of representations. The overhead of the best current PSI protocol [22] is linear in the length of the representations of items in the sets (i.e., the ids of items in the sets). The protocol maps items into bins, and since each bin has very few items in it, it is tempting to hash the ids to shorter values and trust the birthday paradox to ensure that no two items in the same bin are hashed to the same representation. However, a closer examination shows that to ensure that the collision probability is smaller than $2^{-\lambda}$, the length of the representation must be at least λ bits, which is too long.

In this work we utilize the permutation-based hashing techniques of [1] to reduce the bit-length of the ids of items that are mapped to bins. These ideas were suggested in an algorithmic setting to reduce memory us-

[1] See, e.g., `http://www.reuters.com/article/2012/10/01/us-facebook-ads-idUSBRE8900I120121001` .

[2] See, e.g., `https://www.eff.org/deeplinks/2012/09/deep-dive-facebook-and-datalogix-whats-actually-getting-shared-and-how-you-can-opt`.

[3] See `https://whispersystems.org/blog/contact-discovery/` and `https://medium.com/@davidbyttow/demystifying-secret-12ab82fda29f` , respectively.

age, and as far as we know this is the first time that they are used in a cryptographic or security setting to improve performance. Essentially, when using β bins the first $\log\beta$ bits in an item's hashed representation define the bin to which the item is mapped, and the other bits are used in a way which provably prevents collisions. This approach reduces the bit-length of the values used in the PSI protocol by $\log\beta$ bits, and this yields reduced overhead by up to 60%-75% for the settings we examined.

Circuit-Phasing: Improved circuit-based PSI. As we discuss in §3.4 there is a great advantage in using generic secure computation for computing PSI, since this enables to easily compute variants of the basic PSI functionality. Generic secure computation protocols evaluate Boolean circuits computing the desired functionality. The best known circuit for computing PSI was based on the Sort-Compare-Shuffle circuit of [12]. We describe Circuit-Phasing, a new generic protocol that uses hashing (specifically, Cuckoo hashing and simple hashing) and secure circuit evaluation. In comparison with the previous approach, our circuits have a smaller number of AND gates, a lower depth of the circuit (which affects the number of communication rounds in some protocols), and a much smaller memory footprint. These factors lead to a significantly better performance.

OT-Phasing: Improved OT-based PSI. We introduce the OT-Phasing protocol which improves the OT-based PSI protocol of [22] as follows:

- **Improved computation and memory.** We reduce the length of the strings that are processed in the OT from $\mathsf{O}(\log^2 n)$ to $\mathsf{O}(\log n)$, which results in a reduction of computation and memory complexity for the client from $\mathsf{O}(n\log^2 n)$ to $\mathsf{O}(n\log n)$.
- **3-way Cuckoo hashing.** We use 3 instead of 2 hash functions to generate a more densely populated Cuckoo table and thus decrease the overall number of bins and hence OTs.

OT-Phasing improves over state-of-the-art PSI both in terms of run-time and communication. Compared to the previously fastest PSI protocol of [22], our protocol improves run-time by up to factor 10 in the WAN setting and by up to factor 20 in the LAN setting. Furthermore, our OT-Phasing protocol in some cases achieves similar communication as [18], which was shown to achieve the lowest communication of all PSI protocols [22].

1.3 Outline

We give preliminary information in §2 and summarize related work in §3. In §4 we describe Phasing, our optimization for permutation-based hashing that reduces the bit-length of elements in PSI. Afterwards, we apply Phasing to generic secure computation protocols, and present Circuit-Phasing, our new approach for circuit-based PSI §5. Thereafter, we apply Phasing to the previously fastest OT-based PSI protocol of [22] and present several optimizations in §6. In §7 we analyze the hashing failure probability of Circuit- and OT-Phasing. Finally, we provide an evaluation of our PSI protocols in §8.

2 Preliminaries

2.1 Notation

We denote the parties as P_1 and P_2. For all protocols we assume that P_2 obtains the output. The respective input sets are denoted as X and Y, with sizes $n_1 = |X|$ and $n_2 = |Y|$. Often $n_1 = n_2$ and we use the notation $n = n_1 = n_2$. We assume that elements are of bit-length σ.

We call the symmetric security parameter κ, the bit-length of the elliptic curves φ, and the statistical security parameter λ. Throughout the paper we assume 128-bit security, i.e., $\kappa = 128$, $\varphi = 283$ (using Koblitz-curves), and $\lambda = 40$. For symmetric encryption we use AES-128.

We refer to the concatenation of bit-strings by $||$, to the exclusive-OR (XOR) operation by $\oplus$, and to the i-th element in a sequence S by $S[i]$. In many protocols, we shorten the size of hash values that are sent to $\ell = \lambda + \log_2(n_1) + \log_2(n_2)$ instead of 2κ. This yields collision probability $2^{-\lambda}$, which is suited for most applications.

2.2 Security

Two types of adversaries are typically discussed in the secure computation literature: A *semi-honest* adversary is trusted to follow the protocol, but attempts to learn as much information as possible from the messages it receives. This adversary model is appropriate for scenarios where execution of the correct software is enforced by software attestation or where an attacker might obtain the transcript of the protocol *after* its execution, either by stealing it or by legally enforcing its disclosure. In contrast, a *malicious* adversary can behave arbitrarily. Most work on PSI was in the semi-honest setting. Protocols that are secure against malicious adversaries, e.g., [9, 10, 14], are considerably less efficient. We focus on optimal performance and therefore design protocols secure against semi-honest adversaries only. Furthermore, the security of the protocols is proven in the random oracle model, as is justified in the full version [21].

2.3 Hashing to Bins

Our protocols hash the input items to bins and then operate on each bin separately. In general, our hashing schemes use a table T consisting of β bins. An element e is mapped to the table by computing an address $a = H(e)$ using a hash function H that is modeled as a random function. A value related to e is then stored in bin $T[a]$.

There is a rich literature on hashing schemes, which differ in the methods for coping with collisions, the complexity for insertion/deletion/look-up, and the utilization of storage space. In [9, 10, 22], hashing to bins was used to improve the number of comparisons that are performed in PSI protocols. In the following, we detail the two most promising hashing schemes for use in PSI, according to [22]: *simple hashing* and *Cuckoo hashing*. For the OT-based PSI protocol of [22] it was shown that a combination of simple hashing (for P_1) and Cuckoo hashing (for P_2) results in the best performance.

2.3.1 Simple Hashing

Simple hashing builds the table T by mapping each element e to bin $T[H(e)]$ and appending e to the bin. Each bin must, of course, be able to store more than one element. The size of the most populated bin was analyzed in [23], and depends on the relation between the number of bins and the total number of elements. Most importantly for our application, when hashing n elements into $\beta = n$ bins, it was shown that the maximum number of elements in a bin is $\frac{\ln n}{\ln \ln n}(1+o(1))$. In §7.1 we give a theoretical and an empirical analysis of the maximum number of elements in a bin.

2.3.2 Cuckoo Hashing

Cuckoo hashing [19] uses h hash functions $H_1, ..., H_h$ to map an element e to a bin using either one of the h hash functions. (Typically, h is set to be $h = 2$; we also use $h = 3$.) In contrast to simple hashing, it allows at most one element to be stored in a bin. If a collision occurs, Cuckoo hashing evicts the element in the bin and performs the insertion again for the evicted element. This process is repeated until an empty bin is found for the evicted element. If the resulting sequence of insertion attempts fails a certain number of times, the current evicted element is placed in a special bin called stash. In [16] it was shown that for $h = 2$ hash functions, $\beta = 2(1+\varepsilon)n$ bins, and a stash of size $s \leq \ln n$, the insertion of elements fails with small probability of $O(n^{-s})$, which is smaller than $n^{-(s-1)}$ for sufficiently large values of n (cf. §7.2).

2.4 Oblivious Transfer

1-ouf-of-2 oblivious transfer (OT) [8] is a protocol where the receiver with choice bit c, chooses one of two strings (x_0, x_1) held by the sender. The receiver receives x_c but gains no information about x_{1-c}, while the sender gains no information about c.

OT extension protocols [2, 17] precompute a small number (say, $\kappa = 128$) of "real" public-key-based OTs, and then compute any polynomial number of OTs using symmetric-key cryptography alone. The most efficient OT variant that we use computes *random OT*. In that protocol the sender has no input but obtains random (x_0, x_1) as output, while the receiver with input c obtains x_c [2]. The advantage of this protocol is that the sender does not need to send messages based on its inputs, as it does not have any inputs, and instead computes them on-the-fly during the OT extension protocol. As a result, the communication overhead of the protocol is greatly reduced.

An additional improvement that we use, described in [17], efficiently computes 1-out-of-N OT for short strings. The communication for a random 1-out-of-N OT (for $3 \leq N \leq 256$) is only 2κ-bits, whereas the communication for a random 1-out-of-2 OT is κ-bits. The computation for a random 1-out-of-N OT amounts to four pseudo-random generator (PRG) and one correlation-robust function (CRF) evaluations for the receiver and two PRG and N CRF evaluations for the sender. In addition, if the sender only requires $i \leq N$ outputs of the OT, it only needs to perform i CRF evaluations.

We use 1-out-of-N OT since we have to perform OTs for every bit of an element. By using 1-out-of-N OT for $N = 2^\mu$, we process μ bits in parallel with communication equal to that of processing two bits. We denote m 1-out-of-N OTs on ℓ-bit strings by $\binom{N}{1}$-OT$_\ell^m$.

2.5 Generic Secure Computation

Generic secure two-party computation protocols allow two parties to securely evaluate any function that can be expressed as a Boolean circuit. The communication overhead and the number of cryptographic operations that are computed are linear in the number of non-linear (AND) gates in the circuit, since linear (XOR) gates can be evaluated "for free" in current protocols. Furthermore, some protocols require a number of interaction rounds that are linear in the AND depth of the circuit. The two main approaches for generic secure two-party computation on Boolean circuits are *Yao's garbled circuits* [25] and the protocol by *Goldreich-Micali-Wigderson* [11]. We give a summary of these protocols in the full version [21].

3 Related Work

We reflect on existing PSI protocols by following the classification of PSI protocols in [22]: the *naive hashing* protocol (§3.1), *server-aided* PSI protocols (§3.2), *public-key cryptography-based* PSI protocols (§3.3), *generic secure computation-based* PSI protocols (§3.4), and *OT-based* PSI protocols (§3.5). For each category, we review existing work and outline the best performing protocol, according to [22].

3.1 (Insecure) Naive Hashing

In the naive hashing protocol, detailed in the full version [21], P_1 permutes and hashes its elements, and sends the results to P_2 which compares these values to the hashes of its elements. This approach is very efficient and is currently employed in practice, but it allows P_2 to brute-force the elements of P_1 if they do not have high entropy. Furthermore, even if inputs elements have high entropy, forward-secrecy is not provided since P_2 can check at any later time whether an element was in X.

3.2 Server-Aided PSI

To increase the efficiency of PSI, protocols that use a semi-trusted third party were proposed [15]. These protocols are secure as long as the third party does not collude with any of the participants. We mention this set of protocols here for completeness, as they require different trust assumptions as protocols involving no third party.

The protocol of [15] has only a slightly higher overhead than the naive hashing PSI solution described in §3.1. In that protocol, P_1 samples a random κ-bit key k and sends it to P_2. Both parties compute $h_i = F_k(x_i)$ (resp. $h'_j = F_k(y_j)$), where F_k is a pseudo-random permutation that is parametrized by k. Both parties then send the hashes to the third party (in randomly permuted order) who then computes $I = h_i \cap h'_j$, for all $1 \le i \le n_1$ and $1 \le j \le n_2$ and sends I to P_2. P_2 obtains the intersection by computing $F_k^{-1}(e)$ for each $e \in I$.

3.3 Public-Key Cryptography based PSI

The first protocols for PSI were outlined in [13, 18] and were based on the Diffie-Hellmann (DH) key exchange. The overhead of these protocols is $\mathsf{O}(n)$ exponentiations. In [9, 10], a PSI protocol based on El-Gamal encryption was introduced that uses oblivious polynomial evaluation and requires $\mathsf{O}(n \log\log(n))$ public-key encryptions (the advantage of that protocol was that its security was not based on the random oracle model). A PSI protocol that uses blind-RSA was introduced in [3].

We implement the DH-based protocol of [13, 18] based on elliptic-curve-cryptography, which was shown to achieve lowest communication in [22]. We describe the protocol in the full version [21].

3.4 PSI based on Generic Protocols

Generic secure computation can be used to perform PSI by encoding the intersection functionality as a Boolean circuit. The most straightforward method for this encoding is to perform a pairwise-comparison which compares each element of one party to all elements of the other party. However, this circuit uses $\mathsf{O}(n^2)$ comparisons and hence scales very poorly for larger set sizes [12]. The *Sort-Compare-Shuffle (SCS)* circuit of [12] is much more efficient. As indicated by its name, the circuit first *sorts* the union of the elements of both parties, then *compares* adjacent elements for equality, and finally *shuffles* the result to avoid information leakage. The sort and shuffle operations are implemented using a sorting network of only $\mathsf{O}(n \log n)$ comparisons, and the comparison step requires only $\mathsf{O}(n)$ comparisons.

The work of [12] describes a size-optimized version of this circuit for use in Yao's garbled circuits; [22] describes a depth-optimized version for use in the GMW protocol. The size-optimized SCS circuit has $\sigma(3n\log_2 n + 4n)$ AND gates[4] and AND depth $(\sigma + 2)\log_2(2n) + \log_2(\sigma) + 1$ while the depth-optimized SCS circuit has about the same number of gates and AND depth of $(\log_2(\sigma) + 4)\log_2(2n)$, for $n = (n_1 + n_2)/2$.

PSI protocols based on generic secure computation have higher run-time and communication complexity than most special-purpose PSI protocols [4, 22]. Yet, these protocols are of great importance since they enable to easily compute any functionality that is based on basic PSI. Consider, for example, an application that needs to find if the size of the intersection is greater than some threshold, or compute the sum of revenues from items in the intersection. Computing these functionalities using specialized PSI protocols requires to change the protocols, whereas a PSI protocol based on generic computation can be adapted to compute these functionalities by using a slightly modified circuit. In other words, changing specialized protocols to have a new functionality requires to employ a cryptographer to design a new protocol variant, whereas changing the functionality of a generic protocol only requires to design a new circuit computing the new functionality. The latter task is of course much simpler. An approximate PSI protocol that uses generic secure computation protocols in combination with Bloom filters was given in [24].

3.5 OT-based PSI

OT-based PSI protocols are the most recent category of PSI protocols. Their research has been motivated by recent efficiency improvements in OT extension. The garbled Bloom filter protocol of [7] was the first OT-based PSI protocol and was improved in [22]. A novel OT-based PSI protocol, which we denote *OT-PSI* protocol,

[4] The original description of the SCS circuit in [12] embedded input keys into AND gates in the sort circuit to reduce communication. We did not use this optimization in our implementation.

was introduced in [22], combining OT and hashing to achieve the best run-time among all analyzed PSI protocols. We next summarize the OT-PSI protocol of [22] and give a detailed description in the full version [21].

The abstract idea of the OT-PSI protocol is to have both parties hash their elements into bins using the same hash function (Step 1, cf. §3.5.1) and compare the elements mapped to the same bin. The comparison is done using OTs that generate random masks from the elements (Step 2, cf. §3.5.2), such that the intersection of the random masks corresponds to the intersection of the original inputs (Step 3, cf. §3.5.3). Finally, the intersection of the elements in the stash is computed (§3.5.4). We give the overhead of the protocol in §3.5.5.

3.5.1 PSI via Hashing to Bins

In the first step of the protocol, the parties map their elements into their respective hash tables T_1 and T_2, consisting of $\beta = h(1+\varepsilon)n_2$ bins (cf. §7). P_2 uses Cuckoo hashing with h hash functions (with $h = 2$), and obtains a one-dimensional hash table T_2. P_1 hashes each item h times (once for each hash function) using *simple hashing* and obtains a two-dimensional hash table T_1 (where the first dimension addresses the bin and the second dimension the elements in the bin). Each party then pads all bins in its table to the maximum size using respective dummy elements: P_1 pads each bin to max_β elements using a dummy element d_1 (where max_β is computed using β and n_1 as detailed in §7 to set the probability of mapping more items to a bin to be negligible), while P_2 fills each empty bin with dummy element d_2 (different than d_1). The padding is performed to hide the number of elements that were mapped to a specific bin, which would leak information about the input.

3.5.2 Masking via OT

After the hashing, the parties use OT to generate an ℓ-bit random mask for each element in their hash table.

Naively, for each bin, and for each item that P_2 mapped to the bin, the parties run a 1-out-of-2 OT for each bit of this item. P_2 is the receiver and its input to the OT is the value of the corresponding bit in the single item that it mapped to the bin. P_1's input is two random ℓ-bit strings. After running these OTs for all σ bits of the item, P_1 sends to P_2 the XOR of the strings corresponding to the bits of P_1's item. Note that if P_1's item is equal to that of P_2 then the sent value is equal to the XOR of the output strings that P_2 received in the OTs. Otherwise the values are different with high probability, which depends on the length ℓ of the output strings.

This basic protocol was improved upon in [22] in several ways:

- Recall that OT extension is more efficient when applied to 1-out-of-N OT [17]. Therefore, the protocol uses μ-bit characters instead of a binary representation. It splits the elements into t μ-bit characters, and uses t invocations of 1-out-of-N OT where $N = 2^\mu$, instead of $t\mu$ invocations of 1-out-of-2 OT.
- In each bin the parties run OTs for all max_β items that P_1 mapped to the bin, and to all characters in these items. P_2's inputs are the same for all max_β OTs corresponding to the same character. Thus, the parties could replace them with a single OT, where the output string of the OT has max_β longer size.
- Recall that random OT, where the protocol randomly defines the inputs of P_1, is more efficient than an OT where P_1 chooses these inputs by itself. For the purpose of PSI the protocol can use random OT. It is also important to note that if P_1 mapped $m < max_\beta$ elements to a bin, it only needs to evaluate inputs for m random OTs in this bin and not for all max_β random OTs that are taking place. This improves the overhead of the protocol.

3.5.3 Intersection

The parties compute the intersection of their elements using the random masks (XOR values) generated during Step 2: P_1 generates a set V as the masks for all of its non-dummy elements. P_1 then randomly permutes the set V to hide information about the number of elements in each bin, and sends V to P_2. P_2 computes the intersection $X \cap Y$ by computing the plaintext intersection between V and the set of XOR values that it computed.

3.5.4 Including a Stash

The OT-based PSI protocol of [22] uses Cuckoo hashing with a stash of size s. The intersection of P_2's elements with P_1's elements is done by running the masking procedure of Step 2 for all s items in the stash, comparing them with all n_1 items in P_1's input. Finally, P_1 sends the masks it computed to P_2 (in randomly permuted order) which can then check the intersection as in Step 3.

3.5.5 Overhead

The overhead of this protocol is linear in the bit-length of the input elements. Therefore, any reduction in the bit-length of the inputs directly results in a similar improvement in the overhead.

For readers interested in the exact overhead of the protocol, we describe here the details of the overhead. In total, the parties have to evaluate random $\binom{N}{1}$-$\mathrm{OT}^{\beta t}_{max_\beta \ell}$ +$\binom{N}{1}$-$\mathrm{OT}^{st}_{n_1 \ell}$ and send $(h+s)n_1$ masks of ℓ-bit length, where $\beta = h(n_2+\varepsilon)$, $N = 2^\mu$, $t = \lceil \sigma/\mu \rceil$, $\ell = \lambda +$

$\log_2(n_1) + \log_2(n_2)$, and s is the size of the stash. To be exact, the server has to perform $2t(\beta + s)$ pseudo-random generator evaluations during OT extension, $(h+s)n_1t$ correlation-robust function evaluations to generate the random masks, and send $(2+s)n_1\ell$ bits. The client has to perform $4t(\beta+s)$ pseudo-random generator evaluations during OT extension, $n_2tmax_\beta\ell/o + sn_1t\ell/o$ correlation-robust function evaluations to generate the random masks, and send $2(\beta+s)t\kappa$ bits during OT extension, where o is the output length of the correlation-robust function. Note especially that the client has to evaluate the correlation-robust function $\mathsf{O}(n\log^2 n)$ times to generate the random bits which represent the masks of the server's elements. This cost can become prohibitive for larger sets, as we will show in our evaluation in §8.

4 Permutation-based Hashing

The overhead of the OT-based PSI protocol of [22] and of the circuit-based PSI protocols we describe in §5 depends on the bit-lengths of the items that the parties map to bins. The bit-length of the stored items can be reduced based on a permutation-based hashing technique that was suggested in [1] for reducing the memory usage of Cuckoo hashing. That construction was presented in an algorithmic setting to improve memory usage. As far as we know this is the first time that it is used in secure computation or in a cryptographic context.

The construction uses a Feistel-like structure. Let $x = x_L|x_R$ be the bit representation of an input item, where $|x_L| = \log\beta$, i.e. is equal to the bit-length of an index of an entry in the hash table. (We assume here that the number of bins β in the hash table is a power of 2. It was shown in [1] how to handle the general case.) Let $f()$ be a random function whose range is $[0, \beta-1]$. Then item x is mapped to bin $x_L \oplus f(x_R)$. The value that is stored in the bin is x_R, which has a length that is shorter by $\log\beta$ bits than the length of the original item. This is a great improvement, since the length of the stored data is significantly reduced, especially if $|x|$ is not much greater than $\log\beta$. As for the security, it can be shown based on the results in [1] that if the function f is k-wise independent, where $k = \text{polylog}\, n$, then the maximum load of a bin is $\log n$ with high probability.

The structure of the mapping function ensures that if two items x, x' store the same value in the same bin then it must hold that $x = x'$: if the two items are mapped to the same bin, then $x_L \oplus f(x_R) = x'_L \oplus f(x'_R)$. Since the stored values satisfy $x_R = x'_R$ it must also hold that $x_L = x'_L$, and therefore $x = x'$.

As a concrete example, assume that $|x| = 32$ and that the table has $\beta = 2^{20}$ bins. Then the values that are stored in each bin are only 12 bits long, instead of 32 bits in the original scheme. Note also that the computation of the bin location requires a single instantiation of f, which can be implemented with a medium-size lookup table.

A comment about an alternative approach An alternative, and more straightforward approach for reducing the bit-length could map x using a random *permutation* $p()$ to a random $|x|$-bit string $p(x)$. The first $\log\beta$ bits of $p(x)$ are used to define the bin to which x is mapped, and the value stored in that bin holds the remaining $|x| - \log\beta$ bits of $p(x)$. This construction, too, has a shorter length for the values that are stored in the bins, but it suffers from two drawbacks: From a performance perspective, this construction requires the usage of a random *permutation* on $|x|$ bits, which is harder to compute than a random *function*. From a theoretical perspective, it is impossible to have efficient constructions of k-wise independent permutations, and therefore we only know how to prove the $\log n$ maximum load of the bins under the stronger assumption that the permutation is random.

5 Circuit-Phasing

PSI protocols that are based on generic secure computation are of great importance due to their flexibility (cf. §3.4 for details). The best known construction of a circuit computing the intersection (of σ-bit elements) is the SCS circuit of [12] with about $3n\sigma\log_2 n$ AND gates and an AND depth of $\Theta(\log_2\sigma \cdot \log_2 n)$. We describe a new construction of circuits with the same order of AND gates (but with smaller constants), and a much smaller depth. Our experiments, detailed in §8.1, demonstrate that the new circuits result in much better performance.

The new protocol, which we denote as Circuit-Phasing, is based on the two parties mapping their inputs to hash tables before applying the circuit. The idea is similar to the OT-based PSI protocol of [22] described in §3.5, but instead of using OTs for the comparisons, the protocol evaluates a pairwise-comparison circuit between each bin of P_1 and P_2 in parallel:

- Both parties use a table of size $\beta = \mathsf{O}(n)$ to store their elements. Our analysis (§7) shows that setting $\beta = 2.4n$ reduces the error probability to be negligible for reasonable input sizes ($2^8 \leq n \leq 2^{24}$) when setting the stash size according to Tab. 4.
- P_2 maps its input elements to β bins using Cuckoo hashing with two hash functions and a stash; empty bins are padded with a dummy element d_2.
- P_1 maps its input elements into β bins using simple hashing. The size of the bins is set to be max_β, a parameter that is set to ensure that no bin overflows (see §7.1). The remaining slots in each bin are padded with a dummy element $d_1 \neq d_2$. The analy-

sis described in §7.1 shows how max_β is computed and is set to a value smaller than $\log_2 n$.

- The parties securely evaluate a circuit that compares the element that was mapped to a bin by P_2 to each of the max_β elements mapped to it by P_1.
- Finally, each element in P_2's stash is checked for equality with all n_1 input elements of P_1 by securely evaluating a circuit computing this functionality.
- To reduce the bit-length of the elements in the bins, and respectively the circuit size, the protocol uses permutation-based hashing as described in §4. (Note that using this technique is impossible with SCS circuits of [12].)

A detailed analysis of the circuit size and depth Let m be the size of P_1's input to the circuit with $m = \beta max_\beta + sn_1$, i.e., for each of the β bins, P_1 inputs max_β items as well as n_1 items for each of the s positions in the stash. The circuit computes a total of m comparisons between the elements of the two parties. Each element is of length σ' bits, which is the reduced length of the elements after being mapped to bins using permutation-based hashing, i.e. $\sigma' = \sigma - \log_2 \beta$.

A comparison of two σ'-bit elements is done by computing the bitwise XOR of the elements and then a tree of $\sigma' - 1$ OR gates, with depth $\lceil \log_2 \sigma' \rceil$. The topmost gate of this tree is a NOR gate. Afterwards, the circuit computes the XOR of the results of all comparisons involving each item of P_2. (Note that at most one of the comparisons results in a match, therefore the circuit can compute the XOR, rather than the OR, of the results of the comparisons.) Overall, the circuit consists of about $m \cdot (\sigma' - 1) \approx n_1 \cdot (max_\beta + s) \cdot (\sigma' - 1)$ non-linear gates and has an AND depth of $\lceil \log_2 \sigma \rceil$.

Advantages Circuit-Phasing has several advantages over the SCS circuit:

- Compared to the number of AND gates in the SCS circuit, which is $3n\sigma \log n$, and recalling that $\sigma' < \sigma$, and that max_β was shown in our experiments to be no greater than $\log n$, the number of non-linear gates in Circuit-Phasing is smaller by a factor greater than 3 compared to the number of non-linear gates in the SCS circuit (even though both circuits have the same big "O" asymptotic sizes).
- The main advantage of Circuit-Phasing is the low AND depth of $\log_2(\sigma)$, which is also independent of the number of elements n. This affects the overhead of the GMW protocol that requires a round of interaction for every level in the circuit.
- Another advantage of Circuit-Phasing is its simple structure: the same small comparison circuit is evaluated for each bin. This property allows for a SIMD (Single Instruction Multiple Data) evaluation with a very low memory footprint and easy parallelization.

Hashing failures: The correct performance of the protocol depends on the successful completion of the hashing operations: The Cuckoo hashing must succeed, and the simple hashing must not place more than max_β elements in each bin. Tables of size $2(1+\varepsilon)n$ and $max_\beta = \mathsf{O}(\log n)$ guarantee these properties with high probability. We analyze the exactly required table sizes in §7 and set them to be negligible in the statistical security parameter λ.

6 OT-Phasing

We improve the OT-PSI protocol of [22] by applying the following changes to the protocol:

- Reducing the bit-length of the items using the permutation-based hashing technique described in §4. This improvement reduces the length of the items from $|x|$ bits to $|x| - \beta$ bits, where β is the size of the tables, and consequently reduces the number of OTs by a factor of $\beta/|x|$.
- Using OTs on a single mask instead of on $\mathsf{O}(\log n)$ masks before. This improvement is detailed in §6.1.
- Improving the utilization of bins by using 3-way Cuckoo hashing (§6.2).

We call the resulting PSI protocol that combines all these optimizations OT-Phasing. In the full version [21], we evaluate the performance gain of each optimization individually and micro-benchmark the resulting protocol.

6.1 A Single Mask per Bin

In order to hide information about the number of items that were mapped to a bin, the original OT-PSI protocol of [22] (cf. §3.5) padded all bins to a maximum size of $max_\beta = \mathsf{O}(\log n)$. The protocol then ran OTs on max_β masks of ℓ-bit length where the parties had to generate and process all of the max_β masks. We describe here a new construction that enables the parties to compute only a constant number of masks per element, regardless of the number of elements that were mapped to the bin by P_1. While this change seems only small, it greatly increases the performance and scalability of the protocol (cf. iterative performance improvements in the full version [21]). In particular, this change results in two improvements to the protocol:

- The number of symmetric cryptographic operations to generate the masks is reduced from $\mathsf{O}(\log^2 n)$ to $\mathsf{O}(\log n)$. Furthermore, note that P_2 had to compute the plaintext intersection between his $n_2 max_\beta$ generated masks and the $2n_1$ masks sent by P_1. This also greatly improves the memory footprint and plaintext intersection.

- In the previous OT-based protocol, a larger value of the parameter max_β reduced the failure probability of the simple hashing procedure used by P_1, but increased the string size in the OTs. In the new protocol the value of max_β does not affect the overhead. Therefore P_1 can use arbitrarily large bins and ensure that the mapping that it performs never fails.

Recall that in the OT-based PSI protocol of [22] (cf. §3.5) the parties had inputs of t characters, where each character was μ bits long, and we used the notation $N = 2^\mu$. The parties performed OTs on strings of max_β masks per bin. Each mask had length $\ell = \lambda + \log_2(n_1) + \log_2(n_2)$ bits, corresponded to an element that P_1 mapped to the bin, and included a 1-out-of-N random-OT for each of the t characters of this element. P_1 was the sender, received all the N sender input-strings of each OT, and chose from them the one corresponding to the value of the character in its own element. P_2 was the receiver and received the string corresponding to the value of the character in its own element. Then P_1 computed the XOR of the t strings corresponding to the t characters of its element and sent this XOR value to P_2, which compared it to the XOR of its t outputs from OT.

The protocol can be improved by running the t 1-out-of-N OTs on a *single* mask per bin. Denote by u the actual number of items mapped by P_1 to a bin. The value of u is not revealed to P_2 in the new protocol and therefore there is no need to pad the bin with dummy items. Denote the single item that P_2 mapped to the bin as $y = y_1, \ldots, y_t$, and the u items that P_1 mapped to the bin as $x^1, \ldots, x^u$, where each x^i is defined as $x^i = x^i_1, \ldots, x^i_t$.

Define the input strings to the j-th OT as $\{s_{j,\ell}\}_{\ell=1\ldots N}$. The protocol that is executed is a random OT and therefore these strings are chosen by the protocol and not by P_1. The parties run a single set of t OTs and P_2 learns the t strings $s_{1,y_1}, \ldots, s_{t,y_t}$. It computes their XOR $S_{P_2} = s_{1,y_1} \oplus \cdots \oplus s_{t,y_t}$, and the value $H(S_{P_2})$, where $H()$ is a hash function modeled as a random oracle.

P_1 learns all the Nt strings generated in the random-OT protocols. For each input element x^i that P_1 mapped to the bin, it computes the XOR of the strings corresponding to the characters of the input, namely $S^i_{P_1} = s_{1,x^i_1} \oplus \cdots \oplus s_{t,x^i_t}$, and then computes the value $H(S^i_{P_1})$. Note that over all bins, P_1 needs to perform this computation only $O(n_1)$ times and compute $O(n_1)$ hash values. P_1 then sends all these values to P_2 in randomly permuted order. P_2 computes the intersection between these values and the $H(S_{P_2})$ values that it computed in the protocol.

Efficiency: P_2 computes only a single set of t OTs per bin on one mask, compared to t OTs on max_β masks in the OT-based protocol of [22]. As for P_1's work, it computes a single set of OTs per bin, and in addition computes a XOR of strings and a hash for each of its $O(n_1)$ input elements. This is a factor of $max_\beta = O(\log n_1)$ less work as before. Communication is only $O(n\sigma)$ strings, as before.

Security: Assuming that the OT protocols are secure and that the parties are semi-honest, the only information that is received by any party in the protocol is the $H(S^i_{P_1})$ values that are sent from P_1 to P_2. For all values in the intersection of the input sets of the two parties, P_1 sends to P_2 the same hash values as those computed by P_2. Consider the set of input elements $\bar{X}$ that are part of P_1's input and are not in P_2's input, and the set of XOR values corresponding to $\bar{X}$. There might be linear dependencies between the XOR values of $\bar{X}$, but it holds with overwhelming probability that all these values are different, and they are also all different from the XOR values computed by P_2. Therefore, the result of applying a random hash function $H()$ to these values is a set of random elements in the range of the hash function. This property enables to easily provide a simulation based proof of security for the protocol.

6.2 3-Way Cuckoo Hashing

The original OT-based PSI protocol of [22] uses Cuckoo hashing which employs two hash functions to map elements into bins. It was shown in [20] that if n elements are mapped to $2(1+\varepsilon)n$ bins, Cuckoo hashing succeeds with high probability for $\varepsilon > 0$. This means that Cuckoo hashing achieves around 50% utilization of the bins. If the number of hash functions h is increased to $h > 2$, a much better utilization of bins can be achieved [6]. However, using h hash functions in our protocol requires P_1 to map each element h times into its bins using simple hashing and requires P_1 to send hn_1 masks in the intersection step of the protocol.

We detail in Tab. 1 the utilization and total communication of our PSI protocol for $n_1 = n_2 = 2^{20}$ and $n_2 = 2^8 \ll n_1 = 2^{20}$, for $\sigma = 32$-bit elements with different numbers of hash functions. We observe that there is a tradeoff between the communication for the OTs and the communication for the masks that are sent by P_1. Our goal is to minimize the total communication, and this is achieved for $h = 3$ hash functions in the setting of $n_1 = n_2$ and for $h = 2$ in the setting of $n_2 \ll n_1$. For $n_1 = n_2$ using $h = 3$ instead of $h = 2$, as in the original protocol of [22], reduces the overall communication by 33%.

Hashing failures: We observe that with OT-Phasing, there is essentially no bound on the number of items that the server can map to each specific bin, since the client does not observe this value in any way (the message that the client receives only depends on the total number of items that the server has). However, the parameters used

h	Util. [%]	#OTs	#Masks	Comm. [MB]	
				$n_1 = n_2$	$n_2 \ll n_1$
2	50.0	$2.00n_2$ t	$\mathbf{2}n_1\ell$	148.0	**17.0**
3	91.8	$1.09n_2$ t	$3n_1\ell$	**99.8**	25.5
4	97.7	$1.02n_2$ t	$4n_1\ell$	105.3	34.0
5	**99.2**	$\mathbf{1.01}n_2$ **t**	$5n_1\ell$	114.6	42.5

Table 1: Overall communication for a larger number of hash functions h. Communication is given for a) $n_1 = n_2 = 2^{20}$ and b) $n_2 = 2^8 \ll n_1 = 2^{20}$ elements of $\sigma = 32$-bit length. Utilization according to [6].

in the protocol do need to ensure that the Cuckoo hashing procedure does not fail. The analysis appears in §7.

7 Hashing Failures

The PSI schemes we presented use simple hashing (by P_1), and Cuckoo hashing (by P_2). In both hashing schemes, the usage of bins (or a stash) of constant size, might result in hashing failures if the number of items mapped to a bin (or the stash) exceeds its capacity.

When hashing fails, the party which performed the hashing has two options: (1) Ignore the item that cannot be mapped by the hashing scheme. This essentially means that this item is removed from the party's input to the PSI protocol. Consequently, the output of the computation might not be correct (although, if this type of event happens rarely, the effect on correctness is likely to be marginal). (2) Attempt to use a different set of hash functions, and recompute the hash of all items. In this case the other party must be informed that new hash functions are used. This is essentially a privacy leak: for example, the other party can check if the input set S of the first party might be equal to a set S' (if a hashing failure does not occur for S' then clearly $S' \neq S$). The effect of this leak is likely to be weak, too, but it is hard to quantify.

The effect of hashing failures is likely to be marginal, and might be acceptable in many usage settings (for example, when measuring ad conversion rates it typically does not matter if the revenue from a single ad view is ignored). However, it is preferable to set the probability of hashing failures to be negligibly small.

In OT-Phasing, P_2 does not learn the number of items that P_1 maps to each bin, and therefore P_1 can set the size of the bins to be arbitrarily large. However, in that PSI protocol P_1 knows the size of the stash that is used in the Cuckoo hashing done by P_2. In Circuit-Phasing, each party knows the size of the bins (or stash) that is used by the other party. We are therefore interested in learning the failures probabilities of the following schemes, and bound them to be negligible, i.e., at most 2^{-40}:

- §7.1: Simple hashing in the Circuit-Phasing scheme, where n items are mapped using two independent functions to $2.4n$ bins. This is equivalent to mapping $2n$ items to $2.4n$ bins.
- §7.2: Cuckoo hashing, using $2.4n$ bins and either 2-way hashing (for Circuit-Phasing), or 3-way hashing (for OT-Phasing). The failure probability for 3-way hashing is smaller than for 2-way hashing (since there is an additional bin to which each item can be mapped), and therefore we will only examine the failure probability of 2-way Cuckoo hashing.

7.1 Simple Hashing

It was shown in [23] that when n balls are mapped at random to n bins then the maximum number of elements in a bin is with high probability $\frac{\ln n}{\ln \ln n}(1 + o(1))$. Let us examine in more detail the probability of the following event, "$2n$ balls are mapped at random to $2.4n$ bins, and the most occupied bin has at least k balls":

$$\begin{aligned} &\Pr(\exists \text{bin with} \geq k \text{ balls}) && (1)\\ &\leq 2.4n \cdot \Pr(\text{bin \#1 has} \geq k \text{ balls}) && (2)\\ &\leq 2.4n \binom{2n}{k} \left(\frac{1}{2.4n}\right)^k && (3)\\ &\leq \left(\frac{2ne}{k}\right)^k \left(\frac{1}{2.4n}\right)^{k-1} && (4)\\ &= n \left(\frac{2e}{k}\right)^k \left(\frac{1}{2.4}\right)^{k-1}. && (5) \end{aligned}$$

It is straightforward to see that this probability can be bounded to be at most 2^{-40} by setting

$$k \quad \geq \quad \max(6, 2e \log n / \log \log n). \qquad (6)$$

We calculated for some values of n the desired bin sizes based on the upper bound of Eq. (6) and the tighter calculation of Eq. (5), and chose the minimal value of k that reduces the failure probability to below 2^{-40}. The results are in Table 2. It is clear that Eq. (5) results in smaller bins for sufficiently large n, and therefore the maximal bin size should be set according to Eq. (5).

n	2^{12}	2^{16}	2^{20}	2^{24}
Eq. (5)	18	19	20	21
Eq. (6)	19	22	26	29

Table 2: The bin sizes max_β that are required to ensure that no overflow occurs when mapping $2n$ items to $2.4n$ bins, according to Eq. (5) and Eq. (6).

7.2 Cuckoo Hashing

It was shown in [16] that Cuckoo hashing with a stash of size s fails with probability $O(n^{-s})$. The constants in the big "O" notation are unclear, but it is obvious that $O(n^{-s}) \leq n^{-(s-1)}$ for sufficiently large values of n.

s	2^{11}	2^{12}	2^{13}	2^{14}
0	1,068,592,289	1,070,826,935	1,072,132,187	1,072,845,430
1	4,994,200	2,861,137	1,592,951	891,497
2	147,893	52,038	16,404	4,840
3	7,005	1,647	274	56
4	407	62	8	1
5	28	5	0	0
6	2	0	0	0

Table 3: Required stash sizes s accumulated over 2^{30} Cuckoo hashing repetitions mapping $n \in \{2^{11}, 2^{12}, 2^{13}, 2^{14}\}$ elements to $2.4n$ bins.

We would like to find the exact size of the stash that ensures that the failure probability is smaller than 2^{-40}. We ran 2^{30} repetitions of Cuckoo hashing, mapping n items to $2.4n$ bins, for $n \in \{2^{11}, 2^{12}, 2^{13}, 2^{14}\}$, and recorded the stash size s that was needed for Cuckoo hashing to be successful. Tab. 3 depicts the number of repetitions where we required a stash of size s. From the results we can observe that, to achieve 2^{-30} failure probability of Cuckoo hashing, we would require a stash of size $s = 6$ for $n = 2^{11}$, $s = 5$ for $n = 2^{12}$, and $s = 4$ for both $n = 2^{13}$ and $n = 2^{14}$ elements.

However, in our experiments we need the stash sizes for larger values of $n \geq 2^{14}$ to achieve a Cuckoo hashing failure probability of 2^{-40}. To obtain the failure probabilities for larger values of n, we extrapolate the results from Tab. 3 using linear regression and illustrate the results in Fig. 1. We observe that the stash size for achieving a failure probability of 2^{-40} is drastically reduced for higher values of n: for $n = 2^{16}$ we need a stash of size $s = 4$, for $n = 2^{20}$ we need $s = 3$, and for $n = 2^{24}$ we need $s = 2$. This observation is in line with the asymptotic failure probability of $O(n^{-s})$.

Finally, we extrapolate the required stash sizes s to achieve a failure probability of 2^{-40} for smaller values of $n \in \{2^8, 2^{12}\}$ and give the results together with the stash sizes for $n \in \{2^{16}, 2^{20}, 2^{24}\}$ in Tab. 4.

number of elements n	2^8	2^{12}	2^{16}	2^{20}	2^{24}
stash size s	12	6	4	3	2

Table 4: Required stash sizes s to achieve 2^{-40} error probability when mapping n elements into $2.4n$ bins.

8 Evaluation

We report on our empirical performance evaluation of Circuit-Phasing (§5) and OT-Phasing (§6) next. We evaluate their performance separately (§8.1 and §8.2), since special purpose protocols for set intersection were shown to greatly outperform circuit-based solutions in [22]. (The latter are nevertheless of independent interest because their functionality can be easily modified.)

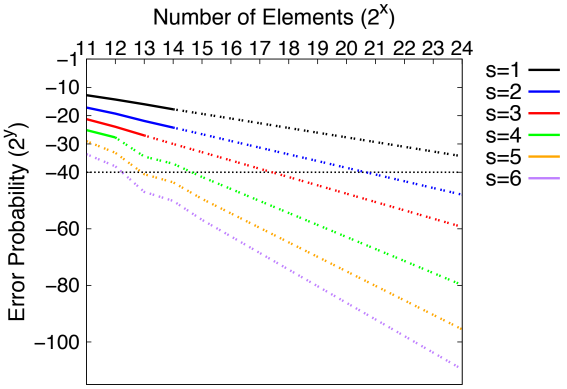


Figure 1: Error probability when mapping n elements to $2.4n$ bins using 2-way Cuckoo hashing for stash sizes $1 \leq s \leq 6$. The solid lines correspond to actual measurements, the dashed lines were extrapolated using linear regression. Both axes are in logarithmic scale.

Benchmarking Environment We consider two benchmark settings: a *LAN* setting and a *WAN* setting. The LAN setting consists of two desktop PCs (Intel Haswell i7-4770K with 3.5 GHz and 16GB RAM) connected by Gigabit LAN. The WAN setting consists of two Amazon EC2 m3.medium instances (Intel Xeon E5-2670 CPU with 2.6 GHz and 3.75 GB RAM) located in the US east coast (North Virginia) and Europe (Frankfurt) with an average bandwidth of 50 MB/s and average latency (round-trip time) of 96 ms.

We perform all experiments for a symmetric security parameter $\kappa = 128$-bit and statistical security parameter $\lambda = 40$ (cf. §2.1), using a single thread (except for GMW, where we use two threads to compute OT extension), and average the results over 10 executions. In our experiments, we frequently encountered outliers in the WAN setting with more than twice of the average run-time, for which we repeated the execution. The resulting variance decreased with increasing input set size; it was between $0.5\% - 8.0\%$ in the LAN setting and between $4\% - 16\%$ in the WAN setting. Note that all machines that we perform our experiments on are equipped with the AES-NI extensions which allows for very fast AES evaluation.

Implementation Details We instantiate the random oracle, the function for hashing into smaller domains, and the correlation-robust function in OT extension with SHA256. We instantiate the pseudo-random generator using AES-CTR and the pseudo-random permutation in the server-aided protocol of [15] using AES. To compute the $\binom{2^\mu}{1}$-OT^t_ℓ functionality, we use the random 1-

out-of-N OT extension of [17] and set $\mu = 8$, i.e., use $N = 256$, since this was shown to result in minimal overhead in [22]. We measure the times for the function evaluation including the cost for precomputing the OT extension protocol and build on the OT extension implementation of [2]. Our OT-Phasing implementation is available online at `https://github.com/encryptogroup/PSI` and our Circuit-Phasing implementation is available as part of the ABY framework of [5] at `https://github.com/encryptogroup/ABY`.

For simple hashing we use the maximum bin sizes that were computed using Equation 5 in §7.1 (cf. Tab. 2). For Cuckoo hashing, we set $\varepsilon = 0.2$ and map n elements to $2(1+\varepsilon)n$ bins for 2-way Cuckoo hashing and to $(1+\varepsilon)n$ bins for 3-way Cuckoo hashing with a stash size according to Tab. 4. The only exception for the stash size are the experiments with different set sizes in §8.2.2, where we use no stash for our OT-Phasing protocol.

For OT-based PSI [22] and OT-Phasing, where the performance depends on the bit-length of elements, we hash the σ-bit input elements into a $\ell = \lambda + \log_2(n_1) + \log_2(n_2)$-bit representation using SHA256 if $\sigma > \ell$.

We use a garbled circuits implementation with most recent optimizations (cf. full version [21] for details).

We emphasize that all implementations are done in the same programming language (C++), use the same underlying libraries for evaluating cryptographic operations (OpenSSL for symmetric cryptography and Miracl for elliptic curve cryptography), perform the plaintext-intersection of elements using a standard hash map, are all executed using a single thread (except for the GMW implementation which uses two threads), and run in the same benchmarking environment.

8.1 Generic Secure Computation-based PSI Protocols

For the generic secure computation-based PSI protocols, we perform the evaluation on a number of elements varying from 2^8 to 2^{20} and a fixed bit-length of $\sigma = 32$-bit. For $n = 2^{20}$ all implementations, except Circuit-Phasing with GMW, exceeded the available memory, which is due to the large number of AND gates in the SCS circuit (estimated 2 billion AND gates) and the requirement to represent bits as keys for Circuit-Phasing with Yao, where storing only the input wire labels to the circuit requires 1 GB. A more careful implementation, however, could allow the evaluation of these circuits. We compare the sort-compare-shuffle (SCS) circuit of [12] and its depth-optimized version of [22], with Circuit-Phasing (§5), by evaluating both constructions using Yao's garbled circuits protocol [25] and the GMW protocol [11] in the LAN and WAN setting. We use the size-optimized version of the SCS circuit in Yao's garbled circuit and the depth-optimized version of the circuit in the GMW protocol (cf. §3.4). For the evaluation in Circuit-Phasing, we set the maximum bin size in simple hashing according to Equation 5 (cf. Tab. 2, set $\varepsilon = 0.2$, set the stash size according to Tab. 4, and assume $n = n_1 = n_2$. The run-time of Circuit-Phasing would increase linear in the bin size max_β, while the stash size s would have a smaller impact on the total run-time as the concrete factors are smaller.

Run-Time (Tab. 5) Our main observation is that Circuit-Phasing outperforms the SCS circuit of [12] for all parameters except Yao's garbled circuits with small set sizes $n = 2^8$. In this case, the high stash size of $s = 12$ greatly impacts the run-time of Circuit-Phasing. When evaluated using Yao's garbled circuits, Circuit-Phasing outperforms the SCS circuit by a factor of 1-2, and when evaluated using GMW it outperforms SCS by a factor of 2-5. Furthermore, the run-time for Circuit-Phasing grows slower with n than for the SCS circuit for all settings except for GMW in the WAN setting. There, the run-time of the SCS circuit grows slower than that of Circuit-Phasing. This can be explained by the high number of communication rounds of the SCS based protocol, which are slowly being amortized with increasing values of n. The slower increase of the run-time of Circuit-Phasing with increasing n is due to the smaller increase of the bin size $max_\beta \in \mathsf{O}(\frac{\ln n}{\ln \ln n})$ vs. $\mathsf{O}(\log n)$ for the SCS circuit, and the use of permutation-based hashing, which reduces the bit-length of the inputs to the circuit. Note that our Yao's garbled circuits implementation suffers from similar performance drawbacks in the WAN setting as our GMW implementation, although being a constant round protocol. This can be explained by the pipelining optimization we implement, where the parties pipeline the garbled circuits generation and evaluation. The performance drawback could be reduced by using an implementation that uses independent threads for sending / receiving.

Communication (Tab. 6) Analogously to the run-time results, Circuit-Phasing improves the communication of the SCS circuit by factor of 1-4 and grows slower with increasing values of n. The improvement of the round complexity, which is mostly important for GMW, is even more drastic. Here, Circuit-Phasing outperforms the SCS circuit by a factor of 16-38. Note that the round complexity of Circuit-Phasing only depends on the bit-length of items and is independent of the number of elements.

8.2 Special Purpose PSI Protocols

For the special purpose PSI protocols we perform the experimental evaluation for equally sized sets $n_1 =$

Protocol	LAN				WAN			
	$n = 2^8$	$n = 2^{12}$	$n = 2^{16}$	$n = 2^{20}$	$n = 2^8$	$n = 2^{12}$	$n = 2^{16}$	$n = 2^{20}$
Yao's garbled circuits [25]								
SCS [12]	309	3,464	63,857	—	2,878	20,184	301,512	—
Circuit-Phasing §5	376	3,154	39,785	—	3,004	17,133	178,865	—
Goldreich-Micali-Wigderson [11]								
SCS [12]	626	2,175	38,727	—	11,870	21,030	218,378	—
Circuit-Phasing §5	280	1,290	14,149	168,397	2,681	8,681	81,534	846,510

Table 5: Run-time in ms for generic secure PSI protocols in the LAN and WAN setting on $\sigma = 32$-bit elements.

Protocol	$n = 2^8$	$n = 2^{12}$	$n = 2^{16}$	$n = 2^{20}$	Asymptotic
Number of AND gates					
SCS [12]	229,120	5,238,784	107,479,009	*2,000,000,000	$\sigma(3n\log_2(n)+4n)$
Circuit-Phasing §5	297,852	3,946,776	49,964,540	600,833,968	$(\sigma-\log_2(n)-2)(6(1+\varepsilon)n\frac{\ln n}{\ln\ln n}+sn)$
Communication in MB for Yao's garbled circuits [25] and GMW [11]					
SCS [12]	7	169	3,485	*64,850	$2\kappa\sigma(3n\log_2(n)+4n)$
Circuit-Phasing §5	9	122	1,550	18,736	$2\kappa(\sigma-\log_2(n)-2)(6(1+\varepsilon)n\frac{\ln n}{\ln\ln n}+sn)$
Number of communication rounds for GMW [11]					
SCS [12]	85	121	157	193	$(\log_2(\sigma)+4)\log_2(2n)+4$
Circuit-Phasing §5	5	5	5	5	$\log_2(\sigma)$

Table 6: Number of AND gates, concrete communication in MB, round complexity, and failure probability for generic secure PSI protocols on $\sigma = 32$-bit elements. Numbers with * are estimated.

n_2 (§8.2.1) and differently sized sets $n_2 \ll n_1$ (§8.2.2), for set sizes ranging from 2^8 to 2^{24} in the LAN setting and from 2^8 to 2^{20} in the WAN setting.

We compare OT-Phasing (§6) to the original OT-based PSI protocol of [22], the naive hashing solution (§3.1), the semi-honest server-aided protocol of [15] (§3.2), and the Diffie-Hellmann (DH)-based protocol of [18] (§3.3) using elliptic curves. Note that the naive hashing protocol and the server-aided protocol of [15] have different security assumptions and cannot directly be compared to the remaining protocols. We nevertheless included them in our comparison to serve as a base-line on the efficiency of PSI. For the protocol of [15], we run the server routine that computes the intersection between the sets on the machine located at the US east coast (North Virginia) and the server and client routine on the machine in Europe (Frankfurt). For the original OT-based PSI and OT-Phasing, we give the run-time and communication for three bit-lengths: *short* $\sigma = 32$ (e.g., for IPv4 addresses), *medium* $\sigma = 64$ (e.g., for credit card numbers), and *long* $\sigma = 128$ (for set intersection between arbitrary inputs).

Note that the OT-based PSI protocol of [22] and our OT-Phasing protocol both evaluate public-key cryptography during the base-OTs, which dominates the run-time for small sets. However, these base-OTs only need to be computed once and can be re-used over multiple sessions. In the LAN setting, the average run-time for computing the 256 base-OTs was 125 ms while in the WAN setting the run-time was 245 ms. Nevertheless, our results all contain the time for the base-OTs to provide an estimation of the total run-time.

8.2.1 Experiments with Equal Input Sizes

In the experiments for input sets of equal size $n = n_1 = n_2$ we set $n \in \{2^8, 2^{12}, 2^{16}, 2^{20}, 2^{24}\}$ in the LAN setting and $n \in \{2^8, 2^{12}, 2^{16}, 2^{20}\}$ in the WAN setting. Note that for larger bit-lengths $\sigma \geq 64$ and for $n = 2^{24}$ elements, the memory needed for the OT-based PSI protocol of [22] exceeded the available memory.

Run-Time (Tab. 7) As expected, the lowest run-time for the equal set-size experiments is achieved by the (insecure) naive hashing protocol followed by the server-aided protocol of [15], which has around twice the run-time. In the LAN setting, however, for short bit-length $\sigma = 32$, our OT-Phasing protocol nearly achieves the same run-time as both of these solutions (which are in a different security model). In particular, when computing the intersection for $n = 2^{24}$ elements, our OT-Phasing protocol requires only 3.5 more time than the naive hashing protocol and 2.5 more time than the server-aided protocol. In comparison, for the same parameters, the original OT-based PSI protocol of [22] has a 68 times higher run-time than the naive hashing protocol, and the DH-based ECC protocol of [18] has a four orders of magnitude higher run-time compared to naive hashing.

While the run-time of our OT-Phasing protocol increases with the bit-length of elements, for $\sigma = 128$-bit its run-time is only 15 times higher than the naive hashing protocol, and is still nearly two orders of magnitude better than the DH-based ECC protocol.

Overall, in the LAN setting and for larger sets (e.g., $n = 2^{24}$), the run time of OT-Phasing is 20x better than that of the original OT-based PSI protocol of [22], and

Setting	**LAN**					**WAN**			
Protocol	$n=2^8$	$n=2^{12}$	$n=2^{16}$	$n=2^{20}$	$n=2^{24}$	$n=2^8$	$n=2^{12}$	$n=2^{16}$	$n=2^{20}$
Naive Hashing$^{(*)}$ §3.1	1	4	48	712	13,665	97	111	558	3,538
Server-Aided$^{(*)}$ [15]	1	5	78	1,250	20,053	198	548	2,024	7,737
DH-based ECC [18]	231	3,238	51,380	818,318	13,065,904	628	10,158	161,850	2,584,212
Bit-length $\sigma = 32$*-bit*									
OT PSI [22]	184	216	3,681	62,048	929,685	957	1,820	9,556	157,332
OT-Phasing §6	179	202	437	4,260	46,631	912	1,590	3,065	14,567
Bit-length $\sigma = 64$*-bit*									
OT PSI [22]	201	485	7,302	125,697	—	977	1,873	18,998	315,115
OT-Phasing §6	180	240	865	10,128	137,036	1,010	1,780	5,009	29,387
Bit-length $\sigma = 128$*-bit*									
OT PSI [22]	201	485	8,478	155,051	—	980	1,879	21,273	392,265
OT-Phasing §6	181	240	915	13,485	204,593	1,010	1,780	5,536	37,422

Table 7: Run-time in ms for protocols with $n = n_1 = n_2$ elements. (Protocols with $^{(*)}$ are in a different security model.)

Protocol	$n=2^8$	$n=2^{12}$	$n=2^{16}$	$n=2^{20}$	$n=2^{24}$	**Asymptotic [bit]**
Naive Hashing$^{(*)}$ §3.1	0.01	0.03	0.56	10.0	176.0	$n_1\ell$
Server-Aided$^{(*)}$ [15]	0.01	0.16	2.5	40.0	640.0	$(n_1+n_2+\lvert X\cap Y\rvert)\kappa$
DH-based ECC [18]	0.02	0.28	4.56	74.0	1,200.0	$(n_1+n_2)\varphi+n_1\ell$
Bit-length $\sigma = 32$*-bit*						
OT PSI [22]	0.09	1.39	22.58	367.20	5,971.20	$0.6n_2\sigma\kappa+6n_1\ell$
OT-Phasing §6	0.06	0.73	8.74	136.8	1,494.4	$2.4n_2\kappa(\lceil\frac{\sigma-\lfloor\log_2(1.2n_2)\rfloor}{8}\rceil)+(3+s)n_1\ell$
Bit-length $\sigma = 64$*-bit*						
OT PSI [22]	0.14	2.59	41.78	674.4	10,886.4	$0.6n_2\kappa * min(\ell,\sigma)+6n_1\ell$
OT-Phasing §6	0.09	1.34	18.34	290.4	3,952.0	$2.4n_2\kappa(\lceil\frac{min(\ell,\sigma)-\log_2(n_2)}{8}\rceil)+(3+s)n_1\ell$
Bit-length $\sigma = 128$*-bit*						
OT PSI [22]	0.14	2.59	46.58	828.0	14,572.8	$0.6n_2\ell\kappa+6n_1\ell$
OT-Phasing §6	0.09	1.34	20.74	367.2	5,795.2	$2.4n_2\kappa(\lceil\frac{\ell-\log_2(n_2)}{8}\rceil)+(3+s)n_1\ell$

Table 8: Communication in MB for PSI protocols with $n = n_1 = n_2$ elements. $\ell = \lambda + \log_2(n_1) + \log_2(n_2)$. Assuming intersection of size $1/2 \cdot n$ for TTP-based protocol. (Protocols with $^{(*)}$ are in a different security model.)

60-278x better than that of the DH-ECC protocol of [18].

When switching to the WAN setting, the run-times of the protocols are all increased by a factor of 2-6. Note that the faster protocols suffer from a greater performance loss (factors of 5 and 6 for 2^{20} elements, for the naive hashing protocol and server-aided protocol) than the slower protocols (factor 3 for the DH-based and our OT-Phasing protocol and 2.5 for the OT-based PSI protocol of [22]). This difference can be explained by the greater impact of the high latency of 97 ms on the run-time of the protocols. The relative performance among the protocols remains similar to the LAN setting.

Communication (Tab. 8) The amount of communication performed during protocol execution is often more limiting than the required computation power, since the latter can be scaled up more easily by using more machines. The naive hashing approach has the lowest communication among all protocols, followed by the server-aided solution of [15]. Among the secure two-party PSI protocols, the DH-based ECC protocol of [18] has the lowest communication. In the setting for $n = 2^{24}$ elements of short bit-length $\sigma = 32$ bit, our OT-Phasing protocol nearly achieves the same complexity as the DH-based ECC protocol, which is due to the use of permutation-based hashing. This is quite surprising, as protocols that use public-key cryptography, in particular elliptic curves, were believed to have much lower communication complexity than protocols based on other cryptographic techniques.

In comparison to the original OT-based PSI protocol of [22], OT-Phasing reduces the communication for all combinations of elements and bit-lengths by factor 2.5 - 4. We also observe that OT-Phasing reduces the impact when performing PSI on elements of longer bit-length. In fact, it even has a lower communication for $\sigma = 128$ than the original OT-based PSI protocol has for $\sigma = 32$.

8.2.2 Experiments with Different Input Sizes

For examining the setting where the two parties have different input sizes, we set $n_1 \in \{2^{16}, 2^{20}, 2^{24}\}$ and $n_2 \in \{2^8, 2^{12}\}$ and run the protocols on all combinations such that $n_2 \ll n_1$. Note that we excluded the original OT-based PSI protocol of [22] from the comparison, since the bin size max_β becomes large when $\beta \ll n$ and the memory requirement when padding all bins to max_β elements quickly exceeded the available memory. In this setting, unlike the equal input sizes experiments in §8.2.1, we use $h = 2$ hash functions instead of $h = 3$,

Setting	LAN						WAN			
Protocol	$n_2 = 2^8$			$n_2 = 2^{12}$			$n_2 = 2^8$		$n_2 = 2^{12}$	
	$n_1 = 2^{16}$	$n_1 = 2^{20}$	$n_1 = 2^{24}$	$n_1 = 2^{16}$	$n_1 = 2^{20}$	$n_1 = 2^{24}$	$n_1 = 2^{16}$	$n_1 = 2^{20}$	$n_1 = 2^{16}$	$n_1 = 2^{20}$
Naive Hashing$^{(*)}$ §3.1	33	464	7,739	35	466	7,836	560	2,775	562	2,797
Server-Aided$^{(*)}$ [15]	74	680	8,935	75	696	8,965	629	2,923	731	2,951
DH-based ECC [18]	28,387	421,115	6,848,215	29,810	422,712	6,849,534	112,336	1,743,400	111,642	1,753,595
OT-Phasing §6										
Bit-length $\sigma = 32$	360	906	9,465	369	2,949	12,634	2,139	4,780	3,143	11,399
Bit-length $\sigma = 64$	555	1,506	15,789	581	6,146	22,368	3,349	6,879	3,923	20,345
Bit-length $\sigma = 128$	571	1,942	21,843	649	7,291	31,932	3,352	7,999	4,391	23,209

Table 9: Run-time in ms for PSI protocols with $n_2 \ll n_1$ elements. (Protocols with $^{(*)}$ are in a different security model.)

Protocol	$n_2 = 2^8$			$n_2 = 2^{12}$			Asymptotic [bit]
	$n_1 = 2^{16}$	$n_1 = 2^{20}$	$n_1 = 2^{24}$	$n_1 = 2^{16}$	$n_1 = 2^{20}$	$n_1 = 2^{24}$	
Naive Hashing$^{(*)}$ §3.1	0.5	8.5	144.0	0.5	9.0	152.0	$n_1\ell$
Server-Aided$^{(*)}$ [15]	1.0	16.0	256.0	1.1	16.1	256.1	$(n_1+n_2+\lvert X\cap Y\rvert)\kappa$
DH-based ECC [18]	2.5	40.5	656.0	2.7	41.1	664.1	$(n_1+n_2)\varphi+n_1\ell$
OT-Phasing §6							
Bit-length $\sigma = 32$	1.1	18.1	288.1	2.0	18.9	320.9	$4.8n_2\kappa(\lceil\frac{\sigma-\lfloor\log_2(2.4n_2)\rfloor}{8}\rceil)+2n_1\ell$
Bit-length $\sigma = 64$	1.1	18.1	288.1	3.2	20.1	322.1	$4.8n_2\kappa(\lceil\frac{\sigma-\lfloor\log_2(2.4n_2)\rfloor}{8}\rceil)+2n_1\ell$
Bit-length $\sigma = 128$	1.1	18.2	288.2	3.5	20.4	322.7	$4.8n_2\kappa(\lceil\frac{\sigma-\lfloor\log_2(2.4n_2)\rfloor}{8}\rceil)+2n_1\ell$

Table 10: Communication in MB for special purpose PSI protocols with $n_2 \ll n_1$ elements. $\ell = \lambda + \log_2(n_1) + \log_2(n_2)$. Assuming intersection of size $1/2 \cdot n_2$ for the TTP-based protocol. (Protocols with $^{(*)}$ are in a different security model.)

since this results in less total computation and communication (cf. §6.2). Since we use $h = 2$ hash functions, we also increase the number of bins from $1.2n_2$ to $2.4n_2$. Furthermore, we do not use a stash for our OT-Phasing protocol with different input sizes, since the stash would greatly increase the overall communication. However, not using a stash reveals some information on P_2's set (cf. §7). We show how to secure our protocol at a much lower cost by increasing the number of bins in the full version [21].

Run-Time (Tab. 9) Similar to the results for equal set sizes, the naive hashing protocol is the fastest protocol for all parameters. The server-aided protocol of [15] is the second fastest protocol but it scales better than the naive hashing protocol for increasing number of elements. The best scaling protocol is our OT-Phasing protocol. It achieves the same performance as the server-aided protocol for $n_2 = 2^8$, $n_1 = 2^{24}$ with short bit-length $\sigma = 32$. For $n_1 = 2^{24}$ its run-time is at most twice that of the server-aided protocol in both network settings.

When switching to the WAN setting, the run-times of all protocols are increased by a factor 4-6 while the relative performance between the protocols remains similar, analogously to the equal set size experiments.

Communication (Tab. 10) As expected, the naive hashing solution again has the lowest communication overhead. Surprisingly, our OT-Phasing protocol achieves nearly the same communication as the server-aided protocol of [15] and has only two times the communication of the naive hashing protocol for all bit-lengths. Furthermore, our OT-Phasing protocol requires a factor of 2-3 less communication than the DH-based ECC protocol of [18] for nearly all parameters. The low communication of our OT-Phasing protocol for unequal set sizes is due to the low number of OTs performed.

Acknowledgements: We thank Elaine Shi and the anonymous reviewers of USENIX Security 2015 for their helpful comments. This work was supported by the European Union's 7$^{\text{th}}$ Framework Program (FP7/2007-2013) under grant agreement n. 609611 (PRACTICE) and via a Marie Curie Career Integration Grant, by the DFG as part of project E3 within the CRC 1119 CROSSING, by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE, by the Hessian LOEWE excellence initiative within CASED, by a grant from the Israel Ministry of Science and Technology (grant 3-9094), by a Magneton grant of the Israeli Ministry of Economy, by the Israel Science Foundation (Grant No. 483/13), and by the Israeli Centers of Research Excellence (I-CORE) Program (Center No. 4/11).

References

[1] Y. Arbitman, M. Naor, and G. Segev. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *FOCS'10*, pages 787–796. IEEE, 2010.

[2] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *CCS'13*, pages 535–548. ACM, 2013.

[3] E. De Cristofaro and G. Tsudik. Practical private set intersection protocols with linear complexity. In *FC'10*, volume 6052 of *LNCS*, pages 143–159. Springer, 2010.

[4] E. De Cristofaro and G. Tsudik. Experimenting with fast private set intersection. In *TRUST'12*, volume 7344 of *LNCS*, pages 55–73. Springer, 2012.

[5] D. Demmler, T. Schneider, and M. Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS'15*. The Internet Society, 2015.

[6] M. Dietzfelbinger, A. Goerdt, M. Mitzenmacher, A. Montanari, R. Pagh, and M. Rink. Tight thresholds for cuckoo hashing via XORSAT. In *ICALP'10*, volume 6198 of *LNCS*, pages 213–225. Springer, 2010.

[7] C. Dong, L. Chen, and Z. Wen. When private set intersection meets big data: An efficient and scalable protocol. In *CCS'13*, pages 789–800. ACM, 2013.

[8] S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. *Communmunications of the ACM*, 28(6):637–647, 1985.

[9] M. J. Freedman, C. Hazay, K. Nissim, and B. Pinkas. Efficient set-intersection with simulation-based security. In *Journal of Cryptology*, pages 1–41. Springer, October 2014.

[10] M. J. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. In *EUROCRYPT'04*, volume 3027 of *LNCS*, pages 1–19. Springer, 2004.

[11] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC'87*, pages 218–229. ACM, 1987.

[12] Y. Huang, D. Evans, and J. Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS'12*. The Internet Society, 2012.

[13] B. A. Huberman, M. Franklin, and T. Hogg. Enhancing privacy and trust in electronic communities. In *EC'99*, pages 78–86. ACM, 1999.

[14] S. Jarecki and X. Liu. Efficient oblivious pseudorandom function with applications to adaptive OT and secure computation of set intersection. In *TCC'09*, volume 5444 of *LNCS*, pages 577–594. Springer, 2009.

[15] S. Kamara, P. Mohassel, M. Raykova, and S. Sadeghian. Scaling private set intersection to billion-element sets. In *FC'14*, volume 8437 of *LNCS*, pages 195–215. Springer, 2014.

[16] A. Kirsch, M. Mitzenmacher, and U. Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal of Computing*, 39(4):1543–1561, 2009.

[17] V. Kolesnikov and R. Kumaresan. Improved OT extension for transferring short secrets. In *CRYPTO'13 (2)*, volume 8043 of *LNCS*, pages 54–70. Springer, 2013.

[18] C. Meadows. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In *S&P'86*, pages 134–137. IEEE, 1986.

[19] R. Pagh and F. F. Rodler. Cuckoo hashing. In *European Symposium on Algorithms (ESA'01)*, volume 2161 of *LNCS*, pages 121–133. Springer, 2001.

[20] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[21] B. Pinkas, T. Schneider, G. Segev, and M. Zohner. Phasing: Private set intersection using permutation-based hashing. Cryptology ePrint Archive, Report 2015/634, 2015. `http://eprint.iacr.org/2015/634`.

[22] B. Pinkas, T. Schneider, and M. Zohner. Faster private set intersection based on OT extension. In *USENIX Security Symposium*, pages 797–812. USENIX, 2014.

[23] M. Raab and A. Steger. "Balls into bins" - a simple and tight analysis. In *RANDOM'98*, volume 1518 of *LNCS*, pages 159–170. Springer, 1998.

[24] X. Shaun Wang, C. Liu, K. Nayak, Y. Huang, and E. Shi. iDASH secure genome analysis competition using ObliVM. Cryptology ePrint Archive, Report 2015/191, 2015. `http://eprint.iacr.org/2015/191`.

[25] A. C. Yao. How to generate and exchange secrets. In *FOCS'86*, pages 162–167. IEEE, 1986.

Faster Secure Computation through Automatic Parallelization

Niklas Buescher
Technische Universität Darmstadt

Stefan Katzenbeisser
Technische Universität Darmstadt

Abstract

Secure two-party computation (TPC) based on Yao's garbled circuits has seen a lot of progress over the past decade. Yet, compared with generic computation, TPC is still multiple orders of magnitude slower. To improve the efficiency of secure computation based on Yao's protocol, we propose a practical parallelization scheme. Its advances over existing parallelization approaches are twofold. First, we present a compiler that detects parallelism at the source code level and automatically transforms C code into parallel circuits. Second, by switching the roles of circuit generator and evaluator between both computing parties in the semi-honest model, our scheme makes better use of computation and network resources. This *inter-party parallelization* approach leads to significant efficiency increases already on single-core hardware without compromising security. Multiple implementations illustrate the practicality of our approach. For instance, we report speed-ups of up to 2.18 on 2 cores and 4.36 on 4 cores for the example application of parallel modular exponentiation.

1 Introduction

In the thirty years since Yao's seminal paper [34], Secure Multiparty Computation (MPC) and Secure Two-Party Computation (TPC) have transitioned from purely theoretic constructions to practical tools. In TPC, two parties jointly evaluate a function f over two inputs x and y provided by the parties in such a way that each party keeps its input unknown to the other. TPC enables the construction of privacy-enhancing technologies which protect sensitive data during processing steps in untrusted environments.

Many privacy enhancing implementations use the approach of "garbled circuits" introduced by Yao in the 1980s, where f is transformed into a Boolean circuit C_f and encrypted in a special way. Beginning with the realization of the first practical implementation of Yao's protocol by Fairplay in 2004 [27], theoretical and practical advances, including Garbled-row-reduction [31], free-XOR [21], garbling from fixed-key blockciphers [5] and others have led to a significant speed-up of Yao's original protocol. Furthermore, high-level language compilers [14, 22, 23] that demonstrate the usability of Yao's protocol have been developed. Nowadays, millions of gates can be garbled on off-the-shelf hardware within seconds. Nonetheless, compared with generic computation, Yao's garbled circuits protocol is still multiple orders of magnitudes slower. Even worse, recently Zahur et al. [36] indicated that the information theoretic lower bound on the number of ciphertexts for gate-by-gate garbling techniques has been reached. Hence, further simplification of computations is unlikely.

Observing the ongoing trend towards parallel hardware, e.g., many-core architectures on a single chip, we investigate whether parallelism within Yao's protocol targeting security against semi-honest adversaries can be exploited to further enhance its performance. To the best of our knowledge, all previous parallelization efforts have focused on Yao's protocol secure against malicious adversaries, which is easily parallelizable by "design", or explored the parallelization possibilities of semi-honest Yao to only limited extent (see § 2) by using manually annotated parallism in handcrafted circuits.

Therefore, in this paper we systematically look at three different levels of automatic parallelization that have the potential to significantly speed up applications based on secure computation:

Fine-grained parallelization (FGP). As the first step, we observe that independent gates, i.e., gates that do not provide input to each other, can be garbled and evaluated in parallel. Therefore, a straight forward parallelization approach is to garble gates in parallel that are located at the same circuit depth, because these are guaranteed to be independent. We refer to this approach as *fine-grained parallelization (FGP)* and show that this approach can be efficient for circuits of suitable shape. For example, when securely computing a matrix-vector multiplication with 3 million gates, we report a speed-up of 2.4 on 4 cores and 7.5 on 16 cores. Nevertheless, the achievable speed-up heavily depends on circuit properties such as the average circuit width, which can be comparably low even for larger problems when compiling from a high-level language, as we show.

Coarse-grained parallelization (CGP). To overcome the limitations of FGP for inadequately shaped circuits, we make use of high-level circuit descriptions, such as program blocks, to automatically detect larger coherent clusters of gates that can be garbled independently. We refer to this parallelization as *coarse-grained parallelization (CGP)*. As one of our main contributions, we extend the CBMC-GC compiler of Holzer et al. [14], which translates functionalities described in ANSI-C into circuits, with the capability to detect concurrency at the source code level. This enables the compilation of parallel circuits. Hence, one large circuit is automatically divided into multiple smaller, independently executable circuits. We show that these circuits lead to more scalable and faster execution on parallel hardware. For example, the matrix-vector multiplication circuit, mentioned above, scales with a speed-up of 3.9 on 4 cores and a speed-up of 12 on 16 cores, thus, significantly outperforming FGP. Furthermore, integrating automatic detection of parallel regions into a circuit compiler gives potential users the opportunity to exploit parallelism without knowledge of the internals of Yao's garbled circuits and relieves them of writing parallel circuits.

Inter-party parallelization (IPP). Finally, we present an extension to Yao's garbled circuit protocol secure against semi-honest adversaries to balance the computation costs of both parties. In the original protocol (using the defacto standard point-and-permute optimization [4, 27]), the garbling party has to perform four times the cryptographic work than the evaluating party. Hence, assuming similar computational capabilities the overall execution time is dominated by the garbling costs. Given the identified coarse-grained parallelism, the idea of our protocol is to divide the work in a symmetric manner between both parties by switching the roles of the garbling and evaluating party to achieve better computational resource utilization without compromising security in the semi-honest model. This approach can greatly reduce the overall runtime. By combining CGP and IPP, we report a speed-up over a serial implementation of 2.14 when using 2 cores and a speed-up of 4.34 when using 4 cores for the example application of a modular exponentiation.

Summarizing our results, the performance of Yao's protocols secure against semi-honest adversaries can significantly be improved by using automatic parallelization.

Outline. Next, we discuss related work. An introduction of the used primitives and tools is given in § 3. In § 4 we discuss FGP, CGP, and present our parallel circuit compiler. Moreover, we introduce the IPP protocol in § 5. In § 6 we evaluate our approaches on practical example applications.

2 Related Work

We give a short overview on parallelization approaches for Yao's garbled circuits in the semi-honest model, before discussing solutions in the malicious model. Furthermore, we discuss parallel compilation approaches for multi-party computation.

Semi-honest model. Husted et al. [17] showed a CPU and GPU parallelization with significant speed-ups on both architectures. Their approach is based on the idea of integrating an additional encryption layer between every circuit level to enable efficient fine-grained parallelization. However, their approach significantly increases the communication costs by sending one additional ciphertext per XOR gate. Moreover, bandwidth saving optimizations, such as garbled row reduction, are incompatible. This is undesirable, as network bandwidth is already a significant bottleneck.

Barni et al. [3] proposed a parallelization scheme similar to ours, which distinguishes between fine- and coarse-grained parallelism. Their approach showed speed-ups for two example applications. However, their coarse-grained approach requires manual user interaction to annotate parallelism in handcrafted circuits. Unfortunately, their timing results are hardly comparable with other work, due to the missing implementation of concurrent circuit generation and evaluation, which is required to garble larger circuits.

Most recently, Nayak et al. [30] presented an orthogonal and complementary work to ours. Their framework *GraphSC* supports the parallel computation of graph oriented applications using RAM based secure computation. GraphSC shows very good scalability for data intensive computations. Parallelism has to be annotated manually and has to follow the Pregel [26] pattern. To exploit further parallelism within different computing nodes of GraphSC, the ideas presented in this work could be exploited.

Malicious model. The "Billion gates" framework by Kreuter et al. [23] was designed to execute large circuits on cluster architectures. The framework supports parallelization in the malicious model using message passing technologies. Frederiksen et al. [11] also addressed the malicious model, yet they targeted the GPU as execution environment. In both cases, the protocol is based upon the idea of *cut-and-choose*, which consists of multiple independent executions of Yao's protocol secure against semi-honest adversaries. This independence enables naive parallelization up to the constant number of circuits required for cut-and-choose. Unfortunately, this degree of parallelism cannot be transferred to the semi-honest setting considered in this paper.

Parallel compiler. Zhang et al. [37] presented a compiler for distributed secure computation with applications for parallelization. Their compiler converts manually annotated parallelism in an extension of C into secure implementations. Even so the compiler is targeting MPC and not TPC, it could be used as an additional front-end to the ideas presented in this work.

In summary, up to now there is no work that addresses parallelization of Yao's protocol in the semi-honest model without making compromises towards the communication costs or relying on manually annotated parallism in handcrafted circuits.

3 Preliminaries

In this section, we give a short introduction into existing tools and techniques required for our parallelization approach. First, we give a brief overview of Yao's protocol (§ 3.1). Next, we introduce the compiler CBMC-GC that transfers ANSI-C to garbled circuits (§ 3.2), followed by an introduction of the Par4all framework that detects parallelism on source code level (§ 3.3).

3.1 TPC and Yao's protocol

In the following paragraphs, we give a short introduction into Yao's TPC protocol. For a complete description, we refer the reader to the detailed work of Lindell and Pinkas [25].

Semi-honest adversary model. In this work, we use the *semi-honest* (passive) adversary model. TPC protocols secure against semi-honest adversaries ensure correctness and guarantee that the participating parties do not learn more about the other party's input than they could already derive from the observed output of the joint computation. The semi-honest model is opposed the *malicious model*, where the adversary is allowed to actively violate the protocol. TPC protocols in the semi-honest model are used for many privacy-preserving applications and are therefore interesting on their own. A discussion on example applications is given in § 5.3.

Oblivious transfers. An Oblivious transfer protocol (OT) is a protocol in which a sender transfers one of multiple messages to a receiver, but it remains oblivious which piece has been transferred. In this paper, we use 1-out-of-2 OTs, where the sender inputs two l-bit strings m_0, m_1 and the receiver inputs a bit $c \in \{0,1\}$. At the end of the protocol, the receiver obliviously receives m_c such that neither the sender learns the choice c nor the receiver learns anything about the other message m_{1-c}. In 2003 Ishai et al. [18] presented the idea of OT Extension, which significantly reduces the computational costs of OTs for most interesting applications of TPC. We use OT_l^n to denote a number n of 1-out-of-2 oblivious transfers with message bit length l.

Yao's protocol. Yao's garbled circuits protocol, proposed in the 1980s [35], is a TPC protocol secure in the semi-honest model. The protocol is executed by two parties P_A, P_B and operates on functionality descriptions in form of Boolean circuits denoted with C_f. A Boolean circuit consists of n Boolean gates, two sets of inputs wires (one for each party), and a set of output wires. A gate is described by two input wires w_l, w_r, one output wire w_o, and a Boolean function $\gamma = g(\alpha, \beta)$ mapping two input bits to one output bit. The output of each gate can be used as input to multiple subsequent gates.

During protocol execution, one party becomes the circuit generator (the garbling party), the other the circuit evaluator. The generator initializes the protocol by assigning each wire w_i in the circuit two random labels w_i^0 and w_i^1 of length κ (the security parameter) representing the respective values 0 and 1. For each gate the generator computes a *garbled truth table*. Each table consists of four encrypted entries of the output wire labels w_o^γ. These are encrypted according to the gate's Boolean functionality using the input wire labels w_l^α and w_r^β as keys. Thus, an entry in the table is encrypted as

$$E_{w_l^\alpha}(E_{w_r^\beta}(w_o^{g(\alpha,\beta)})).$$

After their creation, the garbled tables are randomly permuted and sent to the evaluator, who, so far, is unable to decrypt a single row of any garbled table due to the random choice of wire labels.

To initiate the circuit evaluation, the generator sends its input bits x in form of input wire labels to the evaluator. Moreover, the evaluator's input y is transferred via an OT_κ^m protocol with the generator being the OT sender and m being the number of input bits. After the OT step, the evaluator is in possession of the garbled circuit and one input label per input wire. With this information the evaluator is able to iteratively decrypt the circuit from input wires to output wires. Once all gates are evaluated, all output wire labels are known to the evaluator. In the last step of the protocol, the generator sends an output description table (ODT) to the evaluator, containing a mapping between output label and actual bit value. The decrypted output is then shared with the generator.

Optimizations. Yao's original protocol has seen multiple optimizations in the recent past. Most important are *pipe-lining* [15], which is nescessary for the evaluation of larger circuits and a faster online execution of Yao's protocol, *garbled-row-reduction (GRR)* [31], which reduces the number of ciphertexts that are needed to be transferred per gate, and *free-XOR* [21], which allows to evaluate linear gates (XOR/XNOR) essentially for "free" without any encryption or communication costs. Most recently, Zahur et al. [36] presented an communication optimal garbling scheme, which only requires two ciphertexts per non-linear gate while being compatible with free-XOR.

3.2 CBMC-GC

In 2012, Holzer et al. [14] presented the first compiler for a large subset of C to garbled circuits, named CBMC-GC. The compiler unrolls all loops and recursive statements present in the input program up to a given or statically determined bound. Afterwards, each statement is transformed to a Boolean formula preserving the bit-precise semantics of C. The Boolean formula is then translated into a circuit, which is optimized for Yao's garbled circuits [10].

The only difference between C code and code for TPC is a special naming convention introduced by CBMC-GC. Listing 1 shows example source code for the millionaires' problem. The shown procedure is a standard C procedure, where only the input and output variables are specifically marked as designated input of party P_A or P_B (Lines 2 and 3) or as output (Line 4). Aside from this naming convention, arbitrary C computations are allowed to produce the desired result, in this case a simple comparison (Line 5).

```
1 void millionaires() {
2  int INPUT_A_income;
3  int INPUT_B_income;
4  int OUTPUT_result = 0;
5  if (INPUT_A_income > INPUT_B_income)
6   OUTPUT_result = 1;
7 }
```

Listing 1: CBMC-GC Code for Yao's Millionaires' Problem.

3.3 Automatic source code parallelization

In 2012, Amini et al. [1] presented *Par4all*, an automatic parallelizing and optimizing compiler for C. It was developed to integrate several compilation tools into one single powerful compiler. Par4all is based on the Pips [6] source-to-source compiler infrastructure that detects parallelism and uses the POCC [32] polyhedral loop optimizer to perform memory access optimizations. Par4all is capable of producing parallel OpenMP [7], Cuda and OpenCL code. Par4all operates on any ANSI-C code as input, automatically detects parallel control flow and either annotates or exports parallel regions. Annotations are realized with the OpenMP language, parallel executable kernels for Cuda/OpenCL are exported using static code analysis techniques. In this work, we mainly build upon the OpenMP output.

4 Parallelizing of Yao's Garbled Circuits

To exploit parallelism in Yao's protocol, groups of gates that can be garbled independently need to be identified. Independent gates can be garbled in parallel by the generator, as well as evaluated in parallel by the evaluator. However, detecting independent, similar sized groups of gates is known as the NP-hard graph partitioning problem [28]. The common approach to circumvent the expensive search for an optimal solution is to use heuristics. In this section, we first discuss sequential and parallel composition of functionalities (§ 4.1) and show how circuits can be garbled in parallel (§ 4.2), before introducing the fine-grained parallelization heuristic (§ 4.3) and the coarse-grained parallelization heuristic (§ 4.4).

4.1 Parallel and sequential decomposition

Throughout this paper, we consider functionalities $f(x,y)$ with two input bit strings x, y and an output bit string o. Furthermore, we use C_f to denote the circuit that represents functionality f. We refer to a functionality f as *sequentially* decomposable into *sub functionalities* f_1 and f_2 iff $f(x,y) = f_2(f_1(x,y),x,y)$.

Moreover, we consider a functionality $f(x,y)$ as parallel decomposable into sub functionalities $f_1(x,y)$ and $f_2(x,y)$ with non-zero output bit length, if a bit string permutation σ_f exists such that $f(x,y) = \sigma_f(f_1(x,y)||f_2(x,y))$, where $||$ donates a bitwise concatenation operator. Thus, functionality f can directly be evaluated by independent evaluation of f_1 and f_2. We note that f_1 and f_2 do not necessarily have to be defined over all bits of x and y. Depending on f they could share none, some, or all input bits.

We use the operator $\diamond$ to express a parallel composition of two functionalities through the existence of a permutation σ. Thus, we write $f(x,y) = f_1(x,y,) \diamond f_2(x,y)$ if there exists a permutation σ_f such that $f(x,y) = \sigma_f(f_1(x,y)||f_2(x,y))$.

We call a parallelization of f to be *efficient* if the circuit size (i.e., number of gates) of the parallelized functionality is roughly equal to the circuit size of the sequential functionality: $size(C_f) \approx size(C_{f_1}) + size(C_{f_2})$. Due to the different garbling methods for linear and non-linear gates in Yao's protocol using the free-XOR technique, $size(C_f)$ is better measured by the number of non-linear gates. Furthermore, we refer to a parallelization as *symmetric* if sub functionalities have almost equal circuit sizes: $size(C_{f_1}) \approx size(C_{f_2})$.

Finally, we refer to functionalities that can be decomposed into a sequential and a parallel part as *mixed functionalities*. For example the functionality $f(x,y) = f_3(f_1(x,y) \diamond f_2(x,y), x, y)$ can first be decomposed sequentially in f_3 and $f_1 \diamond f_2$, where the latter part can then be further decomposed in f_1 and f_2.

Without an explicit definition, we note that all definitions can be extended from the dual case f_1 and f_2 to the general case $f_1, f_2, \dots, f_n$.

4.2 Parallel circuit creation and evaluation

A circuit that consists of annotated sequential and parallel parts can be garbled in parallel as follows. Sequential regions of a circuit can be garbled using standard techniques by iterating topologically over all gates. Once a parallel decomposable region of a circuit is reached, parallelization is applied. All independent sub circuits in every parallel region can be garbled in any order by any available thread (see Figure 1). We note that the garbling order has no impact on the security [25]. After every parallel region a synchronization between the different threads is needed to guarantee that all wire labels for the next region of the circuit are computed. Multiple subsequent parallel regions with different degrees of parallelism can be garbled, when ensuring synchronization in-between.

The circuit evaluation can be parallelized in the same manner. Sequential regions are computed sequentially, parallel regions are computed in parallel by different threads. After every parallel region a thread synchronization is required to ensure data consistency.

When using pipe-lining the garbled tables have to be transmitted in an identifiable order to ensure data consistency between generator and evaluator. We propose three different variants. First, all garbled tables can be enriched with a numbering, e.g., an index, which allows a unordered transfer to the evaluator. The evaluator is then able to reconstruct the original order based on the introduced numbering. This approach has the disadvantage of an increased communication cost. The second approach is that garbled tables are sent in a synchronized and predefined order. This approach functions without additional communication, yet can lead to an undesirable 'pulsed' communication pattern. The third approach functions by strictly separating the communication channels for every sub circuit. This can either be realized by multiplexing within the TPC framework or by exploiting the capabilities of the underlying operating system. Due to the aforementioned reasons, our implementation of a parallel framework (presented in § 6.1) builds upon the latter approach.

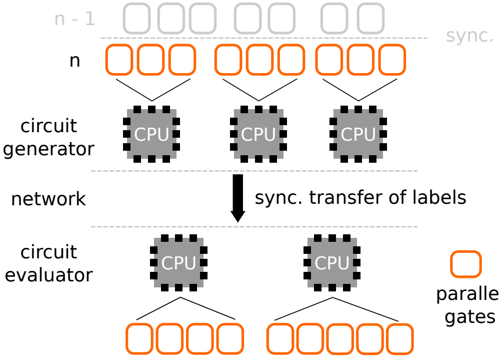


Figure 1: Interaction between a parallel circuit generator and evaluator. The layer n of the presented circuit is garbled and evaluated in parallel. The independent partitions of the circuit can be garbled and evaluated by different threads in any order.

4.3 Fine-grained parallelism

A first heuristic to decompose a circuit into parallel parts is the *fine-grained* gate level approach, described in the following. Similar to the evaluation of a standard Boolean circuit, gates in garbled circuits are processed in topological execution order. Gates provide input to other gates and hence, can be ordered by the circuit level (depth) when all their inputs are ready or the level when their output is required for succeeding gates. Consequently and as proposed by others [3, 17], gates on the same level can be garbled in parallel. Thus, a circuit is sequentially decomposable into different levels and each level is further decomposable in parallel with a granularity up to the number of gates. Figure 2 illustrates fine-grained decomposition of a circuit into three levels L1, L2 and L3.

To achieve an efficient distribution of gates onto threads during protocol execution, it is useful to identify the circuit levels during the circuit compilation process. Furthermore, a reasonable heuristic to symmetrically distribute the workload onto all threads when using the free-XOR optimization is to divide linear and non-linear gates independently. Hence, each thread gets assigned the same number of linear and non-linear gates to garble. Therefore, we extended the circuit compiler CBMC-GC with the capability to mark levels and to strictly separate linear from non-linear gates within each level. This information is stored in the circuit description.

Overhead. In practice, multi-threading introduces an computational overhead to enable thread management and thread synchronization. Therefore, it is useful to experimentally determine a system dependent threshold τ that describes the minimal number of gates that are required per level to profit from parallel execution. In practical settings (see § 6) we observe that at least ~ 8 non-

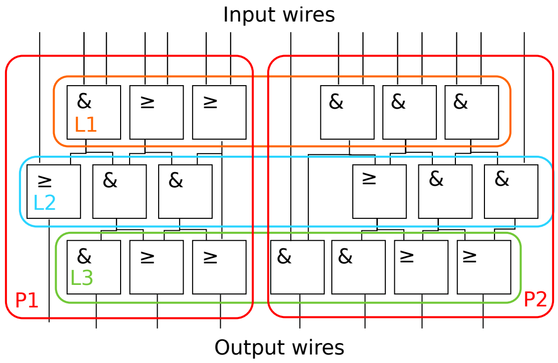


Figure 2: Circuit decomposition. Each level *L1, L2* and *L3* consists of multiple gates that can be garbled using FGP with synchronization in-between. The circuit can also be decomposed in two coarse-grained partitions *P1* and *P2*.

linear gates per core are required to observe first speedups. Achieving a parallelization efficiency of 90%, i.e, a speed up of 1.8 on 2 cores, requires at least 512 nonlinear gates per core. In the next section we present an approach that overcomes the limitations of FGP.

4.4 Coarse-grained parallelism

Another useful heuristic to partition a circuit is the usage of high-level functionality descriptions. Given a circuit description in a high-level language, parallelizable regions of the code can be identified using programming language techniques. These detected code regions can then be tracked during the circuit compilation process to produce parallel decomposable circuits. The different *sub circuits* are guaranteed to be independent of each other and therefore can be garbled in parallel. We refer to this parallelization scheme as *coarse-grained parallelization (CGP)*. Figure 2 illustrates an example decomposition in two coarse-grained partitions P1 and P2. Furthermore, we note that FGP and CGP can be combined by utilizing FGP within all coarse partations. In the following paragraphs, we introduce our CBMC-GC compiler extension that automatically produces coarse-grained parallel circuits.

Compiler for parallel circuits. Our parallel circuit complier *ParCC* extends the CBMC-GC compiler and builds on top of the Par4all compiler introduced in § 3. ParCC takes C code as input, which carries annotations according the CBMC-GC notation. Hence, TPC input and output variables of both parties are marked as such. ParCC detects parallelism within this code with the help of Par4all and produces one *global circuit* that is interrupted by one or multiple *sub circuits* for every parallel region. If a parallel region follows the single-instruction-multiple-data paradigm (SIMD), only one sub circuit per parallel region is compiled, which reduces the circuit storage costs. During protocol runtime, the sub circuit is unrolled and garbled in full extent. The global and sub circuits are interconnected by explicitly defining *inner* input and output wires. These are not exposed as TPC inputs or outputs, but have to be used by TPC frameworks to recompose the complete and parallel executable circuit. The compilation process itself consists of four different steps:

(1) In the first step, parallelism in C code is detected by Par4all and annotated using the OpenMP notation and source-to-source transformations.

(2) The annotated C code is parsed by ParCC in the second step. The source code is decomposed using source-to-source techniques into a *global* sequentially executable part, which is interrupted by one or multiple parallel executable *sub* parts. Additionally, functional OpenMP annotations, such as reduction statements, are replaced with C code that is compiled into the circuits. Furthermore, information about the degree of detected parallelism as well as the interconnection between the global and sub parts is extracted for later compilation steps.

(3) Given the decomposed source code, the different parts are compiled independently with CBMC-GC.

(4) In the final step information about the mapping of wires between gates in the global and the sub circuits is exported for use in TPC frameworks. For performance reasons, we distinguish static wires that are shared between parallel sub circuits and wires that are dedicated for each individual sub circuit.

Example. To illustrate the functionality of ParCC, we discuss the source-to-source compilation steps on a small *fork and join* task, namely the dot product of two vectors **a** and **b**:

$$r = \mathbf{a} \cdot \mathbf{b} = a_0 \cdot b_0 + \cdots + a_n \cdot b_n.$$

The source code of the function dot_product() is presented in Listing 2.

```
int mult(int a, int b) {
  return a * b;
}
void dot_product() {
  int INPUT_A_a[100], INPUT_B_b[100];
  int res = 0;
  for(i = 0; i < 100; i++)
    res += mult(INPUT_A_a[i], \
      INPUT_B_b[i]);
  int OUTPUT_res = res;
}
```

Listing 2: Dot vector product written in C with CBMC-GC input/output notation.

In this example code, two parties provide input for two vectors in form of constant length integer arrays (Line 5). A loop iterates pairwise over all array elements (Line 7), multiplies the elements and aggregates the result. In the first compilation step, Par4all detects the parallelism in the loop body and annotates this parallel region accordingly. Therefore, Par4all adds the statement `#pragma omp parallel for reduction(+:res)` before the `for` loop in Line 7.

ParCC parses the annotations in the second compilation step to export the loop body, in this case the sub function `mult()`, printed in Listing 3.

```
void mult(int INPUT_A_a, int INPUT_A_b,
    int OUTPUT_return)
{
  int a = INPUT_A_a;
  int b = INPUT_A_b;
  OUTPUT_return = a*b;
}
```

Listing 3: Exported sub function with CBMC-GC input-output notation.

The functions arguments are rewritten according the notation of CBMC-GC. Thus, the two arguments `a` and `b` of `mult()` become inner inputs of the sub circuit, and the return statement becomes an inner output variable. Note, that during the protocol execution all inner wires are not assigned to any party, instead they connect global and sub circuits. Yet, to keep compatibility with CBMC-GC a concrete assignment for the party P_A is specified. The later exported mapping information is used to distinguish between inner wires and actual input wires of both parties. In the same step, the global function `dot_product()`, printed in Listing 4, is transformed by ParCC to replace and unroll the loop.

```
1  void dot_product() {
2    int INPUT_A_a[100], INPUT_B_b[100];
3    int res = 0;
4    int OUTPUT_SUB_a[100];
5    int OUTPUT_SUB_b[100];
6    int i;
7    for(i = 0; i <= 99; i++) {
8      OUTPUT_SUB_a[i] = INPUT_A_a[i];
9      OUTPUT_SUB_b[i] = INPUT_B_b[i];
10   }
11   int INPUT_A_SUB_res[100];
12   for(i = 0; i <= 99; i++)
13     res += INPUT_A_SUB_res[i];
14   int OUTPUT_res = res;
15 }
```

Listing 4: Rewritten `dot_product()`. The loop has been replaced by inner input/output variables (marked with `SUB`).

Therefore, the two input arrays `INPUT_A_a` and `INPUT_B_b` are exposed as inner output variables beginning in Line 4. Therefore, two new output arrays using CBMC-GC notation are added based on the statically determined information about parallel variables. Furthermore, an inner input array for the intermediate results is introduced in Line 11. Finally, the reduction statement is substituted by synthesized additions over all intermediate results in Line 13.

In the third and fourth compilation step, the two circuits are compiled and the mapping of wires between global and sub circuits is exported.

5 Inter-Party Parallelization (IPP)

In this section, we describe a novel protocol extension to Yao's protocol to balance computation between parties, assuming *symmetric efficiently parallelizable* functionalities. We refer to this protocol extension as *inter-party parallelization* (IPP). Without compromising security, we show in § 6 that the protocol runtime can be reduced in practical applications when using IPP. This is also the case when using only one CPU core per party.

We recap the initial motivation: The computational costs that each party has to invest in semi-honest Yao is driven by the encryption and decryption costs of the garbled tables as well as the communication costs. Considering the garbling technique with the least number of cryptographic operations, namely GRR combined with free-XOR, the generator has to compute four ciphertexts per non-linear gate, whereas the evaluator has to compute only one ciphertext per non-linear gate. When considering the communication optimal half-gate approach [36], the generator has to compute four and the evaluator two ciphertexts per non-linear gate. Assuming two parties that are equipped with similar computational power, a better overall resource utilization would be achieved, if both parties could be equally involved in the computation process. This can be realized by sharing the roles generator and evaluator. Consequently, the overall protocol runtime could be decreased. Figure 3 illustrates this efficiency gain.

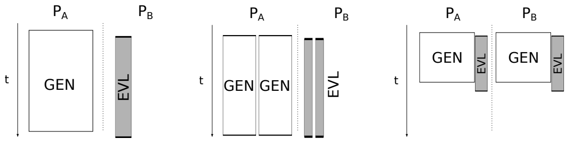
In the following sections we first discuss how to extend Yao's protocol to use IPP for purely parallel functionalities. In a second step we generalize this approach by showing how mixed functionalities profit from IPP.

5.1 Parallel functionalities

We assume that two parties P_A and P_B agree to compute a functionality $f(x,y)$ with x being P_A's input and y being P_B's input. Moreover, we assume $f(x,y)$ to be parallelizable into two (or more) sub functionalities $f_0,\dots,f_n$:

$$f(x,y) = f_0(x,y) \diamond f_1(x,y) \diamond \dots \diamond f_n(x,y).$$

Given such a decomposition, all sub functionalities can be computed independently with any TPC protocols (secure against semi-honest adversaries) without



(a) Original

(b) Independence analysis

(c) Inter-party Parallelization

Figure 3: The idea and performance gain of IPP visualized. The OT phase and output sharing are omitted. In Figure 3(a) the sequential execution of Yao's protocol is visualized. Given a parallel decomposition by two circuits representing parallal program regions as displayed in Figure 3(b), the protocol runtime can be reduced when sharing the roles generator and evaluator, as displayed in Figure 3(c).

any sacrifices towards the security [12]. This observation allows us to run two independent executions of Yao's protocol, each for one half of f's sub functionalities, instead of computing f with a single execution of Yao's protocol. Hence, P_A could garble one half of f's sub functionalities, for example $f_{even} = f_0, f_2, \dots$, and P_A could evaluate the other half $f_{odd} = f_1, f_3, \dots$. Vice versa, P_B could evaluate f_{even} and garble f_{odd}. Following this approach, P_A and P_B have to switch their roles for the OT phase of Yao's protocol. In the output phase, both parties share their output labels and description tables (ODT) with each other.

Analytical performance gain. As discussed, the computational costs for Yao's protocol are dominated by encrypting and decrypting garbled truth tables. Thus, idealizing and highly abstracted, the time spent to perform a computation t_{total} is dominated by the time to garble a circuit t_{garble}. Using GRR with free-XOR and assuming that t_{garble} is approximately four times the time to evaluate a circuit t_{eval}, by symmetrically sharing this task the total time could be reduced to:

$$t'_{total} \approx \frac{(t_{garble} + t_{eval})}{2} \approx \frac{(4 \cdot t_{eval} + t_{eval})}{2} \approx 2.5 \cdot t_{eval}.$$

This result translates to a theoretical speed-up of $t_{total}/t'_{total} = 4/2.5 = 1.6$. When using the half-gate approach the approximate computational speed-up is 1.33.

5.2 Mixed functionalities

To exploit IPP in mixed functionalities, a protocol extension is required, allowing to switch from sequential (dedicated roles) to IPP (shared roles) without violating the privacy property of TPC. Therefore, we introduce the notion of *transfering roles* to securely interchange between IPP and sequential execution.

Transferring roles We introduce the idea of transferring roles in two steps. First we sketch an insecure protocol, which is then made secure in a second step. To switch the roles of evaluator and generator during execution, we consider two parties P_A, P_B and the sequentially composed functionality $f(x,y) = f_2(f_1(x,y),x,y)$. In the following description, f_1 is computed using Yao's protocol with P_A being generator and P_B being evaluator, f_2 is computed with reversed roles.

The transfer protocol begins by computing $f_1(x,y)$ with Yao's original protocol. Once f_1 is computed, the roles have to be switched. Naïvely, the parties 'open' the circuit by interchanging output wires and the ODT. This reveals the intermediate result $o_1 = f_1(x,y)$ to both parties. In the second phase of the protocol, f_2 is computed using Yao's protocol. This time, P_A and P_B switch roles, such that P_B garbles f_2 and P_A evaluates f_2. The decrypted output bits $o_1 = f_1(x,y)$ are used by P_A as input in the OT protocol. After garbling f_2, the output labels and the ODT are shared between both parties. This protocol resembles a pause/continue pattern and preserves correctness. However, this protocol leaks o_1 to both parties, which violates the privacy requirement of TPC. Therefore, we propose to use an XOR-blinding during the role switch. The full protocol is printed below.

Protocol: Transferring Roles

P_A and P_B agree to securely compute the sequentially decomposable functionality $f(x,y) = f_2(f_1(x,y),x,y)$ without revealing the intermediate result $f_1(x,y)$ to either party, where x is P_A's input bit string, y is P_B's input bit string. The protocol consists of two phases, one per sub functionality.

Phase 1: Secure computation of $f_1(x,y)$

1. f_1 is extended with a XOR blinding for every output bit. Thus, the new output $o'_1 = f'_1(x,y||y_r) = f_1(x,y) \oplus y_r$ is calculated by xoring the output of f_1 with additional, randomly drawn input bits by the evaluator of f_1.

2. P_A and P_B securely compute f'_1 using Yao's protocol. We assume P_A to be the generator. Additional randomly drawn input bits are then input of P_B.

3. The blinded output o'_1 of the secure computation is only made visible to the generator P_A. This is realized by transmitting the output wire labels to P_A, but not sharing the ODT with P_B.

Phase 2: Secure computation of $f_2(o_1,x,y)$

1. The circuit representing f_2 is extended with a XOR unblinding for every input bit of o'_1. Hence, $f'_2(o'_1,x,y,y_r) = f_2(o'_1 \oplus y_r,x,y)$.

2. P_A and P_B securely compute f_2' using Yao's protocol. We assume P_B to be the generator. P_A provides the input o_1' and P_B provides the input bits for the blinding with y_r.

3. The output of the computation is shared with both parties.

We observe that, informally speaking the protocol preserves privacy, since the intermediate state o_1 is shared securely between both parties. A detailed formal proof on sequential decomposed functionalities is given by Hazay and Lindell [12, page 42ff]. Correctness is preserved due to blinding and unblinding with the equal bit string y_r:

$$\begin{aligned} f_2'(f_1'(x,y||y_r),x,y,y_r) &= f_2'(f_1(x,y)\oplus y_r,x,y,y_r) \\ &= f_2(f_1(x,y)\oplus y_r \oplus y_r,x,y) \\ &= f_2(f_1(x,y),x,y). \end{aligned}$$

Finally, we note that transferring roles protocol can further be improved. Demmler et al. [8] presented an approach to securely share a state of Yao's protocol that uses the point-and-permute bits [27] as a blinding. This approach has equivalent costs in the number of cryptographic operations, yet removes the need of an ODT transfer. Our implementation uses this optimization.

Transferring roles for mixed functionalities. With the idea of transferring roles, IPP can be realized for mixed functionalities. In the following paragraphs, we show how to switch from IPP to sequential computation. Switching into the other direction, namely from sequential to IPP can be realized analogously. With protocols to switch in both directions, it is possible to garble and evaluate any functionality that consists of an arbitrary number of sequential and parallel regions.

To show the switch from IPP to sequential computation, we assume a functionality that is sequentially decomposable into a parallel and a sequential functionality:

$$f(x,y) = f_3(f_1(x,y) \diamond f_2(x,y),x,y).$$

Note that f_1, f_2 and f_3 could further be composed of any sequential and parallel functionalities. We observe that f_3 can be merged with f_1 (or f_2) into one combined functionality f_c. Thus, $f(x,y)$ can also be decomposed as $f(x,y) = f_c(f_2(x,y),x,y)$ with f_c being the sequential composition of f_3 and f_1. Given such a decomposition, f_c and f_2 can be computed with alternating roles in Yao's protocol by following the transferring roles protocol. Hence, f_c could be garbled by P_A while f_2 could be garbled by P_B to securely compute f.

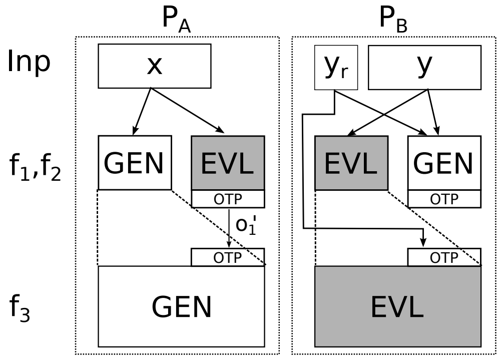


Figure 4: The IPP protocol for a mixed functionality with a switch from parallel to sequential computation. Functionality $f_1 \diamond f_2$ is garbled in parallel using IPP, f_3 is garbled sequentially in combination with f_1. No interaction between parties is shown. The blinded output o_1' of f_1 is only made visible to P_A and used as additional input for the computation of f_2 using the transferring roles protocol.

As a second observation we note that the output of f_2 is not required to start the computation of f_c. Therefore, the computation of f_c can start in parallel to the computation of f_2. This inter-party parallelism can be exploited to achieve further speed-ups. Figure 4 illustrates this approach. Party P_A garbles f_c and P_B garbles f_2. The first part of f_c, namely f_1 can be garbled in parallel to f_2. Once the blinded output o_1' of f_2 is computed, the parties can start computing the second part of f_c, namely f_3. Switching from sequential to IPP computation can be realized in the same manner.

We remark that FGP, CGP and IPP can be combined to achieve even further speed-ups. Therefore, every parallel region has first be decomposed in two parts for IPP. If the parts can further be decomposed in parallel functionalities, these could be garbled following the ideas of CGP and FGP.

Overhead. Investigating the overhead of IPP, we observe that during the cost intensive garbling and evaluation phase, no computational complexity is added. Particularly, the number of cryptographic operations and messages is left unchanged. However, when using OT Extension, a constant one-time overhead for base OTs in the size of the security parameter k is introduced to establish bi-directional OT Extension. To switch from and to IPP in mixed functionalities, additional OTs in the size of the intermediate state are required. Thus, the performance gain through IPP for mixed functionalities not only depends on the ratio between parallel and sequential regions, but also on the ratio of circuit size and shared state. These ratios are application dependent. A practical evaluation of the trade-off between overhead and performance gain is presented in § 6.5.

5.3 Security implications and applications

As discussed in the previous section, Yao's protocol and IPP are secure against semi-honest adversaries. Nevertheless, semi-honest Yao's garbled circuits are often used to bootstrap TPC protocols secure against active adversaries. Therefore, in this section, we sketch the security implications of IPP and its compatibility with the most common techniques to strengthen Yao's protocol. Furthermore, we depict applications and protocols the could profit from IPP.

Yao's original protocol is already secure against malicious evaluators (when using an OT protocol secure against malicious adversaries), yet not secure against malicious generators. We note that this one-sided security does not longer hold when using IPP because both parties incorporate the role of a circuit generator. Consequently, cut-and-choose protocols [24], which garble multiple copies of the same circuit to achieve active security, are incompatible with IPP because they are built on the assumption that only one party can actively manipulate the protocol. A similar observation can be made for Dual-execution protocols [29, 16] that prevent an active adversary from learning more than a small number of bits during the protocol execution. Even so the concept of Dual-execution is close the the idea of IPP, i.e., symmetrically sharing the roles of generator and evaluator, it also requires one-sided security against malicious evaluators and is therefore incompatible with IPP.

Applications of IPP. IPP can be applied in all application scenarios where semi-honest model is sufficient. These are scenarios where either the behaviour is otherwise restricted, e.g. limited physical access, or where the parties have sufficient trust into each other. Moreover, IPP can be used in all the scenarios where the parties inputs and seeds could be revealed at a later point to identify cheating parties. Examples might be negotiations (auctions) or games, such as online poker. Another field of application is the joint challenge creation, e.g. RSA factorization. Using secure computation, two parties could jointly create a problem instance without already knowing the solution. This allows them to create a problem and to participate in the challenge at the same time without a computational advantage. Once a solution is computed, all parts of the secure computation can be verified in hindsight.

For further work, we note that the core idea of IPP could be applied in other TPC protocols. To profit from IPP, a state sharing mechanism in the protocols security model is required as well as an asymmetric workload between the parties. One example might be the highly asymmetric STC protocol by Jarecki and Shmatikov [19] that uses zero-knowledge proofs over every gate.

6 Evaluation

In this section, we evaluate the three proposed parallelization schemes. We begin by introducing our parallel STC framework named UltraSFE and benchmark its performance on a single core in § 6.1. The applications and their circuit descriptions used for benchmarking are described in § 6.2. We evaluate the offline garbling performance of the proposed parallelization techniques in § 6.3, before integrating and evaluating the promising coarse-grained parallelization (CGP) in a online setting in § 6.4. Finally, in § 6.5 we benchmark the inter-party parallelization (IPP) approach.

6.1 UltraSFE

UltraSFE[1] is a parallel framework for Yao's garbled circuits built up on the *JustGarble (JG)* [5] garbling scheme. To realize efficient parallelization, data structures and memory layout are optimized with the purpose of parallelization in mind. This is, to the best of our knowledge, not the case with existing open source frameworks. Therefore, we adapted the JustGarble garbling scheme to support parallelization.

UltraSFE is written in C++ using SSE4, OpenMP and Pthreads to realize multi-core parallelization. Conceptually UltraSFE is using ideas from the Java based memory efficient *ME_SFE* framework [13], which itself is based on the popular *FastGC* framework [15]. The fast hardware AES extension in current CPU generations is exploited by the JustGarble garbling scheme. Oblivious transfers are realized with the help of the highly efficient and parallelized OTExtension library written by Asharov et al. [2]. Moreover, UltraSFE adopts techniques from many recent theoretical and practical advances of Yao's protocol. This includes pipe-lining, point-and-permute, garbled row reduction, free-XOR and the half-gate approach [15, 21, 27, 31, 36].

Framework comparison. To illustrate that UltraSFE is suited to evaluate the scalability of different parallelization approaches, we present a comparison of its garbling performance with other state of the art frameworks using a CPU architecture in Table 1. Namely, we compare the single core garbling speed of UltraSFE, which is practically identical to JG, with the parallel frameworks by Barni et al. (*BCPU*) [3], Husted et al. (*HCPU*) [17] and Kreuter et al. (*KSS*) [23]. Note, these results are compared in the *offline* setting, i.e., truth tables are written to memory. This is because circuit garbling is the most cost intensive part of Yao's protocol and therefore the most interesting when comparing the performance of different frameworks. The previous parallelization efforts HCPU

[1] UltraSFE will be made available as open source on http://www.seceng.informatik.tu-darmstadt.de/research/software/

	Ours / JG [5]	BCPU [3]	HCPU[17]	KSS [23]
gps	8.3M	0.11M	<0.25M	0.1M
cpg	108	>3500	-	>6500 [5]
arch	E5-2680	E5-2609	E5-2620	i7-970

Table 1: Single core garbling speed comparison of different frameworks on circuits with more than 5 million gates. Metrics are non-linear gates per second (*g*ps) in millions (M) and clocks per gate (*cpg*). All results have been observed on the Intel processor specified in row *arch*. Note, for HCPU [17] only circuit evaluation times have been reported on the CPU, the garbling speed can be assumed to be lower.

and BCPU actually abstained from implementing an *online* version of Yao's protocol that supports pipe-lining. As metrics we use garbled gates per second (*gps*) and the average number of CPU clock cycles per gate (*cpg*), as proposed in [5]. The numbers are taken from the cited publications and if not given, the cpq results are calculated based on the CPU specifications (*arch*). Even when considering theses numbers only as rough estimates, due to the different CPU types, we observe that UltraSFE performs approximately 1-2 orders of magnitude faster than existing parallelizations of Yao's protocol. This is mostly due to the efficient fixed-key garbling scheme using the AES-NI hardware extension and a carefully optimized implementation using SSE4. Summarizing, UltraSFE shows competitive garbling performance on a single core and hence, is a very promising candidate for parallelization.

6.2 Evaluation methodology

To evaluate the different parallelization approaches we use three example applications that have been used to benchmark and compare the performance of Yao's garbled circuits in the past.

Biometric matching (BioMatch). The first application that we use is privacy-preserving biometric matching. In this application a party matches one biometric sample against the other's party database of biometric templates. Example scenarios are face-recognition or fingerprint-matching [9]. One of the encompassing concepts is the computation of the Euclidean distance between the sample and all database entries. Once all distances have been computed, the minimal distance determines the best match. Thus, the task is to securely compute the minimal distance $min\left(\sum_{i=1}^{d}(s_{i,1}-e_i)^2,\ldots,\sum_{i=1}^{d}(s_{i,n}-e_i)^2\right)$ with s_i being the sample of degree d provided by the first party and $e_1,\ldots,e_n$ being the database elements with the same degree provided by the other party. Following the examples of [8, 20], the chosen parameters for this circuit are the number of elements in the database $n = 512$, the degree of each element $d = 4$ and the integer size $b = 64$bit.

	BioMatch	**MExp**	**MVMul**
Code size	22 LOC	28 LOC	10 LOC
Circuit size	66M	21.5M	3.3M
Non-linear gates	25%	41%	37%
# Input bits P_A/P_B	131K/256	1K/1K	17K/1K
Offline garbling time	2.07s	1.136s	0.154s

Table 2: *Circuit properties.* Presented are the code size, the overall circuit size in the number of gates, the fraction of non-linear gates that determine the majority of computing costs, the number of input bits as well as the sequential offline garbling time with UltraSFE.

Modular exponentiation (MExp). The second application that we benchmark is parallel modular exponentiation. Modular exponentiation has been used before to benchmark the performance of Yao's garbled circuits [5, 8, 23]. It has many applications in privacy-preserving computation. For example, blind signatures where the message to be signed should not be revealed to the signing party. For this application, we differentiate the circuit by the number of iterated executions $k = 32$, as well as the integer width $b = 32$.

Matrix-vector multiplication (MVMul). Algebraic operations such a matrix multiplication or the dot vector product are building blocks for many privacy-preserving applications and have been used before to benchmark Yao's garbled circuits [14, 22]. We use a Matrix-vector multiplication as required in the learning with errors (LWE) cryptosystem [33]. We parametrize this task according the size of the matrix $m \times k = 16 \times 16$ and vector $k = 16$, as well the integer size of each element $b = 64$ bit.

Circuit creation. All circuits are compiled twice, once with CBMC-GC and once with ParCC using textbook C implementations. The time limit for the circuit minimization through CBMC-GC is set to 10 minutes. The resulting circuits and their properties are shown in Table 2. The BioMatch circuit is the largest circuit and shows the most input bits. The MVMul circuit garbles in a fraction of a second and thus, fits to evaluated the performance of parallelelization on smaller circuits. The MExp circuit shows a large circuit complexity in comparison to the number of input bits. Even so not shown here, we note that the sequential (CBMC-GC) and parallel (ParCC) circuits slightly differ in the overall number of non-linear gates due to the circuit minimization techniques of CBMC-GC, which profit from decomposition.

Environment. As testing environment we used Amazon EC2 cloud instances. These provide a scalable number of CPUs and can be deployed at different sites around the globe. If not state otherwise, for all experiments instances of type `c3.8xlarge` have been used. These instances report 16 physical cores on 2 sockets with CPUs of type Intel Xeon E5-2680v2, and are equipped with a 10Gbps ethernet connection. A fresh installation of Ubuntu 14.04 was used to ensure as little background noise as possible. UltraSFE was compiled with `gcc 4.8 -O2` and `numactl` was utilized when benchmarking with only a fraction of the available CPUs. Numactl allows memory, core and socket binding of processes. Results have been averaged 10 executions.

Methodology. Circuit garbling is the most expensive task in Yao's protocol. Therefore, we begin by evaluating FGP and CGP for circuit garbling independent of other parts of Yao's protocol. This allows an isolated evaluation of the computational performance gains through parallelization. Following the offline circuit garbling phase is an evaluation of Yao's full protocol in an online LAN setting. This evaluation also considers the bandwidth requirements of Yao's protocol. Finally, we present an evaluation of the IPP approach in the same LAN setting. Therefore, we first evaluate the performance of IPP on a single core, before evaluating its performance in combination with CGP. The main metric in all experiments is the overall runtime and the number of non-linear gates that can be garbled per second.

6.3 Circuit garbling (offline)

We begin our evaluation of FGP and CGP with the offline task of circuit garbling. In practice the efficiency of any parallelization is driven by the ratio between computational workload per thread and synchronization between threads. When garbling a circuit with FGP, the workload is bound by the width of each level, when garbling with CGP the workload is bound by the size of parallel partitions. Both parameters are circuit and hence, application dependent.

Artificial circuits and thread utilization. To get a better insight, we first empirically evaluate the possible efficiency gain for different sized workloads, independent of any application. This also allows to observe a system dependent threshold τ, introduced in § 4.3, which describes the minimal number of gates required per thread to profit from parallelization. Therefore, we run the following experiment: For every level width $w = 2^4, 2^5, \ldots, 2^{10}$ we created artificial circuits of depth $d = 1000$. The width is kept homogeneous in all levels. Furthermore, the wiring between gates is randomized and only non-linear gates are used. Each circuit is garbled using FGP and we measured the parallelization efficiency (speed-up divided by the number of cores) when computing with a different numbers of threads. The results are illustrated in Figure 5.

The experiment shows that on the tested system $\tau \approx 8$ non-linear gates per thread are sufficient to observe first performance gains through parallelization. To achieve an efficiency of 90% approximately 512 non-linear gates per thread are required. Investigating the results for 16 parallel threads, we observe that a significantly larger workload per thread (at least one order of magnitude) is required to overcome the communication latency between the sockets on the testing hardware.

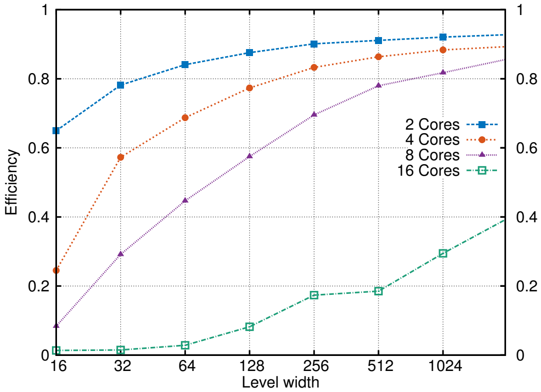


Figure 5: *Level-width experiment.* Displayed is the efficiency of FGP for different circuit level widths. A larger width increases the efficiency of parallelization. The gap between 8 (one socket) and 16 cores (two sockets) is due the communication latency between two sockets.

Example applications. We evaluated the speed-up of circuit garbling when using FGP and CGP for the three applications BioMatch, MExp and MVMul compiled with CBMC-GC (FGP) and ParCC (CGP). The speed-up is calculated in relation to the single core garbling performance given in Table 2. The results are presented in Figure 6. The results have been observed for a security level of $k = 128$ bit. We note, that in this experiment no significant differences where observable when using a smaller security level, e.g., $\kappa = 80$ bit, due to the fixed block size of AES-NI. Discussing the results for FGP, we observe that all applications profit from parallelization. BioMatch and MExp show very limited scalability, whereas the MVMul circuit is executable with a speed-up of 7.5 on 16 cores. Analyzing the performance of CGP, we observe that all applications achieve practically ideal parallelization when using up to 4 threads. In contrast to the FGP approach, scalability with high efficiency is observable with up to 8 threads. Further speed-ups when using the CPU located on the second socket are noticeable in the MExp and MVMul experiments, achieving a throughput of more than 100M non-linear gates per second.

In summary, for all presented applications the CGP approach significantly outperforms the FGP approach regarding scalability and efficiency due to its coarser granularity, which implies a better thread utilization.

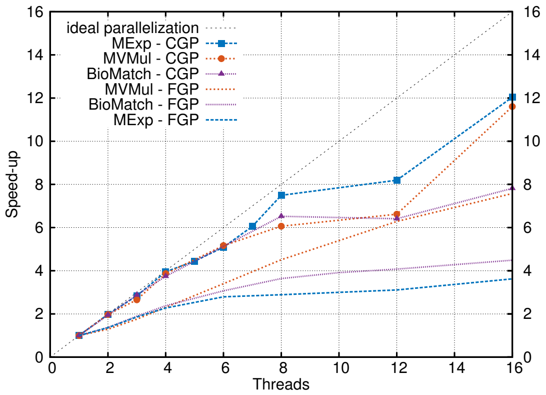


Figure 6: *Circuit garbling.* The speed-up of circuit garbling for all three applications when using the FGP, CGP and different numbers of computing threads. CGP significantly outperforms FGP for all applications.

Circuit width analysis. The limited scalability of FGP is explainable when investigating the different circuit properties. In Figure 7 the distribution of level widths for all circuits is illustrated.

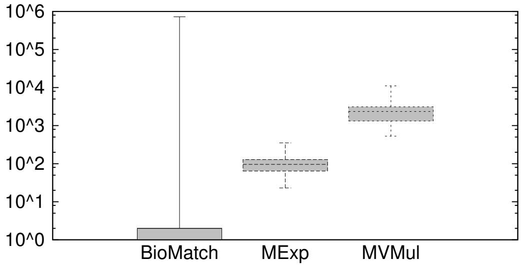


Figure 7: Number of non-linear gates per level and circuit.

For the MVMul application, the CBMC-GC compiler produces a circuit with a median level width of 2352 non-linear gates per level, whereas the BioMatch and MExp circuits only show a median width below 100 non-linear gates per level. The major reasons for small circuit widths in comparison to the overall circuit size is that high-level TPC compilers such as CBMC-GC or the compiler by Kreuter et al. [23] have been developed with a focus on minimizing the number of non-linear gates. Minimizing the circuit depth or maximizing the median circuit width barely influence the sequential runtime of Yao's protocol and is therefore not addressed in the first place. Looking at the building blocks that are used in CBMC-GC, we observe that arithmetic blocks (e.g. adder, multiplier) show a linear increase in the average circuit width when increasing the input size. However, multiplexers, as used for dynamic array accesses and for 'if' statements, show a circuit width that is independent (constant) of the number of choices. Thus, a 2-1 multiplexer and a n-1 multiplexer are compiled to circuits with similar sized levels, yet with different circuit depths. Moreover, comparisons have a constant circuit width for any input bit size. Based on these insights we deduce, that the MVMul circuit shows a significantly larger median circuit width, because of the absence of any dynamic array access, conditionals or comparisons. This is not the case with the BioMatch and MExp applications. Considering that every insufficient saturation of threads leads to an efficiency loss of parallelization, we conclude that scalability of FGP is not guaranteed when increasing input sizes.

6.4 Full protocol (online)

To motivate that the parallelization of circuit garbling can be used in Yao's full protocol, we evaluated the protocol for all applications running on two separated cloud instances in the same Amazon region (LAN setting). We observed an average round trip time of 0.6±0.3ms and a transfer bandwidth of 5.0±0.4 Gbps using `iperf`. Following the results of the offline experiments, we benchmark the more promising CGP approach in the online setting.

To measure the benefits of parallelization, we first benchmark the single core performance of Yao's protocol in the described network environment. Table 3 shows the sequential runtime for all applications using two security levels $\kappa = 80$ bit (short term) and $\kappa = 128$ bit (long term). This runtime includes the time spent on the input as well as the output phase. Furthermore, the observed throughput, measured in non-linear gates per second, as well as the required bandwidth are presented. We observe that for security levels of $\kappa = 80$ and $\kappa = 128$ a similar gate throughput is achieved. Consequently, we deduce that in this setup the available bandwidth is not stalling the computation. We also observe that that the time spent on OTs in all applications is practically negligible ($< 5\%$) in comparison with the time spent on circuit garbling.

In Figure 8 the performance gain of CGP is presented. The speed-up is measured in relation to the sequential total runtime. The timing results show that CGP scales almost linearly with up to 4 threads when using $\kappa = 80$ bit labels. Using $\kappa = 128$ bit labels, no further speed-up beyond 3 threads is noticeable. Thus, the impact of the network limits is immediately visible. Five ($\kappa = 80$ bit), respectively three ($\kappa = 128$ bit) threads are sufficient to saturate the available bandwidth in this experiment. Achieving further speed-ups is impossible without increasing the available bandwidth or developing new TPC techniques. However, to the best of our knowledge with 6M non-linear gates per second on a single core, as well as with approximately 32M non-linear gates per second on a single socket, we report the fastest garbling speed in

Circuits		BioMatch	MExp	MVMul
t_{total}	$\kappa = 128$	2.71±0.02	1.43±0.01	0.20±0.00
[s]	$\kappa = 80$	2.56±0.03	1.42±0.01	0.19±0.00
gps	$\kappa = 128$	6.23±0.04	6.17±0.05	6.22±0.00
[M]	$\kappa = 80$	6.56±0.07	6.21±0.04	6.43±0.00
bw	$\kappa = 128$	1.48±0.01	1.47±0.01	1.48±0.00
[Gbps]	$\kappa = 80$	0.97±0.01	0.92±0.01	0.95±0.00
t_{input}	$\kappa = 128$	<0.02s	<0.01s	< 0.01s
[s]	$\kappa = 80$	<0.02s	<0.01s	< 0.01s

Table 3: *Yao's protocol, single-core performance.* The runtime (t_{total}), non-linear gate throughput in million gates per second (*gps*), required bandwidth (*bw*) and time spent in the input phase (t_{input}), including the OTs when executing Yao's protocol for all applications in a LAN setting.

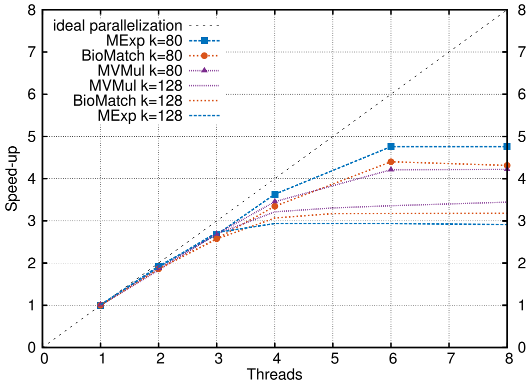


Figure 8: *Yao's protocol - CGP* The speed-up of all three applications in the LAN setting with $\kappa = 128$ bit and $\kappa = 80$ bit security.

an online setting of Yao's protocol. We abstain from an evaluation in a WAN setting due to the high bandwidth that is required to show the scalability of parallelization. The best bandwidth that we could observe during our experiments between two cloud regions was 350 Mbps, which is insufficient to benchmark parallel scalability.

6.5 Inter-party parallelization

A new application of parallelization in Yao's protocol is presented in § 5. We performed two experiments to show the applicability of IPP in practical settings. The first experiment measures the computational efficiency gain in the same setting as described in § 6.4. In the second experiment the benefits of IPP in a WAN setting with limited bandwidth are presented.

Computational efficiency gain. In this experiment the raw IPP performance for all example applications, as well as the combination of CGP and IPP techniques is explored. To realize IPP, our implementation uses multiple threads per core to utilize the load balancing capabilities of the underlying OS without implementing a sophisticated load balancer. Due to the heterogeneous hardware environment, e.g. unpredictable caching and networking behaviour, we evaluated three different workload distribution strategies. The first strategy uses one thread per core and thus only functions with at least two cores. Then, each party has exactly one garbling and one evaluating thread. The second and third strategy use two or four independent threads per core to garble and evaluate at the same time. Moreover, to illustrate that IPP is a modular concept, all circuits are evaluated using a sequential code block that exposes all inner input and output wires before and after every parallel region. This guarantees the evaluation of mixed functionalities. Consequently, all results include the time spent on transferring all required input bits to and from parallel regions. Otherwise applications such as the MVMul application, which is a pure parallel functionality, would profit more easily from IPP. Even though this weakens the results for the example applications, we are convinced that this procedure provides a better insight into the practical performance of IPP.

The results of this experiment are reported in Table 4. We first observe that only the MExp application significantly profits from IPP. This is due to the small sharing state in comparison to the circuit complexity. For both security levels IPP outperforms the raw CGP approach with an additional speed-up of 10-30% on all cores. The performance of the MVMul applications actually decreases when using IPP. This is because of the large state that needs to be transferred. The performance gain through IPP cannot overcome the newly introduced overhead of 31ms, which is more than 15% of the sequential run-time.

In summary, parallelizable applications that show a small switching surface (measured in number of bits compared to the overall circuit size) profit from IPP. Thus, IPP is a promising extension to Yao's protocol that utilizes circuit decomposition beyond naive parallelization, independently of other optimization techniques.

Bi-directional bandwidth exploitation. The second experiment aims towards increasing the available bandwidth by exploiting bidirectional data transfers. Commonly, Ethernet connections have support for full duplex (bi-directional) communication. When using standard Yao's garbled circuits, only one communication direction is fully utilized. However, with IPP the available bandwidth can be doubled by symmetrically exploiting both communication channels. This practical insight is evaluated in a WAN setting between two cloud instances of type `m3.xlarge` with $100 \pm 10ms$ latency and a measured bandwidth of 92 ± 27Mbps. Each hosts runs two threads (a garbling and a evaluating thread) using only

		$\kappa = 80$						$\kappa = 128$					
Environment		**BioMatch**		**MExp**		**MVMul**		**BioMatch**		**MExp**		**MVMul**	
Cores	**IPP**	time[s]	S	time[s]	S	time[s]	S	time[s]	S	time[s]	S	time[s]	S
1	none	2.559	1.000	1.423	1.000	0.192	1.000	2.712	1.000	1.485	1.000	0.199	1.000
	2	2.624	0.975	**1.287**	**1.106**	0.206	0.932	2.781	0.975	**1.370**	**1.084**	0.215	0.926
	4	2.556	1.001	**1.285**	**1.107**	0.208	0.923	2.707	1.002	**1.386**	**1.071**	0.218	0.913
2	none	1.384	1.849	0.734	1.939	0.103	1.864	1.497	1.812	0.780	1.904	0.108	1.844
	1	1.524	1.679	**0.686**	**2.074**	0.126	1.524	1.535	1.767	**0.699**	**2.124**	0.134	1.485
	2	1.472	1.738	**0.699**	**2.036**	0.132	1.455	1.516	1.789	**0.726**	**2.045**	0.137	1.453
	4	1.396	1.833	**0.654**	**2.176**	0.124	1.548	1.465	1.851	**0.693**	**2.143**	0.131	1.519
4	none	0.795	3.219	0.395	3.603	0.057	3.368	0.937	2.894	0.450	3.300	0.064	3.088
	1	0.996	2.569	0.426	3.340	0.084	2.286	1.041	2.605	0.452	3.285	0.087	2.287
	2	0.830	3.083	**0.336**	**4.235**	0.085	2.259	**0.874**	**3.103**	**0.356**	**4.171**	0.088	2.261
	4	0.818	3.128	**0.329**	**4.325**	0.081	2.370	**0.856**	**3.168**	**0.341**	**4.355**	0.084	2.369
8	none	0.652	3.925	0.298	4.775	0.045	4.267	0.872	3.110	0.364	4.080	0.048	4.189
	1	0.676	3.786	**0.239**	**5.954**	0.072	2.667	0.947	2.864	**0.303**	**4.901**	0.080	2.488
	2	**0.629**	**4.068**	**0.204**	**6.975**	0.077	2.494	**0.861**	**3.150**	0.342	**4.342**	0.075	2.653
	4	**0.636**	**4.024**	**0.233**	**6.107**	0.070	2.743	0.871	3.114	**0.337**	**4.407**	0.073	2.726
Transferring roles		0.231s		0.076s		0.031s		0.257s		0.082s		0.031s	

Table 4: Evaluation of IPP in a LAN setting. Column *IPP* specifies the number of threads used per core for load balancing. The total protocol run-time is measured in seconds and the speed-up in comparison with CGP is presented in column *S*. Marked in bold are settings, where IPP leads to performance gains. The time spent on the transferring roles protocol is presented in the last row.

a single core. The results of this experiment are illustrated in Table 5. IPP leads to significant speed-ups of BioMatch and MExp, showing the successful exploitation of bi-directional data transfers. MVMul shows limited performance gains because the time spent on the newly introduced communication rounds for the transferring roles protocol becomes significant. Summarizing, IPP can be very useful for TPC in bandwidth limited environments.

		BioMatch	**MExp**	**MVMul**
$\kappa = 128$	raw	45.02±0.49s	24.13±0.21s	4.83±0.05s
	IPP	29.94±0.31s	16.05±0.12s	4.66 ±0.35s
	S	1.50	1.50	1.03
$\kappa = 80$	raw	30.34±0.62s	14.56±0.21s	4.31±0.23s
	IPP	19.13±0.47s	11.16±0.32s	3.84±0.12s
	S	1.58	1.30	1.12

Table 5: Evaluation of IPP on a single core with limitted networking capabilities. Measured is the total protocol runtime, when sequentially (*raw*) computing and with IPP (*IPP*). Furthermore, the speed-up (*S*) between the two measurements is calculated.

7 Conclusion and Future Work

TPC based on Yao's garbled circuits protocol can greatly benefit from automatic parallelization. The FGP approach can be efficient for some circuits, yet its scalability highly depends on the circuit's width. The CGP approach shows a more efficient parallelization, given parallel decomposable applications. In contrast to previous work, a complete compile chain, which takes C code as input and automatically compiles parallel circuits, supports the practicability of our parallelization scheme. Moreover, we proposed the idea of IPP to achieve a symmetric workload distribution between two computing parties. With this technique, IPP achieves speed-ups through parallelization, even when using a single physical core. Concluding, in this work we presented an efficient, versatile and practical parallelization scheme for Yao's garbled circuits.

Further work includes the investigation of different parallel compilation targets for ParCC, such as the GMW protocol or RAM based secure computation frameworks. Also worthwhile for future investigations is the compilation of circuits optimized for FGP. Likewise, the application of IPP to other protocols is of interest.

8 Acknowledgements

We thank David Evans and all anonymous reviewers for their very helpful and constructive comments. This work has been co-funded by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE, by the DFG as part of project S5 within the CRC 1119 CROSSING, and by the Hessian LOEWE excellence initiative within CASED.

References

[1] Amini, M., Creusillet, B., Even, S., Keryell, R., Goubier, O., Guelton, S., McMahon, J. O., Pasquier, F.-X., Péan, G., and Villalon, P. Par4All: From Convex Array Regions to Heterogeneous Computing. In *Workshop on Polyhedral Compilation Techniques* (2012).

[2] Asharov, G., Lindell, Y., Schneider, T., and Zohner, M. More efficient oblivious transfer and extensions for faster secure computation. In *ACM Conference on Computer and Communications Security CCS* (2013).

[3] Barni, M., Bernaschi, M., Lazzeretti, R., Pignata, T., and Sabellico, A. Parallel Implementation of GC-Based MPC Protocols in the Semi-Honest Setting. In *Data Privacy Management and Autonomous Spontaneous Security*. 2014.

[4] Beaver, D., Micali, S., and Rogaway, P. The Round Complexity of Secure Protocols. In *ACM Symposium on Theory of Computing STOC* (1990).

[5] Bellare, M., Hoang, V. T., Keelveedhi, S., and Rogaway, P. Efficient garbling from a fixed-key blockcipher. In *IEEE Symposium on Security and Privacy S&P* (2013).

[6] Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P. A practical automatic polyhedral parallelizer and locality optimizer. *ACM SIGPLAN Notices 43*, 6 (2008).

[7] Dagum, L., and Menon, R. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering 5*, 1 (1998).

[8] Demmler, D., Schneider, T., and Zohner, M. ABY A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. *Network and Distributed System Security NDSS* (2015).

[9] Erkin, Z., Franz, M., Guajardo, J., Katzenbeisser, S., Lagendijk, I., and Toft, T. Privacy-preserving face recognition. In *Privacy Enhancing Technologies PETS* (2009).

[10] Franz, M., Holzer, A., Katzenbeisser, S., Schallhart, C., and Veith, H. CBMC-GC: An ANSI C compiler for secure two-party computations. In *Compiler Construction CC* (2014).

[11] Frederiksen, T. K., Jakobsen, T. P., and Nielsen, J. B. Faster maliciously secure two-party computation using the GPU. In *Security and Cryptography for Networks SCN*. 2014.

[12] Hazay, C., and Lindell, Y. Efficient secure two-party protocols. *Information Security and Cryptography. Springer, Heidelberg* (2010).

[13] Henecka, W., and Schneider, T. Faster secure two-party computation with less memory. In *ACM Conference on Computer and Communications Security ASIACCS* (2013).

[14] Holzer, A., Franz, M., Katzenbeisser, S., and Veith, H. Secure Two-Party Computations in ANSI C. In *ACM Conference on Computer and Communications Security CCS* (2012).

[15] Huang, Y., Evans, D., Katz, J., and Malka, L. Faster Secure Two-Party Computation Using Garbled Circuits. In *USENIX Security Symposium* (2011).

[16] Huang, Y., Katz, J., and Evans, D. Quid-pro-quo-tocols: Strengthening semi-honest protocols with dual execution. In *IEEE Symposium on Security and Privacy S&P* (2012).

[17] Husted, N., Myers, S., Shelat, A., and Grubbs, P. GPU and CPU parallelization of honest-but-curious secure two-party computation. In *Annual Computer Security Applications Conference ACSAC* (2013).

[18] Ishai, Y., Kilian, J., Nissim, K., and Petrank, E. Extending Oblivious Transfer Efficiently. In *Advances in Cryptology CRYPTO*. Springer, 2003.

[19] Jarecki, S., and Shmatikov, V. Efficient Two-Party Secure Computation on Committed Inputs. In *Advances in Cryptology EUROCRYPT*, vol. 4515. Springer, 2007.

[20] Kerschbaum, F., Schneider, T., and Schröpfer, A. Automatic protocol selection in secure two-party computations. In *Applied Cryptography and Network Security ACNS* (2014).

[21] Kolesnikov, V., and Schneider, T. Improved garbled circuit: Free XOR gates and applications. In *International Conference on Automata, Languages and Programming ICALP*. 2008.

[22] Kreuter, B., Mood, B., Shelat, A., and Butler, K. PCF: A Portable Circuit Format for Scalable Two-Party Secure Computation. In *USENIX Security Symposium* (2013).

[23] Kreuter, B., Shelat, A., and Shen, C. Billion-Gate Secure Computation with Malicious Adversaries. *USENIX Security Symposium* (2012).

[24] Lindell, Y., and Pinkas, B. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Advances in Cryptology EUROCRYPT*. 2007.

[25] Lindell, Y., and Pinkas, B. A proof of security of yao's protocol for two-party computation. *Journal of Cryptology 22*, 2 (2009).

[26] Malewicz, G., Austern, M. H., Bik, A. J. C., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. Pregel: a system for large-scale graph processing. In *ACM Conference on Management of Data SIGMOD* (2010).

[27] Malkhi, D., Nisan, N., Pinkas, B., and Sella, Y. Fairplay-Secure Two-Party Computation System. In *USENIX Security Symposium* (2004).

[28] McCreary, C., and Gill, H. Efficient Exploitation of Concurrency Using Graph Decomposition. In *International Conference on Parallel Processing ICPP* (1990).

[29] Mohassel, P., and Franklin, M. Efficiency tradeoffs for malicious two-party computation. In *Public Key Cryptography PKC*. 2006.

[30] Nayak, K., Wang, X. S., Ioannidis, S., Weinsberg, U., Taft, N., and Shi, E. GraphSC: Parallel Secure Computation Made Easy. In *IEEE Symposium on Security and Privacy S&P* (2015).

[31] Pinkas, B., Schneider, T., Smart, N. P., and Williams, S. C. Secure two-party computation is practical. *Advances in Cryptology ASIACRYPT* (2009).

[32] Pouchet, L.-N. Polyhedral Compiler Collection (PoCC), 2012.

[33] Regev, O. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM 56*, 6 (2009).

[34] Yao, A. C. Protocols for secure computations. In *Symposium on Foundations of Computer Science SFCS* (1982).

[35] Yao, A. C. How to generate and exchange secrets. In *Symposium on Foundations of Computer Science SFCS* (1986).

[36] Zahur, S., Rosulek, M., and Evans, D. Two halves make a whole. *Advances in Cryptology - EUROCRYPT* (2015).

[37] Zhang, Y., Steele, A., and Blanton, M. PICCO: A general-purpose compiler for private distributed computation. In *ACM Conference on Computer and Communications Security CCS* (2013).

The Pythia PRF Service

Adam Everspaugh[⋆], *Rahul Chatterjee*[⋆], *Samuel Scott*[⋆⋆], *Ari Juels*[†], *and Thomas Ristenpart*[‡]
[⋆]University of Wisconsin–Madison, {ace,rchat}@cs.wisc.edu
[⋆⋆]Royal Holloway, University of London, sam.scott.2012@live.rhul.ac.uk
[†]Jacobs Institute, Cornell Tech, juels@cornell.edu
[‡]Cornell Tech, ristenpart@cornell.edu

Abstract

Conventional cryptographic services such as hardware-security modules and software-based key-management systems offer the ability to apply a pseudorandom function (PRF) such as HMAC to inputs of a client's choosing. These services are used, for example, to harden stored password hashes against offline brute-force attacks.

We propose a modern PRF service called PYTHIA designed to offer a level of flexibility, security, and ease-of-deployability lacking in prior approaches. The keystone of PYTHIA is a new cryptographic primitive called a *verifiable partially-oblivious PRF* that reveals a portion of an input message to the service but hides the rest. We give a construction that additionally supports efficient bulk rotation of previously obtained PRF values to new keys. Performance measurements show that our construction, which relies on bilinear pairings and zero-knowledge proofs, is highly practical. We also give accompanying formal definitions and proofs of security.

We implement PYTHIA as a multi-tenant, scalable PRF service that can scale up to hundreds of millions of distinct client applications on commodity systems. In our prototype implementation, query latencies are 15 ms in local-area settings and throughput is within a factor of two of a standard HTTPS server. We further report on implementations of two applications using PYTHIA, showing how to bring its security benefits to a new enterprise password storage system and a new brainwallet system for Bitcoin.

1 Introduction

Security improves in a number of settings when applications can make use of a cryptographic key stored on a remote system. As an important example, consider the compromise of enterprise password databases. Best practice dictates that passwords be hashed and salted before storage, but attackers can still mount highly effective brute-force cracking attacks against stolen databases.

Well-resourced enterprises such as Facebook [38] have therefore incorporated remote cryptographic operations to harden password databases. Before a password is stored or verified, it is sent to a *PRF service* external to the database. The PRF service applies a cryptographic function such as HMAC to client-selected inputs under a service-held secret key. Barring compromise of the PRF service, its use ensures that stolen password hashes (due to web server compromise) cannot be cracked using an offline brute-force attack: an attacker must query the PRF service from a compromised server for each password guess. Such online cracking attempts can be monitored for anomalous volumes or patterns of access and throttled as needed.

While PRF services offer compelling security improvements, they are not without problems. Even large organizations can implement them incorrectly. For example, Adobe hardened passwords using 3DES but in ECB mode instead of CBC-MAC (or another secure PRF construction) [23], a poor choice that resulted in disclosure of many of its customers' passwords after a breach. Perhaps more fundamental is that existing PRF services do not offer graceful remediation if a compromise is detected by a client. Ideally it should be possible to cryptographically erase (i.e., render useless via key deletion) any PRF values previously used by the client, without requiring action by end users and without affecting other clients. In general, PRF services are so inaccessible and cumbersome today that their use is unfortunately rare.

In this paper, we present a next-generation PRF service called PYTHIA to democratize cryptographic hardening. PYTHIA can be deployed within an enterprise to solve the issues mentioned above, but also as a public, multi-tenant web service suitable for use by any type of organization or even individuals. PYTHIA offers several security features absent in today's conventional PRF services that are critical to achieving the scaling and flexibility required to simultaneously support a variety of clients and applications. As we now explain, achieving these features necessitated innovations in both cryptographic primitive design and system architecture.

Key features and challenges. We refer to an entity using PYTHIA as a *client*. For example, a client might be a web server that performs password-based authentication for all of its end users. Intuitively, PYTHIA allows such a client to query the service and obtain the PRF output $Y = F_k(t, m)$ for a message m and a tweak t of the client's choosing under a client-specific secret key k held by the service. Here, the tweak t is typically a unique identifier for an end user (e.g., a random salt). In our running password storage example, the web server stores Y in a database to authenticate subsequent logins.

PYTHIA offers security features that at, first glance, sound mutually exclusive. First, PYTHIA achieves message privacy for m while requiring clients to reveal t to the server. Message privacy ensures that the PRF service obtains no information about the message m; in our password-storage example, m is a user's password. At the same time, though, by revealing t to the PRF service, the service can perform fine-grained monitoring of related requests: a high volume or otherwise anomalous pattern of queries on the same t would in our running example be indicative of an ongoing brute-force attack and might trigger throttling by the PRF service.

By using a unique secret key k for each client, PYTHIA supports individual key rotation should the value Y be stolen (or feared to be stolen). With traditional PRF services and password storage, such key rotation is a headache, and in many settings impractical, because it requires transitioning stored values $Y_1, \ldots, Y_n$ (one for each user account) to a new PRF key. The only way to do so previously was to have all n users re-enter or reset their passwords. In contrast, the new primitive employed for F_k in PYTHIA supports fast key rotation: the server can erase k, replace it with a new key k', and issue a compact (constant-sized) token with which the client can quickly update all of its PRF outputs. This feature also enables forward-security in the sense that the client can proactively rotate k without disrupting its operation.

PYTHIA provides other features as well, but we defer their discussion to Section 2. Already, those listed above surface some of the challenging cryptographic tensions that PYTHIA resolves. For example, the most obvious primitive on which to base PYTHIA is an oblivious PRF (OPRF) [27], which provides message privacy. But for rate-limiting, PYTHIA requires clients to reveal t, and existing OPRFs cannot hide only a portion of a PRF input. Additionally, the most efficient OPRFs (c.f., [28]) are not amenable to key rotation. We discuss at length other related concepts (of which there are many) in Section 7.

Partially-oblivious PRFs. We introduce *partially oblivious PRFs* (PO-PRFs) to rectify the above tension between fine-grained key management and bulk key management and achieve a primitive that supports batch key rotation. We give a PO-PRF protocol in the random oracle model (ROM) similar to the core of the identity-based non-interactive key exchange protocol of Sakai, Ohgishi, and Kasahara [44]. This same construction was also considered as a left-or-right constrained PRF by Boneh and Waters [13]. That said, the functionality achieved by our PO-PRF is distinct from these prior works and new security analyses are required. Despite relying on pairings, we show that the full primitive is fast even in our prototype implementation.

In addition to a lack of well-matched cryptographic primitives, we find no supporting formal definitions that can be adapted for verifiable PO-PRFs. (Briefly, previous definitions and proofs for fast OPRFs rely on hashing in the ROM before outputting a value [16, 28]; in our setting, hashing breaks key rotation.) We propose a new assumption (a one-more bilinear decisional Diffie-Hellman assumption), give suitable security definitions, and prove the security of the core primitive in PYTHIA under these definitions (in the appendix; complete results in [25]). Our new definitions and technical approaches may be of independent interest.

Using PYTHIA in applications. We implement PYTHIA and show that it offers highly practical performance on Amazon EC2 instances. Our experiments demonstrate that PYTHIA is practical to deploy using off-the-shelf components, with combined computation cost of client and server under 12 milliseconds. A single 8-core virtualized server can comfortably support over 1,000 requests per second, which is already within a factor of two of a standard HTTPS server in the same configuration. (Our PYTHIA implementation performs all communication over TLS.) We discuss scaling to handle more traffic volume in the body; it is straightforward given current techniques.

We demonstrate the benefits and practicality of PYTHIA for use in a diverse set of applications. First is our running example above: we build a new password-database system using a password "onion" that combines parallelized calls to PYTHIA and a conventional key hashing mechanism. Our onion supports PYTHIA key rotation, hides back-end latency to PYTHIA during logins (which is particularly important when accessing PYTHIA as a remote third-party service), and achieves high security in a number of compromise scenarios.

Finally, we show that PYTHIA provides valuable features for different settings apart from enterprise password storage. We implement a client that hardens a type

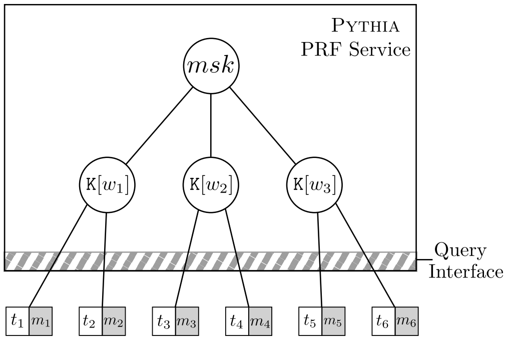


Figure 1: Diagram of PRF derivations enabled by Pythia. Everything inside the large box is operated by the server, which only learns tweaks and not the shaded messages.

of password-protected virtual-currency account called a "brainwallet" [14]; use of Pythia here prevents offline brute-force attacks of the type that have been common in Bitcoin.

Our prototype implementation of Pythia is built with open-source components and itself is open-source. We have also released Amazon EC2 images to allow companies, individuals, and researchers to spin-up Pythia instances for experimentation.

2 Overview and Challenges

We now give a high-level overview of Pythia, the motivations for its features, what prior approaches we investigated, and the threat models we assume. First we fix some terminology and a high-level conceptual view of what a PRF service would ideally provide. The service is provisioned with a master secret key msk. This will be used to build a tree that represents derived sub-keys and, finally, output values. See Figure 1, which depicts an example derivation tree associated with Pythia as well as which portions of the tree are held by the server (within the large box) and which are held by the client (the leaves). Keys of various kinds are denoted by circles and inputs by squares.

From the msk we derive a number of *ensemble keys*. Each ensemble key is used by a client for a set of related PRF invocations — the ensemble keys give rise to isolated PRF instances. We label each ensemble key in the diagram by $\texttt{K}[w]$. Here w indicates a client-chosen *ensemble selector*. An *ensemble pre-key* $\texttt{K}[w]$ is a large random value chosen and held by the server. Together, msk and $\texttt{K}[w]$ are used to derive the ensemble key $k_w = \text{HMAC}(msk, \texttt{K}[w])$. A table is necessary to support cryptographic erasure of (or updates to) individual ensemble keys, which amounts to deleting (or updating) a table entry.

Each ensemble key can be used to obtain PRF outputs of the form $F_{k_w}(t, m)$ where F is a (to-be-defined) PRF keyed by k_w, and the input is split into two parts. We call t a *tweak* following [30] and m the message. Looking ahead t will be made public to Pythia while m will be private. This is indicated by the shading of the PRF output boxes in the figure.

Deployment scenarios. To motivate our design choices and security goals, we relay several envisioned deployment scenarios for Pythia.

Enterprise deployment: A single enterprise can deploy Pythia internally, giving query access only to other systems they control. A typical setup is that Pythia fields queries from web servers and other public-facing systems that are, unfortunately, at high risk of compromise. PRF queries to Pythia harden values stored on these vulnerable servers. This is particularly suited to storing check-values for passwords or other low-entropy authentication tokens, where one can store $F_{k_w}(t, m)$ where t is a randomly chosen, per-user identifier (a salt) and m is the low-entropy password or authentication token. Here w can be distinct for each server using Pythia.

Public cloud service: A public cloud such as Amazon EC2, Google Compute Engine, or Microsoft Azure can deploy Pythia as an internal, multi-tenant service for their customers. Multi-tenant here means that different customers query the same Pythia service, and the cloud provider manages the service, ensemble pre-key table, etc. This enables smaller organizations to obtain the benefits of using Pythia for other cloud properties (e.g., web servers running on virtual machine instances) while leaving management of Pythia itself to experts.

Public Internet service: One can take the public cloud service deployment to the extreme and run Pythia instances that can be used from anywhere on the Internet. This raises additional performance concerns, as one cannot rely on fast intra-datacenter network latencies (sub-millisecond) but rather on wide-area latencies (tens of milliseconds). The benefit is that Pythia could then be used by arbitrary web clients, for example we will explore this scenario in the context of hardening brainwallets via Pythia.

One could tailor a PRF service to each of these settings, however it is better to design a single, application-agnostic service that supports all of these settings simultaneously. A single design permits reuse of open-source implementations; standardized, secure-by-default configurations; and simplifies the landscape of PRF services.

Security and functionality goals. Providing a single suitable design requires balancing a number of security and functionality goals. The most obvious requirements are for a service that: provides low-latency protocols (i.e., single round-trip and amenable for implementation as simple web interfaces); scales to hundreds of millions of ensembles; and produces outputs indistinguishable from random values even when adversaries can query the service. To this list of basic requirements we add:

- *Message privacy*: The PRF service must learn nothing about m. Message privacy supports clients that require sensitive values such as passwords to remain private even if the service is compromised, or to promote psychological acceptability in the case that a separate organization (e.g., a cloud provider) manages the service.
- *Tweak visibility*: The server must learn tweak t to permit fine-grained rate-limiting of requests.[1] In the password storage example, a distinct tweak is assigned to each user account, allowing the service to detect and limit guessing attempts against individual user accounts.
- *Verifiability:* A client must be able to verify that a PRF service has correctly computed F_{k_w} for a ensemble selector w and tweak/message pair t, m. This ensures, after first use of an ensemble by a client, that a subsequently compromised server cannot surreptitiously reply to PRF queries with incorrect values.[2]
- *Client-requested ensemble key rotations:* A client must be permitted to request a rotation of its ensemble pre-key $\mathtt{K}[w]$ to a new one $\widehat{\mathtt{K}[w]}$. The server must be able to provide an update token Δ_w to roll forward PRF outputs under $\mathtt{K}[w]$ to become PRF outputs under $\widehat{\mathtt{K}[w]}$, meaning that the PRF is *key-updatable* with respect to ensemble keys. Additionally, Δ_w must be *compact*, i.e., constant in the number of PRF invocations already performed under w. Clients can mandate that rotation requests be authenticated (to prevent malicious key deletion). A client must additionally be able to *transfer* an ensemble from one selector w to another selector w'.
- *Master secret rotations:* The server must be able to rotate the master secret key msk with minimal impact on clients. Specifically, the PRF must be key-updatable with respect to the master secret key msk so that PRF outputs under msk can be rolled forward to a new master secret $\widehat{msk}$. When such a rotation occurs, the server must provide a compact update token δ_w for each ensemble w.
- *Forward security:* Rotation of an ensemble key or master secret key results in complete erasure of the old key and the update token.

Two sets of challenges arise in designing PYTHIA. The first is cryptographic. It turns out that the combination of requirements above are not satisfied by any existing protocols we could find. Ultimately we realized a new type of cryptographic primitive was needed that proves to be a slight variant of oblivious PRFs and blind signatures. We discuss the new primitive, and our efficient protocol realizing it, in the next section. The second set of challenges surrounds building a full-featured service that provides the core cryptographic protocol, which we treat in Section 4.

3 Partially-oblivious PRFs

We introduce the notion of a (verifiable) partially-oblivious PRF. This is a two-party protocol that allows the secure computation of $F_{k_w}(t, m)$, where F is a PRF with server-held key k_w and t, m are the input values. The client can verify the correctness of $F_{k_w}(t, m)$ relative to a public key associated to k_w. Following our terminology, t is a tweak and m is a message. We say the PRF is partially oblivious because t is revealed to the server, but m is hidden from the server.

Partially oblivious PRFs are closely related to, but distinct from, a number of existing primitives. A standard oblivious PRF [27], or its verifiable version [28], would hide both t and m, but masking both prevents granular rate limiting by the server. Partially blind signatures [1] allow a client to obtain a signature on a similarly partially blinded input, but these signatures are randomized and the analysis is only for unforgeability which is insufficient for security in all of our applications.

We provide more comparisons with related work in Section 7 and a formal definition of the new primitive in Appendix B. Here we will present the protocol that suffices for PYTHIA. It uses an admissible bilinear pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ over groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of prime order q, and a pair of hash functions $H_1 : \{0,1\}^* \to \mathbb{G}_1$ and $H_2 : \{0,1\}^* \to \mathbb{G}_2$ (that we will model as random oracles). More details on pairings are provided in Appendix B. A secret key k_w is an element of $\mathbb{Z}_p$. The PRF F that the protocol computes is:

$$F_{k_w}(t, m) = e\big(H_1(t), H_2(m)\big)^{k_w} .$$

This construction coincides with the Sakai, Ohgishi, and Kasahara [44] construction for non-interactive identity-based key exchange, where t and m would be different identities and k_w a secret held by a trusted key authority. Likewise, this construction is equivalent to the left-or-right constrained PRF of Boneh and Waters [13]. The

[1] In principle, the server need only be able to link requests involving the same t, not learn t. Explicit presentation of t is the simplest mechanism that satisfies this requirement.

[2] This matters, for example, if an attacker compromises the communication channel but not the server's secrets (msk and $\mathtt{K}[w]$). Such an attacker must not be able to convince the client that arbitrary or incorrect values are correct.

```
                                  PRF-Srv ($msk$)
PRF-Cl ($w, t, m$)
$r \leftarrow\!\$\ \mathbb{Z}_q$
$x \leftarrow H_2(m)^r$
                    $\xrightarrow{w, t, x}$
                                  $\tilde{x} \leftarrow e(H_1(t), x)$
                                  $k_w \leftarrow \text{HMAC}(msk, \texttt{K}[w])$
                                  $p_w \leftarrow g^{k_w}$
                                  $y \leftarrow \tilde{x}^{k_w}$
                                  $\pi \leftarrow\!\$\ \text{ZKP}(\text{DL}_g(p_w) = \text{DL}_{\tilde{x}}(y))$
                    $\xleftarrow{p_w, y, \pi}$
If $p_w$ matches &
$\pi$ verifies then
    Ret $y^{1/r}$
Else Ret $\perp$
```

Figure 2: The partially-oblivious PRF protocol used in PYTHIA. The value π is a non-interactive zero-knowledge proof that the indicated discrete logs match. The client also checks that p_w matches ones seen previously when using selector w.

contexts of these prior works are distinct from ours and our analyses will necessarily be different, but we note that all three settings similarly exploit the algebraic structure of the bilinear pairing. See Section 7 for further discussion of related work.

The client-server protocol that computes $F_{k_w}(t, m)$ in a partially-oblivious manner is given in Figure 2. There we let g be a generator of $\mathbb{G}_1$. We now explain how the protocol achieves our requirements described in the last section.

Blinding the message: In our protocol, the client blinds the message m, hiding it from the server, by raising it to a randomly selected exponent $r \leftarrow\!\$\ \mathbb{Z}_q$. As $e\big(H_1(t), H_2(m)^r\big) = e\big(H_1(t), H_2(m)\big)^r$, the client can unblind the output y of PRF-Srv by raising it to $1/r$. This protocol hides m unconditionally, as $H_2(m)^r$ is a uniformly random element of $\mathbb{G}_2$.

Verifiability: The protocol enables a client to verify that the output of PRF-Srv is correct, assuming the client has previously stored p_w. The server accompanies the output y of the PRF with a zero-knowledge proof π of correctness.

Specifically, for a public key $p_w = g^{k_w}$, where g is a generator of $\mathbb{G}_1$, the server proves $\text{DL}_g(p_w) = \text{DL}_{\tilde{x}}(y)$. Standard techniques (see, e.g., Camenisch and Stadler [17]) permit efficient ZK proofs of this kind in the random oracle model. [3] The notable computational costs for the server are one pairing and one exponentiation in $\mathbb{G}_T$; for the client, one pairing and two exponentiations in $\mathbb{G}_T$. [4]

Efficient key updates: The server can quickly and easily update the key k_w for a given ensemble selector w by replacing the table entry $s = \texttt{K}[w]$ with a new, randomly selected value s', thereby changing $k_w = \text{HMAC}(msk, s)$ to $k'_w = \text{HMAC}(msk, s')$. It can then transmit to the client an update token of the form $\Delta_w = k'_w / k_w \in \mathbb{Z}_q$.

The client can update any stored PRF value $F_{k_w}(t, m) = e\big(H_1(t), H_2(m)\big)^{k_w}$ by raising it to Δ_w; it is easy to see that $F_{k_w}(t, m)^{\Delta_w} = F_{k'_w}(t, m)$.

The server can use the same mechanism to update msk, which requires generating a new update token for each w and pushing these tokens to clients as needed.

Unblinded variants. For deployments where obliviousness of messages is unnecessary, we can use a faster, unblinded variant of the PYTHIA protocol that dispenses with pairings. The only changes are that the client sends m to the server, there is no unblinding of the server's response, and, instead of computing

$$\tilde{x} \leftarrow e(H_1(t), x)$$

the server computes

$$\tilde{x} \leftarrow H_3(t \,\|\, m) \ .$$

All group operations in this unblinded variant are over a standard elliptic curve group $\mathbb{G} = \langle g \rangle$ of order q and we use a hash function $H_3 : \ \{0,1\}^* \to \mathbb{G}$.

An alternative unblinded construction would be to have the server apply the Boneh-Lynn-Shacham short signatures [12] to the client-submitted $t \,\|\, m$; verification of correctness can be done using the signature verification routine, and we can thereby avoid ZKPs. This BLS variant may save a small amount of bandwidth.

These unblinded variants provide the same services (verifiability and efficient key updates) and security with the obvious exception of the secrecy of the message m. In some deployment contexts an unblinded protocol may be sufficient, for example when the client can maintain state and submit a salted hash m instead of m directly. In this context, the salt should be held as a secret on the client and never sent to the server.

4 The PYTHIA Service Design

Figure 3 gives the high-level API exposed by PYTHIA to a client. We now describe its functions in terms of the lifecycle of an ensemble key. We assume a security parameter n specifying symmetric key lengths; a typical choice would be $n = 128$.

[3] Some details: The prover picks $v \leftarrow\!\$\ \mathbb{Z}_q$ and then computes $t_1 = g^v$ and $t_2 = \tilde{x}^v$ and $c \leftarrow H_3(g, p_w, \tilde{x}, y, t_1, t_2)$. Let $u = v - c \cdot k$. The proof is $\pi = (c, u)$. The verifier computes $t'_1 = g^u \cdot p_w^c$ and $t'_2 = \tilde{x}^u y^c$. It outputs true if $c = H_3(g, p_w, \tilde{x}, y, t'_1, t'_2)$.

[4] The client's pairing can be pre-computed while waiting for the server's reply.

Command	Description
$\mathsf{Init}(w\ [, options])$	Create table entry $\mathtt{K}[w]$ (for ensemble key k_w)
$\mathsf{Eval}(w, t, m)$	Return PRF output $F_{k_w}(t, m)$
$\mathsf{Reset}(w,\ authtoken)$	Update $\mathtt{K}[w]$ (and thus k_w); return update token Δ_w
$\mathsf{GetAuth}(w)$	Send one-time authentication token $authtoken$ to client

Figure 3: The basic PYTHIA API.

We defer to later sections the underlying client-server protocols and to Appendix A details on key lifecycle management options, additional API calls for token management and ensemble transfer, and a discussion of master secret key rotation.

Ensemble initialization. To begin using the PYTHIA service, a client creates an ensemble key for selector w by invoking $\mathsf{Init}(w\ [, options])$. PYTHIA generates a fresh, random table entry $\mathtt{K}[w]$. Recall that ensemble key $k_w = \text{HMAC}(msk, \mathtt{K}[w])$. So Init creates k_w as a byproduct.

Ideally, w should be an unguessable byte string. (An easily guessed one may allow attackers to squat on a key selector, thereby mounting a denial-of-service (DoS) attack.) For some applications, as we explain below, this isn't always possible. If an ensemble key for w already exists, then the PYTHIA service returns an error to the client. Otherwise, the client receives a message signifying that initialization is successful.

Init includes a number of options we detail in Appendix A.

PRF evaluation. To obtain a PRF value, a client can perform an evaluation query $\mathsf{Eval}(w, t, m)$, which returns $F_{k_w}(t, m)$. Here t is a tweak and m is a message. To compute the PRF output, the client and server perform a one-round cryptographic protocol (meaning a single message from client to server, and one message back). We present details in Section 3, but remind the reader that t is visible to the server in the client-server protocol invoked by Eval, while m is blinded.

The server rate-limits requests based on the tweak t, and can also raise an alert if the rate limit is exceeded. We give example rate limiting policies in Section 5.

Ensemble-key reset. A client can request that an ensemble key k_w be reset by invoking $\mathsf{Reset}(w)$. This reset is accomplished by overwriting $\mathtt{K}[w]$ with a fresh, random value. The name service returns a compact (e.g., 256-bit) update token Δ_w that the client may use to update all PRF outputs for the ensemble. It stores this token locally, encrypted under a public key specified by the client, as explained below.

Note that reset *results in erasure of the old value of* k_w. Thus a client that wishes to delete an ensemble key k_w permanently at the end of its lifecycle can do so with a Reset call.

Reset is an authenticated call, and thus requires the following capability.

Authentication. To authenticate itself for API calls, the client must first invoke GetAuth, which has the server transmit an (encrypted) authentication token $authtoken$ to the client out-of-band. The token expires after a period of time determined by a configuration parameter in PYTHIA. Our current implementation uses e-mail for this, see Appendix A for more details. Of course, in some deployments one may want authentication to be performed in other ways, such as tokens dispensed by administrators (for enterprise settings) or simply given out on a first-come-first-serve basis for each ensemble identifier (for public Internet services).

4.1 Implementation

We implemented a prototype PYTHIA PRF service as a web application accessed over HTTPS. All requests are first handled by an nginx web server with uWsgi as the application server gateway that relays requests to a Django back-end. The PRF-Srv functionality is implemented as a Django module written in Python. Storage for the server's key table and rate-limiting information is done in MongoDB.

We use the Relic cryptographic library [2] (written in C) with our own Python wrapper. We use Barreto-Naehrig 254-bit prime order curves (BN-254) [4]. These curves provide approximately 128-bits of security.

In our experiments the service is run on a single (virtual) machine, but our software stack permits components (web server, application sever, database) to be distributed among multiple machines with updates to configuration files.

For the purpose of comparison, we implemented three variants of the PYTHIA service. The first two are the unblinded protocols described in Section 3. In these two schemes, the client sends m in the clear (possibly hashed with a secret salt value first) and the server replies with $y = H_1(t \,\|\, m)^k$. In the first scheme, denoted UNB, the server provides $p = g_1^k$ and a zero-knowledge proof where g_1 is a generator of $\mathbb{G}_1$. The second scheme, denoted BLS, uses a BLS signature for verification. The server provides $p = g_2^k$ where g_2 is a generator of $\mathbb{G}_2$ and the client verifies the response by computing and comparing the values: $e(y, g_2) = e(H_1(t \,\|\, m), p)$.

Our partially-oblivious scheme is denoted PO.

For the evaluation below we use a Python client implementing PRF-Cl for all three schemes using the same

	Time (µs)		
Group	Group Op	Exp	Hashing
$\mathbb{G}_1$	5.7	175	77
$\mathbb{G}_2$	6.7	572	210
$\mathbb{G}_T$	9.8	1145	–
pairing operation (e) takes 1005 µs			

Figure 4: Time taken by each operation in BN-254 groups. Hashing times are for 64-byte inputs.

libraries indicated above for the server and httplib2 to perform HTTPS requests.

4.2 Performance

For performance and scalability evaluation we hosted our PYTHIA server implementation on Amazon's Elastic Compute Cloud (EC2) using a c4.xlarge instance which provides 8 virtual CPUs (Intel Xeon third generation, 2.9GHz), 15 GB of main memory, and solid state storage. The web server, nginx, was configured with basic settings recommended for production deployment including one worker process per CPU.

Latency. We measured client query latency for each protocol using two clients: one within the same Amazon Web Service (AWS) availability zone (also c4.xlarge) and one hosted at the University of Wisconsin–Madison with an Intel Core i7 CPU (3.4 GHz). We refer to the first as the LAN (local-area network) setting and the second as the WAN (wide-area network) setting. In the LAN settings we used the AWS internal IP address. All queries were made over TLS and measurements include the time required for clients to blind messages and unblind results (PO), as well as verify proofs provided by the server (unless indicated otherwise). All machines used for evaluation were running Ubuntu 14.04.

Microbenchmarks for group operations appear in Figure 4 and Figure 5 shows the timing of individual operations that comprise a single PRF evaluation. All results are mean values computed over 10,000 operations. These values were captured on an EC2 c4.xlarge instance using the Python profiling library line_profiler. The most expensive operations, by a large margin, are exponentiation in $\mathbb{G}_t$ and the pairing operation. By extension, PO sign, prove, and verify operations become expensive.

We measured latencies averaged over 1,000 PRF requests (with 100 warmup requests) for each scheme and the results appear in Figure 6. Computation time dominates in the LAN setting due to the almost negligible network latency. The WAN case with cold connections (no HTTP KeepAlive) pays a performance penalty due to the four round-trips required to set up a new TCP and TLS connection. While even 400 ms latencies are not prohibitive in our applications, straightforward engineering improvements would vastly improve WAN timing: using TLS session resumption, using lower-latency secure protocol like QUIC [43], or even switching to a custom UDP protocol (for an example one for oblivious PRFs, see [5]).

Server Op	Time (ms)		
Table	1.2		
Rate-limit	0.9		
	UNB	BLS	PO
Sign	0.3	0.3	1.5
Prove	0.5	0.3	2.5

Client Op	UNB	BLS	PO
Blind	-	-	0.3
Unblind	-	-	1.2
Verify	0.9	2.0	4.0

Figure 5: Computation time for major operations to perform a PRF evaluation. Table retrieves $K[w]$ from database; Rate-limit updates rate-limiting record in database; and Sign generates the PRF output;

	Latency (ms)					
	LAN			WAN		
Scheme	Cold	Hot	No π	Cold	Hot	No π
UNB	7.0	3.8	2.4	389	82	80
BLS	7.9	4.9	2.4	392	85	80
PO	14.9	11.8	5.2	403	96	84
RTT ping	0.1			82		

Figure 6: Average latency to complete a PRF-Cl with client-server communication over HTTPS. LAN: client and server in the same EC2 availability zone. WAN: server in EC2 US-West (California) and client in Madison, WI. Hot connections made with HTTP KeepAlive enabled; cold connections with KeepAlive disabled. No π: KeepAlive enabled; prove and verify computations are skipped.

Throughput. We used the distributed load testing tool autobench to measure maximum throughput for each scheme. We compare to a static page containing a typical PRF response served over HTTPS as a baseline. We used two clients in the same AWS region as the server. All connections were cold: no TLS session resumption or HTTP KeepAlive. The maximum throughput for a static page is 2,200 connections per second (cps); UNB and BLS 1,400 cps; and PO 1,350 cps. Thus our PYTHIA implementation can handle a large number of clients on a single EC2 instance. If needed, the implementation can be scaled with standard techniques (e.g., a larger number of web servers and application servers on the front-end with a distributed key-value store on the back-end).

Storage. Our implementation stores all ensemble prekey table (K) entries and rate-limiting information in MongoDB. A table entry is two 32 byte values: a SHA-

256 hash of the ensemble selector w and its associated value $\mathtt{K}[w]$. In MongoDB the average storage size is 195 bytes per entry (measured as the average of 100K entries), including database overheads and indexes. This implementation scales easily to 100 M clients with under 20 GB of storage.

To rate-limit queries, our implementation stores tweak values along with a counter and a timestamp (to expire old entries) in MongoDB. Tweak values are also hashed using SHA-256 which ensures entries are of constant length. In our implementation each distinct tweak requires an average of 144 bytes per entry (including overheads and indexes). Note however that rate limiting entries are purged periodically as counts are only required for one rate-limiting period. Our implementation imposes rate-limits at hour granularity. Assuming a maximum throughput of 2,000 requests per second, rate-limiting storage never exceeds 1 GB.

All told, with only 20 GB stored data, PYTHIA can serve over 100 M clients and perform rate-limiting at hour granularity. Thus fielding a database for PYTHIA can be accomplished on commodity hardware.

5 Password Onions

Web servers and other systems frequently store passwords in hashed form. A *password onion* is the result of additionally invoking a PRF service to harden the hash. In currently suggested onions, one sequentially combines local hashing and application of the PRF service.

We now present a service that we have implemented on top of PYTHIA for managing password onions. First, we describe the limitations of contemporary systems as exemplified by a recently disclosed architecture employed by Facebook [39]. Then we show how our password-onion system, which was easily engineered on top of PYTHIA, can address these limitations.

In what follows, we use the term “client” or “web server” to denote the server performing authentication and storing derived values from passwords and “PRF server” to denote the PYTHIA service.

5.1 Facebook password onion

An example of a contemporary system, used by Facebook, is given in Figure 7.[5] Their PRF service applies HMAC using a service-held secret and returns the result. In this architecture, an adversary that compromises the web server and the password hashes it stores must still mount an online attack against the PRF service to compromise accounts. This is a big advance on the hashing-only practices that are commonly used.

```
PW-Onion($pw$)
$h_1 \leftarrow$ MD5($pw$)
$sa \leftarrow_{\$} \{0,1\}^{160}$
$h_2 \leftarrow$ HMAC[SHA-1]($h_1, sa$)
$h_3 \leftarrow$ PRF-Cl($h_2$) = HMAC[SHA-256]($h_2, msk$)
$h_4 \leftarrow$ scrypt($h_3, sa$)
$h_5 \leftarrow$ HMAC[SHA-256]($h_4$)
Ret ($sa, h_5$)
```

Figure 7: The Facebook password onion. PRF-Cl(h_2) invokes the Facebook PRF service HMAC[SHA-256](h_2, K_s) with PRF-service secret key K_s.

The Facebook architecture nevertheless has some shortcomings. It is easy to see from Figure 7 that Facebook’s system, like most contemporary PRF services, lacks several important features present in PYTHIA. One is message privacy: the Facebook PRF service applies HMAC to h_2. This is the salted hash of the password, and so learning the salt as well as compromising the PRF service suffices to re-enable offline brute-force attacks. This threat is avoided by PYTHIA due to blinding.

Another feature is batch key updates. In fact, the Facebook PRF service doesn’t permit autonomous key updates at all, in the sense of an update to msk that can be propagated into PRF output updates. Should the client (password database) be compromised, the only way to reconstitute a hash in an existing password onion is to *wait until a user logs in and furnishes* pw. It is not clear whether the Facebook PRF service performs granular rate-limiting, although no such capability is indicated in [38]. PYTHIA, as we shall see, addresses all of these issues by design in our password onion system.

The Facebook onion also presents a subtle performance issue. By applying cryptographic primitives serially, the time to hash a password equals the time for local computations, call it t_{local}, *plus* the time for the round-trip PRF service call, call it t_{prf}. An attacker that compromises the web service and PRF service incurs no network latency, and thus may gain a considerable advantage in guessing time over an honest web server. In our PYTHIA-based password onion service, we address this issue by observing that it is possible to avoid serialization of key derivation functions on the web server and the PRF service call. That is, we introduce in our PYTHIA-based service the idea of *parallelizable password onions.*

[5] This figure is of “archaeological” interest. It appears that vulnerabilities in MD5 led to the addition of a layer of processing under SHA-1; when vulnerabilities were found in SHA-1, Facebook then added layers of SHA-256. As we explain later, full-blown replacement of MD5 and SHA-1 with SHA-256 was not easily accomplished.

```
UpParOnion$(w, sa, pw)$
$z \leftarrow$ PBKDF$(pw, sa)$
$u \leftarrow$ PRF-Cl$(w, sa, pw)$
$h \leftarrow u^z$
Ret $(h, sa)$
```

Figure 8: An updatable, parallelizable password onion. PRF-Cl returns elements of a group $\mathbb{G}$. The value w is a unique PRF-service identifier for the web server (e.g., a random 256-bit string) and sa is a random per-user salt value.

5.2 PYTHIA password onion

The onion algorithm we construct for PYTHIA is shown in Figure 8. For PYTHIA, the output of PRF-Cl is an element of a group $\mathbb{G}_T$. To use this service, a web server stores (h, sa) upon password registration; it verifies a proffered password pw' by checking that UpParOnion$(w, sa, pw') = h$. Written out we have that:

$$h = u^z = e(H_1(sa), H_2(pw))^{k_w z}.$$

This design ensures that the key update functions in the PYTHIA API may be used to update onions as well. For example, to update an ensemble key k_w to k'_w, the service computes and furnishes to the web server an update token $\Delta_w = k'_w/k_w$. The web server may compute h^{Δ_w} for each stored value h.

Parallelization. Password verification here is *parallelizable* in the sense that z and u may be computed independently and then combined. Such parallel implementation of the onion achieves a password verification latency of $\max\{t_{local}, t_{prf}\}$ (plus a single exponentiation), as opposed to $t_{local} + t_{prf}$ in a serialized implementation.

A web server generally aims to achieve a verification latency equal to some latency target T that is high enough to slow offline brute-force attacks, but low enough not to burden users. For a parallelized onion a web server can meet its latency target by setting $t_{local}, t_{prf} \approx T$. At the same time an offline attacker that has compromised the web server and PYTHIA must perform about $t_{local}+t_F > T$ work to check a single password guess, where t_F is the computation time of F_{k_w} (i.e., t_{prf} minus network latency). An attacker can parallelize, but her total work still goes up relative to the serial onion approach for the same latency target T.

We estimate the security improvement of parallel onions over serial onions using our benchmarks from Section 4.2. We fix a login latency budget of $T = 300$ ms.[6] The latency costs for a PYTHIA query with hot connections are 12 ms (LAN) and 96 ms (WAN). If one performs computations serially with a fixed T then PBKDF computations need to be reduced by 4% (LAN) and 32% (WAN) compared to the parallel approach. In the event that the PYTHIA server and password database are compromised, the serial onion enables speedup of offline dictionary attacks by the same percentages.

Rate limiting and logging. The transparency of tweaks enables the PYTHIA PRF service in this setting to execute any of a wide range of rate-limiting policies with per-account visibility (in contrast to what may be in Facebook an account-blind PRF service). As an example demonstrating the flexibility of our architecture, in our implementation PYTHIA performs a tiered rate-limiting: for a given account (t), it limits queries to at most 10 per hour per account, and at most 300 per month. (In expectation, guessing a random 4-digit PIN would require 1.4 years under this policy.) It logs violations of these thresholds. In a production environment, it could also send alerts to security administrators.

We emphasize that a wide range of other rate-limiting policies is possible. We also point out that PYTHIA's rate limiting supplements that normally implemented at the web server for remote login requests. PYTHIA performs rate limiting and may issue alerts even if the web server is compromised.

Key update. The key update calls in the PYTHIA API, and the ability to rotate either k_w or msk efficiently, propagates up to the password onion service. Key updates instantly invalidate the web server's existing password database—a useful capability in case of compromise. A compromised database becomes useless to an attacker attempting to recover passwords, even with the ability to query PYTHIA. Using a key update token, the web server can then recover from compromise by refreshing its database.

We created a client simulator with MongoDB and the mongoengine Python module. With this we benchmarked key updates with 100,000 database entries. The client requested a key update from PYTHIA, received the update token Δ_w, and updated each database entry. The complete update required less than 1 ms per entry, and terminated in less than 97 seconds for all 100,000 entries. For a larger database we assume updates scale linearly, and so an update for 1 million users completes in under 17 minutes.

The web server need not need lock the database to perform updates; it can execute them in parallel with normal login operations. Doing so does require additional versioning information for each entry to indicate the version of k_w (in the simplest form, whether or not it has received the latest update).

[6] This is the default setting for Python's bcrypt and scrypt modules, though all PBDKFs are tunable so one can choose T to be any value desired.

6 Hardened Brainwallets

Brainwallets are a common but dangerous way to secure accounts in the popular cryptocurrency Bitcoin, as well as in less popular cryptocurrencies such as Litecoin. Here we describe how the PYTHIA service can be used directly as a means to harden brainwallets. This application showcases the ease with which a wide variety of applications can be engineered around PYTHIA.

How brainwallets work. Every Bitcoin account has an associated private / public key pair (sk, pk). The private key sk is used to produce digital (ECDSA) signatures that authorize payments from the account. The public key pk permits verification of these signatures. It also acts as an account identifier; a Bitcoin address is derived by hashing pk (under SHA-256 and RIPEMD-160) and encoding it (in base 58, with a check value).

Knowledge of the private key sk equates with control of the account. If a user loses a private key, she therefore loses control over her account. For example, if a high entropy key sk is stored exclusively on a device such as a mobile phone or laptop, and the device is seized or physically destroyed, the account assets become irrecoverable.

Brainwallets offer an attractive remedy for such physical risks of key loss. A brainwallet is simply a password or passphrase P memorized by a Bitcoin account holder. The private key sk is generated directly from P. Thus the user's memory serves as the only instrument needed to authorize access to the account.

In more detail, the passphrase is typically hashed using SHA-256 to obtain a 256-bit string $sk =$ SHA-256(P). Bitcoin employs ECDSA signatures on the `secp256k1` elliptic curve; with high probability ($\approx 1 - 2^{-126}$), sk is less than the group order, and a valid ECDSA private key. (Some websites employ stronger key derivation functions. For example, WrapWallet by keybase.io [29] derives sk from an XOR of each of PBKDF2 and scrypt applied to P and permits use of a user-supplied salt.)

Since a brainwallet employs only P as a secret, and does not necessarily use any additional security measures, an attacker that guesses P can seize control of a user's account. As account addresses are posted publicly in the Bitcoin system (in the "blockchain"), an attacker can easily confirm a correct guess. Brainwallets are thus vulnerable to brute-force, offline guessing attacks. Numerous incidents have come to light showing that brainwallet cracking is pandemic [14].[7]

6.1 A PYTHIA-hardened brainwallet

PYTHIA offers a simple, powerful means of protecting brainwallets against offline attack. Hardening P in the same manner as an ordinary password yields a strong key $\tilde{P}$ that can serve in lieu of P to derive sk.

To use PYTHIA, a user chooses a unique identifier id, e.g., her e-mail address, an account identifier $acct$, and a passphrase P. The identifier $acct$ might be used to distinguish among Bitcoin accounts for users who wish to use the same password for multiple wallets. The client then sends $(w = id, t = id \,\|\, acct, m = P)$ to the PYTHIA service to obtain the hardened value $F_{k_w}(t, m) = \tilde{P}$. Here, id is used both as an account identifier and as part of the salt. Message privacy in PYTHIA ensures that the service learns nothing about P. Then $\tilde{P}$ is hashed with SHA-256 to yield sk. The corresponding public key pk and address are generated in the standard way from sk [7].

PYTHIA forces a would-be brainwallet attacker to mount an online attack to compromise an account. Not only is an online attack much slower, but it may be rate-limited by PYTHIA and detected and flagged. As the PYTHIA service derives $\tilde{P}$ using a user-specific key, it additionally prevents an attacker from mounting a dictionary attack against multiple accounts. While in the conventional brainwallet setting, two users who make use of the same secret P will end up controlling the same account, PYTHIA ensures that the same password P produces distinct per-user key pairs.

Should an attacker compromise the PYTHIA service and steal msk and K, the attacker must still perform an offline brute-force attack against the user's brainwallet. So in the worst case, a user obtains security with PYTHIA at least as good as without it.

Additional security issues. A few subtle security issues deserve brief discussion:

- *Stronger KDFs:* To protect against brute-force attack in the event of PYTHIA compromise, a resource-intensive key-derivation function may be desirable, as is normally used in password databases. This can be achieved by replacing the SHA-256 hash of $\tilde{P}$ above with an appropriate KDF computation, or alternatively using an onion approach described in Section 5.
- *Denial-of-service:* By performing rate-limiting, PYTHIA creates the risk of targeted denial-of-service attacks against Bitcoin users. As Bitcoin is pseudonymous, use of an e-mail address as a PYTHIA key-selector suffices to prevent such attacks against users based on their Bitcoin addresses alone. Users also have the option, of course, of using a semi-secret id. A general DoS attack against

[7] At one point, rumor had it that cracking brainwallets was more profitable than "mining,", the basic process of generating fresh Bitcoins.

the PYTHIA service is also possible, but of similar concern for Bitcoin itself [8].

- *Key rotation:* Rotation of an ensemble key k_w (or the master key msk) induces a new value of $\tilde{P}$ and thus a new (sk, pk) pair and account. A client can handle such rotations in the naïve way: transfer funds from the old address to the new one.
- *Catastrophic failure of* PYTHIA*:* If a PYTHIA service fails catastrophically, e.g., msk or $\mathtt{K}$ is lost, then in a typical setting, it is possible simply to reset users' passwords. In the brainwallet case, the result would be loss of virtual-currency assets protected by the server—a familiar event for Bitcoin users [35]. This problem can be avoided, for instance, using a threshold implementation of PYTHIA, as mentioned in Section 6.2 or storing sk in a secure, offline manner like a safe-deposit box for disaster recovery.

6.2 Threshold Security

In order to gain both redundancy and security, we give a threshold scheme that can be used with a number of Pythia servers to protect a secret under a single password. This scheme uses Shamir's secret sharing threshold scheme [45] and gives (k, n) threshold security. That is, initially, n Pythia servers are contacted and used to protect a secret s, and then any k servers can be used to recover s and any adversary that has compromised fewer than k Pythia servers learns no information about s.

Preparation. The client chooses an ensemble key selector w, tweak t, password P, and contacts n Pythia servers to compute $q_i = \text{PRF-Cl}_i(w, t, P) \mod p$ for $0 < i \leq n$. The client selects a random polynomial of degree $k - 1$ with coefficients from $\mathbb{Z}_p^*$ where p is a suitably large prime: $f(x) = \sum_{j=0}^{k-1} x^j a_j$. Let the secret $s = a_0$. Next the client computes the vector $\Phi = (\phi_1, ..., \phi_n)$ where $\phi_i = f(i) - q_i$. The client durably stores the value Φ, but does not need to protect it (it's not secret). The client also stores public keys p_i from each Pythia server to validate proofs when issuing future queries.

Recovery. The client can reconstruct s if she has Φ by querying any k Pythia servers giving k values q_i. These q_i values can be applied to the corresponding Φ values to retrieve k distinct points that lie on the curve $f(x)$. With k points on a degree $k - 1$ curve, the client can use interpolation to recover the unique polynomial $f(x)$, which includes the curve's intercept $a_0 = s$.

Security. If an adversary is given Φ, w, t, the public keys p_i, a ciphertext based on s, and the secrets from $m < k$ Pythia servers, the adversary has no information that will permit her to verify password guesses offline. Compared to [45], this scheme reduces the problem of storing n secrets to having access to n secure OPRFs and durable (but non-secret) storage of the values Φ and public keys p_i.

Verification. Verification of server responses occurs within the Pythia protocol. If a server is detected to be dishonest (or goes out of service), it can be easily replaced by the client without changing the secret s. To replace a Pythia server that is suspected to be compromised or detected as dishonest, the client reconstructs the secret s using any k servers, executes Reset operations on all remaining servers: this effects a cryptographic erasure on the values Φ and $f(x)$. The client then selects a new, random polynomial, keeping a_0 fixed, and generates and stores an updated Φ' that maps to the new polynomial.

7 Related Work

We investigated a number of designs based on existing cryptographic primitives in the course of our work, though as mentioned none satisfied all of our design goals. Conventional PRFs built from block ciphers or hash functions fail to offer message privacy or key rotation. Consider instead the construction $H(t \,\|\, m)^{k_w}$ for $H : \{0,1\}^* \to \mathbb{G}$ a cryptographic hash function mapping onto a group $\mathbb{G}$. This was shown secure as a conventional PRF by Naor, Pinkas, and Reingold assuming decisional Diffie-Hellman (DDH) is hard in $\mathbb{G}$ and when modeling H as a random oracle [40]. It supports key rotations (in fact it is key-homomorphic [11]) and verifiability can be handled using non-interactive zero-knowledge proofs (ZKP) as in PYTHIA. But this approach fails to provide message privacy if we submit both t and m to the server and have it compute the full hash.

One can achieve message-hiding by using blinding: have the client submit $X = H(t \,\|\, m)^r$ for random $r \in \mathbb{Z}_{|\mathbb{G}|}$ and the server reply with X^{k_w} as well as a ZKP proving this was done correctly. The resulting scheme is originally due to Chaum and Pedersen [19], and suggested for use by Ford and Kaliski [26] in the context of threshold password-authenticated secret sharing (see also [3, 15, 20, 34]). There an end user interacts with one or more blind signature servers to derive a secret authentication token. If $\mathbb{G}$ comes equipped with a bilinear pairing, one can dispense with ZKPs. The resulting scheme is Boldyreva's blinded version [10] of BLS signatures [12]. However, neither approach provides granular rate limiting when blinding is used: the tweak t is hidden from the server. Even if the client sends t as well, the server cannot verify that it matches the one used to compute X and attackers can thereby bypass rate limits.

To fix this, one might use Ford-Kaliski with a separate secret key for each tweak. This would result in having a different key for each unique w, t pair. Message privacy is maintained by the blinding, and querying $w, t, H(t' \| m)^r$ for $t \neq t'$ does not help an attacker circumvent per-tweak rate limiting. But now the server-side storage grows in the number of unique w, t pairs, a client using a single ensemble w must now track N public keys when they use the service for N different tweaks, and key rotation requires N interactions with the PRF server to get N separate update tokens (one per unique tweak for which a PRF output is stored). When N is large and the number of ensembles w is small as in our password storage application, these inefficiencies add significant overheads.

Another issue with the above suggestions is that their security was only previously analyzed in the context of one-more unforgeability [42] as targeted by blind signatures [18] and partially blind signatures [1]. (Some were analyzed as conventional PRFs, but that is in a model where adversaries do not get access to a blinded server oracle.) The password onion application requires more than unforgeability because message privacy is needed. (A signature could be unforgeable but include the entire message in its signature, and this would obviate the benefits of a PRF service for most applications.) These schemes, however, can be proven to be one-more PRFs, the notion we introduce, under suitable one-more DDH style assumptions using the same proof techniques found in Appendix B.

Fully oblivious PRFs [27] and their verifiable versions [28] also do not allow granular rate limiting. We note that the Jarecki, Kiayias, and Krawczyk constructions of verifiable OPRFs [28] in the RO model are essentially the Ford-Kaliski protocol above, but with an extra hash computation, making the PRF output $H'(t \| m \| H(t \| m)^{k_w})$. Our notion of one-more unpredictability in the appendix captures the necessary requirements on the inner cryptographic component, and might modularize and simplify their proofs. Their transform is similar to the unique blind signature to OPRF transformation of Camenisch, Neven, and shelat [16]. None of these efficient oblivious PRF protocols support key rotations (with compact tokens or otherwise) as the final hashing step destroys updatability.

The setting of capture-resilient devices shares with ours the use of an off-system key-holding server and the desire to perform cryptographic erasure [32, 33]. They only perform protocols for encryption and signing functionalities, however, and not (more broadly useful) PRFs. They also do not support granular rate limiting and master secret key rotation.

Our main construction coincides with prior ones for other contexts. The Sakai, Ohgishi, and Kasahara [44] identity-based non-interactive key exchange protocol computes a symmetric encryption key as $e(H_1(ID_1), H_2(ID_2))^k$ for k a master secret held by a trusted party and ID_1 and ID_2 being the identities of the parties. See [41] for a formal analysis of their scheme. Boneh and Waters suggest the same construction as a left-or-right constrained PRF [13]. The settings and their goals are different from ours, and in particular one cannot use either as-is for our applications. Naïvely one might hope that returning the constrained PRF key $H_1(t)^{k_w}$ to the client suffices for our applications, but in fact this totally breaks rate-limiting. Security analysis of our protocol requires new techniques, and in particular security must be shown to hold when the adversary has access to a half-blinded oracle — this rules out the techniques used in [13, 41].

Key-updatable encryption [11] and proxy re-encryption [9] both support key rotation, and could be used to encrypt password hashes in a way supporting compact update tokens and that prevents offline brute-force attacks. But this would require encryption and decryption to be handled by the hardening service, preventing message privacy.

Verifiable PRFs as defined by [21, 22, 31, 36] allow one to verify that a known PRF output is correct relative to a public key. Previous verifiable PRF constructions are not oblivious, let alone partially oblivious.

Threshold and distributed PRFs [21, 37, 40] as well as distributed key distribution centers [40] enable a sufficiently large subset of servers to compute a PRF output, but previous constructions do not provide the granular rate limiting and key rotation we desire. However, it is clear that there are situations where applications would benefit from a threshold implementation of PYTHIA, for both redundancy and distribution of trust, as discussed in Section 6.2 for the case of brainwallets.

8 Conclusion

We presented the design and implementation of PYTHIA, a modern PRF service. Prior works have explored the use of remote cryptographic services to harden keys derived from passwords or otherwise improve resilience to compromise. PYTHIA, however, transcends existing designs to simultaneously support granular rate limiting, efficient key rotation, and cryptographic erasure. This set of features, which stems from practical requirements in applications such as enterprise password storage, proves to require a new cryptographic primitive that we refer to as a partially oblivious PRF.

Unlike a (fully) oblivious PRF, a partially oblivious PRF causes one portion of an input to be revealed to the server to enable rate limiting and detection of online brute-force attacks. We provided a bilinear-pairing based construction for partially oblivious PRFs that is highly efficient and simple to implement (given a pairings library), and also supports efficient key rotations. A formal proof of security is unobtainable using existing techniques (such as those developed for fully oblivious PRFs). We thus gave new definitions and proof techniques that may be of independent interest.

We implemented PYTHIA and show how it may be easily integrated it into a range of applications. We designed a new enterprise "password onion" system that improves upon the one recently reported in use at Facebook. Our system permits fast key rotations, enabling practical reactive and proactive key management, and uses a parallelizable onion design which, for a given authentication latency, imposes more computational effort on attackers after a compromise. We also explored the use of PYTHIA to harden brainwallets for cryptocurrencies.

Acknowledgements

The authors thank Kenny Paterson for feedback on an early draft of this paper. This work was supported in part by NSF grants CNS-1330308, CNS-1065134, and CNS-1330599, as well as a gift from Microsoft.

References

[1] Masayuki Abe and Tatsuaki Okamoto. Provably secure partially blind signatures. In *Advances in Cryptology–CRYPTO*. Springer, 2000.

[2] D. F. Aranha and C. P. L. Gouvêa. RELIC is an Efficient LIbrary for Cryptography. https://github.com/relic-toolkit/relic.

[3] Ali Bagherzandi, Stanislaw Jarecki, Nitesh Saxena, and Yanbin Lu. Password-protected secret sharing. In *Computer and Communications Security*. ACM, 2011.

[4] Paulo SLM Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In *Selected Areas in Cryptography*. Springer, 2006.

[5] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. Dupless: server-aided encryption for deduplicated storage. In *USENIX Security*. USENIX, 2013.

[6] Mihir Bellare, Chanathip Namprempre, David Pointcheval, Michael Semanko, and Matthew Franklin. The one-more-RSA-inversion problems and the security of Chaum's blind signature scheme. *Journal of Cryptology*, 16(3), 2003.

[7] Technical background of version 1 Bitcoin addresses. https://en.bitcoin.it/wiki/Technical_background_of_version_1_Bitcoin_addresses.

[8] Bitcoin wiki, "weaknesses". https://en.bitcoin.it/wiki/Weaknesses.

[9] Matt Blaze, Gerrit Bleumer, and Martin Strauss. Divertible protocols and atomic proxy cryptography. In *Advances in Cryptology–EUROCRYPT*. Springer, 1998.

[10] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In *Public Key Cryptography*. Springer, 2002.

[11] Dan Boneh, Kevin Lewi, Hart Montgomery, and Ananth Raghunathan. Key homomorphic PRFs and their applications. In *Advances in Cryptology–CRYPTO*. Springer, 2013.

[12] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In *Advances in Cryptology–ASIACRYPT*. Springer Berlin Heidelberg, 2001.

[13] Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In *Advances in Cryptology-ASIACRYPT*. Springer, 2013.

[14] Brainwallet. https://en.bitcoin.it/wiki/Brainwallet.

[15] Jan Camenisch, Anna Lysyanskaya, and Gregory Neven. Practical yet universally composable two-server password-authenticated secret sharing. In *Computer and Communications Security*. ACM, 2012.

[16] Jan Camenisch, Gregory Neven, and abhi shelat. Simulatable adaptive oblivious transfer. In *Advances in Cryptology–EUROCRYPT*. Springer Berlin Heidelberg, 2007.

[17] Jan Camenisch and Markus Stadler. Proof systems for general statements about discrete logarithms. Technical Report No. 260, Dept. of Computer Science, ETH Zurich, 1997.

[18] David Chaum. Blind signatures for untraceable payments. In *Advances in Cryptology*. Springer, 1983.

[19] David Chaum and Torben Pryds Pedersen. Wallet databases with observers. In *Advances in Cryptology–CRYPTO*. Springer, 1993.

[20] Mario Di Raimondo and Rosario Gennaro. Provably secure threshold password-authenticated key exchange. In *Advances in Cryptology–EUROCRYPT*. Springer, 2003.

[21] Yevgeniy Dodis. Efficient construction of (distributed) verifiable random functions. In *Public Key Cryptography*. Springer, 2002.

[22] Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In *Public Key Cryptography*. Springer, 2005.

[23] Paul Ducklin. Anatomy of a password disaster – Adobe's giant-sized cryptographic blunder, 2013. https://nakedsecurity.sophos.com/2013/11/04/anatomy-of-a-password-disaster-adobes-giant-sized-cryptographic-blunder/.

[24] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in Cryptology–CRYPTO*. Springer, 1985.

[25] Adam Everspaugh, Rahul Chatterjee, Samuel Scott, Ari Juels, and Thomas Ristenpart. The Pythia PRF service. Full version of this paper. http://pages.cs.wisc.edu/~ace/papers/pythia.pdf.

[26] Warwick Ford and Burton S. Kaliski, Jr. Server-assisted generation of a strong secret from a password. In *International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*. IEEE, 2000.

[27] Michael J Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In *Theory of Cryptography*. Springer, 2005.

[28] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and t-PAKE in the password-only model. In *Advances in Cryptology–ASIACRYPT*. Springer, 2014.

[29] Max Krohn and Chris Coyne. Wrap Wallet. https://keybase.io/warp.

[30] Moses Liskov, Ronald L Rivest, and David Wagner. Tweakable block ciphers. In *Advances in Cryptology–CRYPTO*. Springer, 2002.

[31] Anna Lysyanskaya. Unique signatures and verifiable random functions from the DH-DDH separation. In *Advances in Cryptology–CRYPTO*. Springer, 2002.

[32] Philip MacKenzie and Michael K Reiter. Delegation of cryptographic servers for capture-resilient devices. *Distributed Computing*, 16(4), 2003.

[33] Philip MacKenzie and Michael K Reiter. Networked cryptographic devices resilient to capture. *International Journal of Information Security*, 2(1), 2003.

[34] Philip MacKenzie, Thomas Shrimpton, and Markus Jakobsson. Threshold password-authenticated key exchange. In *Advances in Cryptology–CRYPTO*. Springer, 2002.

[35] R. McMillan. The inside story of Mt. Gox, bitcoin's $460 million disaster. *Wired*, 2014.

[36] Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *Foundations of Computer Science*. IEEE, 1999.

[37] Silvio Micali and Ray Sidney. A simple method for generating and sharing pseudo-random functions, with applications to clipper-like key escrow systems. In *Advances in Cryptology–CRYPTO*. Springer, 1995.

[38] Alec Muffet. Facebook: Password hashing & authentication. Presentation at Real World Crypto, 2015.

[39] Allec Muffet. Facebook: Password hashing and authentication. `https://video.adm.ntnu.no/pres/54b660049af94`.

[40] Moni Naor, Benny Pinkas, and Omer Reingold. Distributed pseudo-random functions and KDCs. In *Advances in Cryptology–EUROCRYPT*. Springer, 1999.

[41] Kenneth G Paterson and Sriramkrishnan Srinivasan. On the relations between non-interactive key distribution, identity-based encryption and trapdoor discrete log groups. *Designs, Codes and Cryptography*, 52(2), 2009.

[42] David Pointcheval and Jacques Stern. Provably secure blind signature schemes. In *Advances in Cryptology–ASIACRYPT*. Springer, 1996.

[43] Jim Roskind. QUIC: Multiplexed stream transport over UDP. *Google working design document*, 2013.

[44] R. Sakai, K. Ohgishi, and M. Kasahara. Cryptosystems based on pairing. In *Cryptography and Information Security*, 2000.

[45] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

A Additional PYTHIA API details

Many PYTHIA-dependent services can benefit from additional API features and calls beyond the primary ones discussed in the body of the paper. (For example, the PYTHIA password onion system in Section 5 uses the Transfer API call.) We detail these other API features in this appendix.

Key-management options. The client can specify a number of options in the call Init regarding management of the ensemble key k_w. The client can provide a contact email address to which alerts and authentication tokens may be sent. (If no e-mail is given, no API calls requiring authentication are permitted at present and no alerts are provided. Later versions of PYTHIA will support other authentication and alerting methods.)

Selector option	Description
Email	Contact email for selector
Resettable	Whether client-requested rotations allowed
Limit	Establish rate-limit per t
Time-out	Date/time to delete k_w
Public-key	Key under which to encrypt and store update and authentication tokens
Alerts	Whether to email contact upon rate limit violation

Figure 9: Optional settings for establishing key selectors in PYTHIA.

Command	Description
$\mathsf{Transfer}(w, w' \, [, options])$	Creates new ensemble w'; outputs update token $\Delta_{w \to w'}$; resets k_w
$\mathsf{SendTokens}(w, authtoken)$	Sends stored update tokens to client
$\mathsf{PurgeTokens}(w, authtoken)$	Purges all stored update tokens for ensemble w

Figure 10: The PYTHIA API. The individual calls are explained in detail in the text.

The client can specify whether k_w should be resettable (default is "yes"). The client can specify a limit on the total number of F_{k_w} queries that should be allowed before resetting $\mathtt{K}[w]$ (default is unlimited) and/or an absolute expiration date and time in UTC at which point $\mathtt{K}[w]$ is deleted (default is no time-out). Either of these options overrides the resettable flag. The client can specify a public key pk_{cl} for a public-key encryption scheme under which to encrypt authentication tokens and update tokens (for Reset, Transfer, as described below, and for master secret key rotations). Finally, the client can request that alerts be sent to the contact email address in the case of rate limit violations. This option is ignored if no contact email is provided. The options are summarized in Figure 9.

PYTHIA also offers some additional API calls, given in Figure 10, which we now describe.

Ensemble transfer. A client can create a new ensemble w' (with the same options as in Init) while receiving an update token that allows PRF outputs under ensemble w to be rolled forward to w'. This is useful for importing a password database to a new server. The PYTHIA service returns an update token $\Delta_{w \to w'}$ for this purpose and stores it encrypted under pk_{cl}. For the case $w' = w$, this call also allows option updates on an existing ensemble w.

Update-token handling. The PYTHIA service stores update tokens encrypted under pk_{cl}, with accompanying timestamps for versioning. The API call SendTokens causes these to be e-mailed to the client, while PurgeTokens causes update-token ciphertexts to be deleted from PYTHIA.

Note that once an update token is deleted, old PRF values to which the token was not applied become cryptographically erased — they become random values unrelated to any messages. A client can therefore delete the key associated with an ensemble by calling Reset and PurgeTokens.

Master secret rotations. PYTHIA can also rotate its master secret key msk to a new key msk'. Recall that ensemble keys are computed as $k_w = \text{HMAC}(msk, \mathrm{K}[w])$, so rotation of msk results in rotation of all ensemble keys. To rotate to a new msk', the server computes k_w for all ensembles w with entries in K, and stores δ_w encrypted under pk_{cl}. If no encryption key is set, then the token is stored in the clear. This is a forward-security issue while it remains, but only for that particular key ensemble. At this point msk is safe to delete. Clients can be informed of the key rotation via e-mail.

Subsequent SendTokens requests will return the resulting update token, along with any other stored update tokens for the ensemble. If multiple rotations occur between client requests, then these can be aggregated in the stored update token for each ensemble. This is trivial if they are stored in the clear (just multiply the new token against the old) and also works if they are encrypted with an appropriately homomorphic encryption scheme such as ElGamal [24].

B Formal Security Analyses

We provide formal security notions for partially oblivious PRFs, and proofs of security relative to them for our scheme from Section 3.

Partially-oblivious PRFs. A partially oblivious PRF protocol $\Pi = (\mathcal{K}, \text{PRF-Cl}, \text{PRF-Srv}, F)$ consists of the following. The key generation algorithm $\mathcal{K}$ outputs a public key and private key pair (pk, sk). We assume that from sk one can compute pk easily. The PRF-Srv algorithm takes input the secret key sk and a client request message (a bit string) and returns a server response message (another bit string). The client algorithm PRF-Cl takes inputs a tweak t and message m, can make a single call to PRF-Srv, and outputs a value. Finally we associate to the protocol a keyed function $F_{sk}:\ \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$. A scheme is correct if executing $\text{PRF-Cl}^{\text{PRF-Srv}_{sk}(\cdot)}(t,m)$ with fresh coins matches $F_{sk}(t,m)$ with probability one. In words, the protocol computes the appropriate function of t, m.

Bilinear pairing setups. Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ be groups all of order p that have associated to them an admissible bilinear pairing $e\colon\ \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. Recall that for generators $g_1 \in \mathbb{G}_1$, $g_2 \in \mathbb{G}_2$, there exists a generator $g_T \in \mathbb{G}_T$ such that $e(g_1^\alpha, g_2^\beta) = g_T^{\alpha\beta}$ for all $\alpha, \beta \in \mathbb{Z}_p$. As shorthand for below we refer to a pairing setup $\mathbb{G} = (g_1, g_2, g_T, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ and assume some compact description of $\mathbb{G}$ as a bit-string where appropriate.

The scheme. The partially-oblivious PRF at the core[8] of our bilinear pairing scheme from Section 3 is as follows for some fixed pairing setup $\mathbb{G}$. Let $H_1:\ \{0,1\}^* \to \mathbb{G}_1$ and $H_2:\ \{0,1\}^* \to \mathbb{G}_2$ be hash functions that we will later model as random oracles.

Key generation $\mathcal{K}$ picks a random exponent sk and computes a public key $pk = g_1^sk$. The PRF-Cl(t,m) algorithm computes a mask $r \leftarrow\!\!\$\ \mathbb{Z}_p$ and sends t and $x = H_2(m)^r$ to the server. The PRF-Srv(sk,t,x) computes $y = e(H_1(t), x)^sk$ and a ZKP π that $\text{DL}_{g_1}(pk) = \text{DL}_{\tilde{x}}(y)$ where $\tilde{x} = e(H_1(t), x)$. It sends pk, y, π to the client, who verifies the ZKP, deletes it, and then outputs $y^{1/r}$. The correctness of the scheme follows from the correctness of the ZKP and the properties of the pairing.

The ZKP is used to ensure that a malicious server responds as per the protocol. In the following security analyses we focus primarily on malicious clients, and for simplicity analyze a simpler version of the protocol that omits the ZKP. The proofs found below can be extended to the full protocol by applying the zero-knowledge security of the proof systems that we use (i.e., use the zero-knowledge simulator to produce fake, but realistic-looking to the client, proofs).

B.1 Unpredictability Security

We define a one-more unpredictability security notion. It modifies one-more unforgeability [42] to be suitable for the setting of unpredictable functions (as opposed to publicly verifiable signatures). The game is shown in Figure 11. We associate to any protocol Π, adversary $\mathcal{A}$, and query number q the one-more-unpredictability advantage defined as

$$\mathbf{Adv}^{\text{om-unp}}_{\Pi,q}(\mathcal{A}) = \Pr\left[\,\text{om-UNP}^{\mathcal{A}}_{\Pi,q} \Rightarrow \mathsf{true}\,\right]\,.$$

The probability here (and for games defined later below) is over all random coins used by the procedures and the adversary. The event refers to the probability that the value returned by the main procedure is true. In words, the definition requires that an adversary cannot produce ℓ outputs of the PRF using less than ℓ queries on partially-blinded inputs to the server. One can easily extend this notion to deal with full blinded inputs as well, but we will not need this.

[8] For brevity we omit key selectors here, and instead focus on analyzing security for a single key instance. Assuming properly generated keys for each selector, one can show that security for a single key instance implies security for many.

```
Game om-UNP$_\Pi^\mathcal{A}$
$(pk, sk) \leftarrow\!\!\$\; \mathcal{K}$
$c \leftarrow 0$
$(t_1, m_1, \sigma_1), \ldots, (t_\ell, m_\ell, \sigma_\ell) \leftarrow\!\!\$\; \mathcal{A}^{\text{PRF-Srv}, H_1, H_2}$
If $\exists i \neq j \;.\; (t_i, m_i) = (t_j, m_j)$ then Ret false
Ret $(\wedge_i(\sigma_i = F_{sk}^{H_1,H_2}(t_i, m_i)) \wedge c < \ell)$

PRF-Srv$(t, Y)$
$c \leftarrow c + 1$
Ret PRF-Srv$_{sk}^{H_1,H_2}(t, Y)$
```

Figure 11: Security game for one-more unpredictability.

This notion of security is sufficient for PYTHIA in applications where the output of the protocol is not stored, but rather used as an unforgeable credential such as with our hardened Brainwallet application (Section 6).

The security of our scheme is based on the following one-more bilinear computational Diffie-Hellman (BCDH) problem, an extension of the one-more CDH assumption given by Boldyreva [10]. To the best of our knowledge this assumption is new, but it is a straightforward adaptation of previous one-more assumptions [6, 10] to our setting. For a pairing setup $\mathbb{G}$, game om-BCDH$_\mathbb{G}$ is defined in Figure 12. In words, the adversary gets a group element $g_1^sk \in \mathbb{G}_1$ as well as target oracles Targ$_1$, Targ$_2$ that return random group elements in $\mathbb{G}_1, \mathbb{G}_2$ respectively. Finally the adversary can query a helper oracle Help that raises $\mathbb{G}_T$ elements to the k. To win, it must compute ℓ values $e(X_i, Y_j)^k$ for ℓ larger than the number of helper queries and each X_i, Y_j a unique pair of (distinct) values returned by the target oracle. Let $\mathbf{Adv}_\mathbb{G}^{\text{om-cdh}}(\mathcal{B}) = \Pr\left[\text{om-BCDH}_\mathbb{G}^\mathcal{B} \Rightarrow \text{true}\right]$.

We have the following theorem establishing the one-more unpredictability of our scheme The proof is essentially identical to the proof of Boldyreva's blind signatures [10].

Theorem 1 *Let Π be the simplified partially oblivious PRF protocol for a pairing setup $\mathbb{G}$ and H_1, H_2 modeled as random oracles. Then for any one-more unpredictability adversary $\mathcal{A}$ making at most q* PRF-Srv *queries, we give in the proof below a one-more CDH adversary $\mathcal{B}$ such that*

$$\mathbf{Adv}_\Pi^{\text{om-unp}}(\mathcal{A}) \leq \mathbf{Adv}_\mathbb{G}^{\text{om-cdh}}(\mathcal{B})$$

where $\mathcal{B}$ runs in time that of $\mathcal{A}$ plus $\mathcal{O}(q)$ group operations.

Proof: We assume without loss of generality that $\mathcal{A}$ never repeats a query to either random oracle and makes a random oracle $H_1(t_i)$ and $H_2(m_i)$ query for each (t_i, m_i, σ_i) triple it outputs. The adversary $\mathcal{B}$ will work as follows when given inputs $\mathbb{G}, X$ and access to oracles Targ$_1$, Targ$_2$, Help. First, it runs $\mathcal{A}$. Whenever $\mathcal{A}$ makes an $H_1(t)$ query, $\mathcal{B}$ queries Targ$_1$ to obtain a $\mathbb{G}_1$-element that we will denote $X[t]$, sets c_t to be the number of H_1 queries so far (including the current), and returns $X[t]$ to $\mathcal{A}$. Whenever $\mathcal{A}$ makes an $H_2(m)$ server query, $\mathcal{B}$ queries Targ$_2$, obtains a $\mathbb{G}_2$-element that we will denote $Y[m]$, sets d_m to be the number of H_2 queries so far (including the current), and returns $Y[m]$ to $\mathcal{A}$. Whenever $\mathcal{A}$ makes a PRF-Srv(t, Y) query, the adversary $\mathcal{B}$ computes $Z \leftarrow e(H_1(t), Y)$, and then queries Z to its helper oracle Help to obtain a value $\sigma \in \mathbb{G}_T$. It returns σ to $\mathcal{A}$.

```
Game om-BCDH$_\mathbb{G}^\mathcal{B}$
$sk \leftarrow\!\!\$\; \mathbb{Z}_p$
$q_h, q_{1,t}, q_{2,t} \leftarrow 0$
$(i_1, j_1, \sigma_1), \ldots, (i_\ell, j_\ell, \sigma_\ell) \leftarrow\!\!\$\; \mathcal{A}^{\text{Targ}_1, \text{Targ}_2, \text{Help}}(\mathbb{G}, g_1^sk)$
If $q_h \geq \ell$ then Ret false
If $\exists \alpha \;.\; (i_\alpha > q_{1,t}) \vee (j_\alpha > q_{2,t})$ then Ret false
If $\exists \alpha \neq \beta \;.\; (i_\alpha, j_\alpha) = (i_\beta, j_\beta)$ then Ret false
Ret $\forall \alpha \;.\; e(X_{i_\alpha}, Y_{j_\alpha})^k = \sigma_\alpha$

Targ$_1$
$q_{1,t} \leftarrow q_{1,t} + 1$ ; $X_{q_{1,t}} \leftarrow\!\!\$\; \mathbb{G}_1$ ; Ret $X_{q_{1,t}}$

Targ$_2$
$q_{2,t} \leftarrow q_{2,t} + 1$ ; $Y_{q_{2,t}} \leftarrow\!\!\$\; \mathbb{G}_2$ ; Ret $Y_{q_{2,t}}$

Help$(Z)$
$q_h \leftarrow q_h + 1$ ; Ret $Z^{sk}$
```

Figure 12: Security game for a one-more BCDH assumption for bilinear pairing setting $\mathbb{G} = (g_1, g_2, g_t, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$.

Eventually $\mathcal{A}$ outputs a series of triples $(t_1, m_1, \sigma_1), \ldots, (t_q, m_q, \sigma_q)$. At this point adversary $\mathcal{B}$ outputs the sequence of pairs $(c_{t_1}, d_{m_1}, \sigma_1), \ldots, (c_{m_q}, d_{m_q}, \sigma_q)$.

Suppose $\mathcal{A}$ wins its game. Then it made at most $q - 1$ queries to PRF-Srv and so $\mathcal{B}$ makes at most $q - 1$ queries to Help. It is also the case that all predictions by $\mathcal{A}$ are for unique tag, message pairs, meaning that $\mathcal{B}$'s output will also be for unique pairs of targets. Finally, it is clear that correct predictions σ_i are also BCDH solutions. ▌

B.2 Pseudorandomness Security

See the full version of this paper [25] for *one-more PRF* security definitions and associated proofs.

EVILCOHORT: Detecting Communities of Malicious Accounts on Online Services

Gianluca Stringhini[§], Pierre Mourlanne[⋆], Gregoire Jacob[‡],
Manuel Egele[†], Christopher Kruegel[⋆], and Giovanni Vigna[⋆]

[§]University College London [⋆]UC Santa Barbara [‡]Lastline Inc. [†]Boston University
g.stringhini@ucl.ac.uk pmourlanne@gmail.com
gregoire@lastline.com megele@bu.edu {chris,vigna}@cs.ucsb.edu

Abstract

Cybercriminals misuse accounts on online services (e.g., webmails and online social networks) to perform malicious activity, such as spreading malicious content or stealing sensitive information. In this paper, we show that accounts that are accessed by botnets are a popular choice by cybercriminals. Since botnets are composed of a finite number of infected computers, we observe that cybercriminals tend to have their bots connect to multiple online accounts to perform malicious activity.

We present EVILCOHORT, a system that detects online accounts that are accessed by a common set of infected machines. EVILCOHORT only needs the mapping between an online account and an IP address to operate, and can therefore detect malicious accounts on any online service (webmail services, online social networks, storage services) regardless of the type of malicious activity that these accounts perform. Unlike previous work, our system can identify malicious accounts that are controlled by botnets but do not post any malicious content (e.g., spam) on the service. We evaluated EVILCOHORT on multiple online services of different types (a webmail service and four online social networks), and show that it accurately identifies malicious accounts.

1 Introduction

Online services, such as online social networks (OSNs), webmail, and blogs, are frequently abused by cybercriminals. For example, miscreants create fake accounts on popular OSNs or webmail providers and then use these accounts to spread malicious content, such as links pointing to spam pages, malware, or phishing scams [27, 31, 40]. A large fraction of the malicious activity that occurs on online services is driven by *botnets*, networks of compromised computers acting under the control of the same cybercriminal [9].

Leveraging existing services to spread malicious content provides three advantages to the attacker. First, it is easy to reach many victims, since popular online services have many millions of users that are well connected. In traditional email spam operations miscreants have to harvest a large number of victim email addresses (on the web or from infected hosts) before they can start sending spam. On online services such as OSNs, on the other hand, cybercriminals can easily find and contact their victims or leverage existing friends of compromised accounts [15]. In some cases, such as blog and forum spam, cybercriminals do not even have to collect a list of victims, because their malicious content will be shown to anybody who is visiting the web page on which the spam comment is posted [21,31]. A second advantage of using online services to spread malicious content is that while users have become aware of the threats associated with email, they are not as familiar with scams and spam that spreads through other communication channels (such as social networks) [5, 18, 27]. The third advantage is that while online services have good defenses against threats coming from the outside (e.g., emails coming from different domains), they have a much harder time detecting misuse that originates from accounts within the service itself (e.g., emails sent by accounts on the service to other accounts on the same one) [28].

To carry out malicious campaigns via online services, attackers need two resources: *online accounts* and *connection points*. Almost all online services require users to sign up and create accounts before they can access the functionality that these services offer. Accounts allow online services to associate data with users (such as emails, posts, pictures, etc.), and they also serve as a convenient way to regulate and restrict access. Connection points are the means through which attackers access online accounts. They are the devices (hosts) that run the client software (e.g., web browsers or dedicated mobile applications) that allow the miscreants to connect to online services. Often, connection points are malware-infected machines (bots) that serve as a convenient way for the attacker to log into the targeted service and issue the necessary commands to send spam or harvest personal information of legitimate users. However, malicious connection points do not need to be bots. They can also be compromised servers, or even the personal device of a cybercriminal.

In this paper, we propose EVILCOHORT, a novel approach that detects accounts on online services that are controlled by cybercriminals. Our approach is based on the analysis of the interactions between attackers and an online service. More precisely, we look at the interplay between accounts, connection points, and actions. That is, we observe which account carries out what action, and which connection point is responsible for triggering it.

The intuition behind our approach is that cybercriminals use online services differently than regular users. Cybercriminals need to make money, and this often requires operations at a large scale. Thus, when such operations are carried out, they involve many accounts, connection points, and actions. Moreover, accounts and connection points are related in interesting ways that can be leveraged for detection. A key reason for these interesting relationships is the fact that attackers use bots (as connection points) to access the online accounts that participate in an orchestrated campaign. By linking accounts and the connection points that are used to access these accounts, we see that malicious *communities* emerge, and these communities can be detected.

EVILCOHORT works by identifying communities (sets) of online accounts that are all accessed from a number of shared connection points (we use IP addresses to identify these connection points). That is, we observe a number of IP addresses and accounts, and each account is accessed by a non-trivial portion of these IP addresses. Typically, these IP addresses correspond to bot-infected machines, and they are used to log into the accounts that are under the control of the attacker. To identify communities, we consume a log of *interaction events* that the online service records. An interaction event can be any action that a user performs in relation to an account on an online service, such as logging in, sending an email, or making a friend request. Each event also contains the account that is involved, as well as the IP address that sends the request. Our results show that the overwhelming majority of accounts that are identified by our community detection approach are actually malicious, and that therefore the detection by EVILCOHORT is reliable enough on its own. As an additional step to better understand the detected communities and help us assess potential false positives we present techniques to analyze the characteristics of accounts within a community and identify typical behaviors that are indicative of malicious activity. Such characteristics include suspicious activity frequencies over time, synchronized activity of the accounts in the community, and the distribution of the types of browsers used by the infected machines to connect to the online accounts.

One key advantage of our approach is that it is generic, as it does not rely on service-specific information. This is different from previous research, which typically leverages service-specific information to perform detection. For example, BOTGRAPH [39] looks at accounts that are accessed by multiple IP addresses, similarly to our approach, but relies on heuristics based on the email-sending behavior of such accounts to limit false positives. This fact not only makes deployment more problematic, but also limits the applicability of the system to accounts that are misused to send spam. Contrast this with our broad definition of interaction events that is satisfied by a large variety of data that naturally accumulates at online service providers, and makes our approach applicable to any online service that requires users to create an account to interact with it. We demonstrate this by leveraging our approach to detect spammers on a webmail service, as well as to identify malicious accounts on multiple OSNs.

An additional advantage of our approach is that it can be applied to different types of actions. These actions can include account generation and login operations. In these cases, it might be possible to detect malicious accounts before they distribute any malicious content, as an early warning system. Also, it can help to identify abuses where no malicious content is distributed at all. An example of this are botnets that use social networks as part of their command-and-control (C&C) infrastructure [26], or botnets that crawl the online profiles of users harvesting personal information [17]. To show the versatility of our approach, we apply it to two different types of interaction events: on the webmail service we look at events that correspond to the sending of emails, while on the OSNs an interaction event is recorded when a user logs into her account. Over a period of five months, EVILCOHORT detected more than one million online accounts as malicious on the analyzed services. In summary, this paper makes the following contributions:

- We show that a significant amount of malicious activity is carried out by accounts that form communities (when looking at the connection points that access them). We also find that these accounts tend to remain active for extended periods of time on a large webmail provider.
- We present EVILCOHORT, a novel approach to detect malicious communities (and hence, accounts controlled by cybercriminals) on online services. This approach works by detecting accounts that are accessed by a common, shared set of IP addresses.
- We evaluated EVILCOHORT on datasets of different types of interactions collected on five different online services. Over a period of five months, EVIL-

Cohort detected more than one million accounts used to perform malicious activities. We show that EvilCohort is effective in detecting malicious communities regardless of the type of accounts analyzed, making it a valuable tool to protect a variety of online services.

2 Motivation: Analysis of Malicious Activity on a Webmail Service

We want to understand the way in which cybercriminals abuse accounts on online services, to identify weak points that we could leverage for detection. To this end, we observed the email-sending activity on a large webmail service. Our dataset was composed of the emails generated by 21,387,006 distinct online accounts over a period of one day. In total, this dataset contained 72,471,992 emails. We call the dataset containing information about this email-sending activity **T**. For each email-sending event, the dataset **T** contains the IP address that accessed the account, the user ID of the account that sent the email, and a timestamp. In addition, each email-sending event contains information on whether the email was considered as spam by the webmail provider or not. Note that the dataset **T** only contains information about sent emails, and provides no insights on the number of times an account is accessed without sending any email (e.g., to check the account's inbox).

Two Types of Malicious Accounts. We analyzed the accounts that sent spam in the dataset **T**. We identify two types of malicious accounts:

1. *Accounts that are used in isolation*. Each account is accessed by a single IP address, which could be the attacker's computer or a single infected machine.
2. *Accounts that are accessed by multiple IP addresses*. The same account is accessed by multiple infected computers.

We looked at how many malicious accounts of each type are active on the webmail service. For this analysis we considered an account as malicious if the account sent at least 10 emails during the day under consideration, and the majority of these emails were flagged as spam by the webmail provider. We selected this threshold because we needed a set of "labeled" accounts that sent spam on the webmail provider. Picking accounts whose majority of emails was flagged as spam by the email provider gives us confidence that this dataset does not contain false positives. Note that this preliminary analysis was purely qualitative, and it was used to give us an idea on the behavior of malicious accounts on a webmail service. We call this set of labeled accounts **L**. In total, **L** is composed of 66,509 malicious accounts that were accessed by a single IP address, and 103,918 malicious accounts that were accessed by two or more.

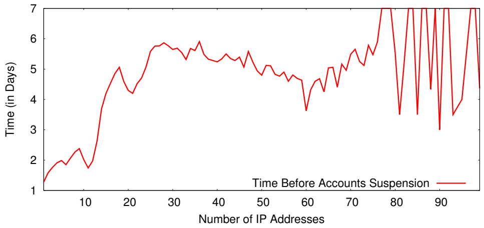


Figure 1: Average time (in days) before a spamming account was suspended in **L**, given the number of IP addresses accessing that account.

Accounts Shared by Many IP Addresses Are More Dangerous. We then investigated the effectiveness of the two identified types of spam accounts in sending emails, and their ability to evade detection by the webmail provider. With detection, we mean triggering a mechanism on the webmail provider that leads to the account being suspended. Figure 1 shows the average time (in days) that it took for a malicious account in **L** to be suspended after it sent the first spam email, given the number of IP addresses that accessed that account. As it can be seen, accounts that are used in isolation have a shorter lifespan than the ones that are used by multiple IP addresses: accounts that are only accessed by a single IP address are typically detected and suspended within a day, while ones that are accessed by many different IPs can survive for as long as a week.

We then studied the difference in the activity of the two types of accounts with regards to the number of spam emails sent. Figure 2 shows that accounts that are used in isolation are less effective for cybercriminals, as they send a smaller number of emails per day before being shut down. Alternatively, attackers can have each of their infected computers send a small number of emails and stay under the radar. Figure 3 shows that IP addresses accessing accounts used in isolation send 19 emails per day on average before being blocked, while having multiple computers accessing the same account allows cybercriminals to have each IP address send a lower number of emails, as low as one email per IP address in some cases. The longevity of the accounts that are accessed by more than one IP address suggests that the webmail service lacks effective countermeasures to prevent abuse of the service by such accounts. We acknowledge that this could be due to shortcomings in the countermeasures deployed by this particular webmail service, but it still shows us that accounts that are accessed by a multitude of infected computers are a problem for online services.

Detecting Malicious Accounts Shared by Many IP Addresses. Can we use the fact that malicious accounts tend to be accessed by many IP addresses to flag these

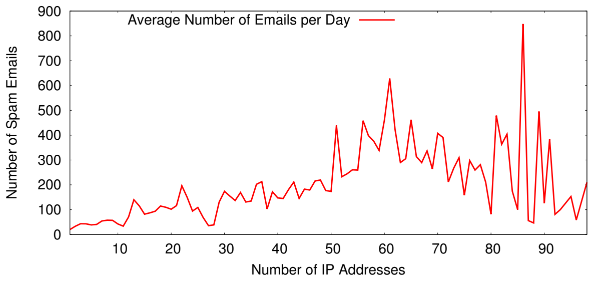


Figure 2: Average number of spam emails sent per day per *account* accessed by a certain number of IP addresses.

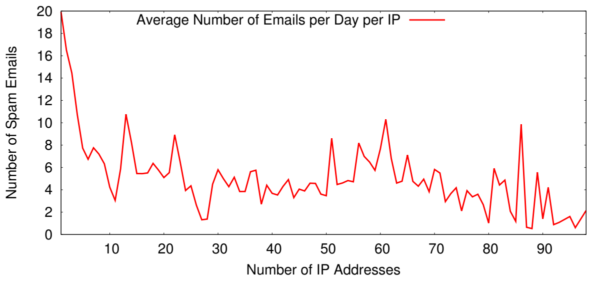


Figure 3: Average number of spam emails sent per day per *IP address* that accessed a certain account.

accounts as malicious? Unfortunately, the number of IP addresses that accessed an account is not a strong enough indicator and basing a detection system only on this element would generate a number of false positives that is too high for most practical purposes. For example, considering as malicious accounts that were accessed by two or more IP addresses in **T** would cause 77% of the total detections to be false positives (i.e., accounts that did not send any spam email). This makes sense, because many users access their webmail account from different devices, such as a mobile phone and a desktop computer. Even looking at accounts accessed by a higher number of IP addresses does not solve the false positive problem: looking at accounts that were accessed by ten or more distinct IP addresses in **T** 32% would be labeled malicious incorrectly (i.e., false positives); by increasing the number of required IP addresses false positives decrease, but they remain well above the level considered acceptable in a production environment.

To overcome the false positive problem, we leverage another property of cybercriminal operations that use online services: cybercriminals can only count on a limited number of infected machines (bots) [26], as well as a limited number of accounts on the online service. Because of this limitation, and to make their operations more resilient to takedowns, cybercriminals have multiple bots connect to the same set of accounts over time. We can think of a set of accounts that are accessed by the same set of bots as a *community*. In the following, we present EVILCOHORT, a system that detects communities of accounts that are accessed by a common set of IP addresses. We show that, by looking at these communities of accounts, we can detect most of the malicious accounts that are accessed by multiple IP addresses, while generating a false positive rate that is orders of magnitude lower than just looking at accounts in isolation. In Appendix 5.2, we compare the two methods in detail, and show that EVILCOHORT outperforms the method that looks at individual accounts only.

3 EVILCOHORT: Overview

EVILCOHORT operates on inputs in the form of *account interaction events*. Users create their own accounts and connect to online services to perform a number of actions. Depending on the service, these actions range from sending messages to the user's friends and colleagues, to performing friend requests, to browsing pictures, to updating the user's profile. Accounts allow the online service to attribute any activity performed to a specific user, in a more precise way than source IP addresses do. For instance, it is possible to correctly attribute the activity of a certain user regardless of the place she is connecting from (her home computer, her office, or her mobile phone). We define a user interaction with an online service as a tuple

$$\mathbf{A} = < H, U, T >,$$

where H is the host that the user is connecting from (identified by an IP address), U is her user ID on the online service, and T is a timestamp.

Approach Overview. EVILCOHORT works in three phases. First, it collects interaction events from the monitored online service, and builds a bipartite graph where one set of vertices is the online accounts observed and the other set of vertices is the list of IP addresses that accessed them. Then, it computes the weighted one-mode projection of the bipartite graph onto the account vertex set. The result of this phase is a graph, which we call *projected graph representation*, in which the vertices are the accounts and the edge labels (i.e., weights) indicate how many shared IP addresses connected to each pair of accounts. As a third phase, EVILCOHORT performs clustering on the projected graph representation to find communities of online accounts that were accessed by a common set of IP addresses. A last, optional step consists of analyzing the discovered communities, to characterize them and possibly identify security relevant activity, such as campaigns. In the remainder of this section, we provide more details about the three steps involved in identifying communities.

3.1 Data Collection

In the first phase, EVILCOHORT collects interaction events on an online service for a given observation period (a day in our current implementation). Based on these interaction events, EVILCOHORT builds a bipartite graph where the first set of vertices **A** are the online accounts that generated the events, while the second set of vertices **I** are the IP addresses that accessed these accounts. An account vertex has an edge to an IP address vertex if that account was accessed by that IP address. We call this bipartite graph $\mathbf{G}_A$.

3.2 Building the Projected Graph Representation

We expect that cybercriminals instruct their bots to connect to multiple accounts under their control. As discussed in Section 2, this is because they have control of a limited number of bots and want to optimize the effectiveness of their malicious operation. For this reason, we represent the relation between online accounts and IP addresses as a weighted graph. To this end, we perform the weighted one-mode projection of the bipartite graph $\mathbf{G}_A$ onto the account vertex set **A**. More precisely, we define the *projected graph representation* of the set of accounts **A** as

$$\mathbf{R} =< \mathbf{V}, \mathbf{E} >,$$

where each element in the set of vertices **V** is one of the accounts in **A**, and the set of edges **E** is weighted as follows: for each pair of accounts u_1, $u_2 \in \mathbf{V}$, the edge connecting them has a weight equal to the number of IP addresses that u_1 and u_2 share, based on the bipartite graph $\mathbf{G}_A$. If the accounts u_1 and u_2 do not share any IP address, there is no edge between them.

As we showed in Section 2, many legitimate accounts are accessed by more than one IP address. To focus on detecting communities of accounts that share a higher number of IP addresses, we filter the bipartite graph $\mathbf{G}_A$ on the in-degree of the accounts in **A**. More precisely, we introduce a threshold s, and consider as inputs for the projection only those accounts that have a degree higher than s, which means that they were accessed by more than s IP addresses during the observation period. Since the number of IP addresses that legitimate accounts share is low, communities of accounts sharing many IP addresses are suspicious. We investigate the possible choices for the threshold s in Section 5.1. By increasing the value of s we can reduce false positive considerably, but we also reduce the number of accounts that EVILCOHORT can detect as malicious. The graph **R** is then passed to the next phase of our approach, which finds communities of online accounts that are accessed by a common set of IP addresses.

3.3 Finding Communities

After obtaining the projected graph representation **R**, we identify communities of accounts. To this end, we use the "*Louvain Method*" [6]. This clustering method leverages an iterative algorithm based on modularity optimization, and is particularly well-suited to operate on sparse graphs, as most graphs obtained from "real life" situations are [12]. In their paper, Blondel et al. [6] show that their method outperforms several community-detection algorithms that are based on heuristics.

The Louvain method operates in two steps, which are iteratively repeated until convergence is reached. At the beginning, each vertex in **R** is assigned to its own community of size one. Each iteration of the algorithm proceeds as follows:

1. For each account u_1 in **U**, we consider each of its neighbors u_2, and we calculate a gain value g that represents the effect of removing u_1 from its community and adding it to u_2's community. We explain how we calculate g later in this section. If any of the gain values g is positive, we move u_1 to the community of the account that yields the highest gain.
2. We rebuild the graph **R**, whose nodes are now the communities built during the previous step. Each edge between two communities c_1 and c_2 is weighted with the number of IP addresses that are shared between the two communities.

The algorithm repeats these two steps until convergence. Blondel et al. [6] describe how the gain value g is calculated in detail. In a nutshell, the gain obtained by moving an account i to a community C is

$$g_{in} = [\frac{\sum_{in} + k_{i,in}}{2m} - (\frac{\sum_{tot} + k_i}{2m})^2] - [\frac{\sum_{in}}{2m} - (\frac{\sum_{tot}}{2m})^2 - (\frac{k_i}{2m})^2],$$

where $\sum_{in}$ is the sum of the weights of the edges between the accounts in C, $\sum_{tot}$ is the sum of the weights of the edges incident to the accounts in C, k_i is the sum of the weights of the edges incident to i, $k_{i,in}$ is the sum of the weights of the edges that connect i to the accounts in C, and m is the number of edges in **R**. Blondel et al. show how a similar weight is calculated for the gain obtained by removing an account i from its community (g_{out}) [6]. If the sum of the two gains g = $g_{in} + g_{out}$ is positive, the account i gets added to the community C.

3.4 Optional Step: Characterizing Communities

As an optional step, after the detection of (malicious) communities, we propose a number of techniques that extract interesting properties of these communities. These properties allow the operator of EVILCOHORT to easily characterize security-relevant behaviors of the communities. As we will see later, these properties can be useful to both assess false positives and identify whether the accounts in a community are fake (i.e., Sybils) or compromised accounts that are accessed both by cybercriminals and by their legitimate owners. To gain insight into the behavior of accounts associated with communities these properties can incorporate auxiliary sources of information that are not strictly part of the collected *account interaction events* (e.g., web-browser user agents).

User agent correlation. Regular users of online services likely connect to their accounts from a limited set of devices corresponding to a largely consistent set of connection points. During the course of a day, a typical user would access, for example, her account from home using her personal browser, then from work using the browser mandated by company policy, and finally from her phone using her mobile client. In other words, we expect to have a one-to-one relation between the connection points and the client programs (agents) that run on these machines and are used to perform the activity. When online services are accessed via the web, the client used to perform the activity can be identified by the HTTP user agent field. Proprietary clients often have similar attributes that can be used for this purpose. On iOS, for example, the system-provided HTTP library uses the application's name and version as the user agent string.

For malicious communities, the activity is no longer generated by humans operating a browser. Instead, the activity is frequently generated by autonomous programs, such as bots, or programs used to administer multiple accounts at once. These programs can be designed to use either hard-coded user agent strings, or, as we observed in recent malware, slight variations of legitimate user agent strings. Presumably, this is a technique aimed at evading detection mechanisms. However, these practices significantly change the distribution of user agent strings and their corresponding connection points.

To measure the correlation between connection points and user agents within a community c, we compute the following ratio:

$$log(c) = log\left(\frac{\text{number of user agents}}{\text{number of IP addresses}}\right) \qquad (1)$$

For a typical benign user, the correlation is very strong because there is a one-to-one relationship between connection point and user agent: That is, each connection point is associated with a different user agent, and as a result $log(c)$ tends towards 0. For malicious communities, where the relationship becomes one-to-n, negative values will be observed in case of hard-coded user agent strings, and positive values in case of permutations of the user agent strings. Note that we exclude from the computation user agent strings coming from mobile phones or tablets because these mobile devices can be connected to any network, meaning that no correlation can be expected in this case.

Event-based time series. This property captures *account interaction events* in a community over time. Time series represent the frequency of events per time period. As we will show in Section 6, the time series representations fundamentally differ for legitimate accounts and those in malicious communities. Time series for legitimate users commonly contain daily activity patterns depending on the night and day cycle. Furthermore, weekly patterns can often be identified too. Automated malicious activity, however, commonly results in either highly regular activity (e.g., botnets using the online service as their command and control service), or irregular bursts (e.g., during the execution of a spam campaign).

IP address and account usage. This analysis, similarly to the previous one, relies on timing analysis. The main difference is that events are no longer aggregated for the entire community but, instead, individual IP addresses and accounts are represented separately over time.

The IP addresses usage graph is generated in the following way: Time is represented on the x-axis, and each unique IP address is represented by a separate entry on the y-axis of the graph. Events are then plotted as points in the graph using this set of coordinates. The account usage graph is generated in a similar way, with unique accounts instead of IPs on the y-axis. We will show an example for IP address and account usage graphs in Section 6. Similar to the above time series representation, malicious communities show a high degree of synchronization, which is not present for communities formed by legitimate users. This observation has been confirmed by independent research that has been recently published [7]. Using this type of representation, any suspicious alignment in the events recorded for different IP addresses or different accounts can easily be identified.

Automated post-processing. In our current implementation we exclusively use the analysis of community properties to infer interesting characteristics of identified communities. Our analysis indicates that communities formed by malicious accounts exhibit vastly different characteristics than those formed by legitimate accounts, as we show in Section 6. Thus, the techniques described in this section could also be used to automatically distinguish malicious from legitimate communities. Similar to Jacob et al. [17], detection could be implemented based on automated classifiers working on statistical features characterizing the shape of a time series or plot. While such an automated post-processing approach would be a potential avenue for reducing the false positives of EVILCOHORT even further, our current false positives are already well within the range where a deployment would benefit an online service. An implementation of this automated post-processing step is, thus, left for future work.

4 Description of the Datasets

In Section 2, we analyzed **T**, a labeled dataset of email-sending events on an webmail provider. Since EvilCohort only takes into account the mapping between IP address and online account of an event, however, it can operate on any online service that allows users to create accounts. Such services include web-based email services, online social networks, blogs, forums, and many others. In addition, EvilCohort can operate on activities of different type, such as login events, message postings, message shares, etc. To show the versatility of our approach, we evaluated it on multiple datasets of activities on five different online services. The first dataset is composed of email sending events logged by a large webmail service, with similar characteristics to **T**. The second dataset is composed of login events recorded on four different online social networks. In the following, we describe these datasets in more detail.

4.1 Webmail Activity Dataset

Our first dataset is composed of email-sending events logged by a large webmail provider. Every time an email is sent, an activity is logged. We call this dataset $\mathbf{D}_1$. Note that the email-sending events in this dataset are generated by accounts on the webmail service, which send emails to other email addresses.

The dataset $\mathbf{D}_1$ contains the events logged over a five-month period by the webmail provider for a subset of the accounts that were active on the service during that period. In total, this dataset contains 1.2 billion email-sending events, generated by an average of 25 million accounts per day. This data was collected according to the webmail provider's terms of service, and was only accessed on their premises by a company's employee. Beyond the above-discussed information, no further email related information was accessible to our research team (i.e., no content, no recipients). In addition to the activity events, the webmail provider logged whether the email was flagged as spam by their anti-spam systems. The dataset **T** presented in Section 2 is a subset of $\mathbf{D}_1$. We used **T** to study in-depth the properties of legitimate and malicious accounts on a webmail service (see Section 2). As we explained in Section 2, **T** is a dataset containing email events observed on a large webmail provider over a period of one day, while **L** contains the accounts in **T** that are heavy senders of spam (meaning that they sent 10 or more emails during the day of observation, and that a majority of these emails was detected as spam by the defenses in place at the webmail provider).

It is worth noting that **L** does not contain all the accounts that sent spam in **T**. Such ground truth does not exist, because if a perfect detection system existed we would not need new approaches such as EvilCohort. Instead, **L** contains a set of "vetted" spam accounts that were detected by the webmail provider, and using this dataset as a reference allows us to get a good idea of how well EvilCohort works in detecting previously-unseen malicious accounts on online services.

4.2 Online Social Network Login Dataset

Online Social Network	**OSN$_1$**	**OSN$_2$**	**OSN$_3$**	**OSN$_4$**
Login events	14,077,316	311,144	83,128	42,655
Unique Accounts	6,491,452	16,056	25,090	21,066
Unique IPs	6,263,419	17,915	11,736	4,725
Avg. daily events	2,067,486	51,832	11,897	6,601
Account singletons	74.6%	40.0%	51.7%	72.2%

Table 1: Statistics of activity events of the dataset $\mathbf{D}_2$.

Our second dataset is composed of login events collected from four different OSNs, spanning a period of 8 days. We call this dataset $\mathbf{D}_2$. We obtained the dataset $\mathbf{D}_2$ from a security company. For each activity event, the dataset contained additional information such as the user agent of the web browser performing the login and the HTTP headers of the response. Sensitive information such as the user IDs and the IP addresses was anonymized. Note that this does not affect our community detection algorithm at all.

Statistics on the number of login events for each social network can be found in Table 1. These statistics reflect the size and activity observed on these networks, ranging from tens of thousands up to 14 million login events. One interesting observation is the high percentage of account singletons on a daily basis, i.e., the percentage of users connecting at most once a day. On a weekly basis, the percentage tends to drop but remain surprisingly high. These users are probably legitimate users that are not very active on the social network.

5 Evaluation

In this section, we analyze how EvilCohort performs in the real world. We first study the effectiveness of our approach by using the dataset **T** and its subset **L** of labeled malicious accounts. We then select a suitable threshold s that allows us to have a small number of false positives. Finally, we run EvilCohort on multiple real-world datasets, and we analyze the communities of malicious accounts that we detected.

Value of s	# of accounts	# of communities	Known accounts in L	Additional detections over L	FP communities	FP accounts
			(% of tot. accounts in L)	(% of add. detections over L)	(% of tot. communities)	(% of tot. accounts)
2	135,602	3,133	94,874 (58%)	40,728 (23.8%)	1,327 (42%)	12,350 (9.1%)
5	77,910	1,291	51,868 (30.4%)	26,042 (15.2%)	580 (44.9%)	2,337 (3%)
10	25,490	116	16,626 (9.7%)	8,864 (5.2%)	48 (41.3%)	433 (1.7%)
65	1,331	6	1,247 (0.7%)	84 (0.04%)	0	0

Table 2: Summary of the results reported by EvilCohort for different values of the threshold s.

5.1 In-degree Threshold Selection

As with every detection system, EvilCohort has to make a trade-off between false negatives and false positives. As we mentioned in Section 3.2, we can adjust the value of the minimum in-degree for account vertices that we use to generate the one-mode projection graph **R** to influence the quality of EvilCohort's results. We call this threshold s. In particular, increasing the value of s decreases the number of false positives of our system, but also reduces the number of accounts that can be detected. That is, any account that is accessed by less than s IP addresses during an observation period is excluded from evaluation for community membership, and thus cannot be detected as malicious by EvilCohort.

In this section we run EvilCohort on the datasets **T** and **L** and analyze the quality of its results. The goal is to identify a suitable value of s for running EvilCohort in the wild. Recall that **L** is the set of accounts that were classified as malicious as explained in Section 2. In the absence of complete ground-truth, we use **L** as a partial ground-truth to help us assess how well EvilCohort operates.

The first element that we use to evaluate the effectiveness of EvilCohort is the fraction of accounts in **L** that our system is able to detect. Ideally, we want EvilCohort to detect a large fraction of our labeled malicious accounts. Unfortunately, as discussed above, increasing the value of s decreases the number of accounts that EvilCohort can possibly detect. The percentage of malicious accounts in **L** detected by EvilCohort provides us with an estimate of the false negatives that EvilCohort would report if it was run in the wild.

As a second element of effectiveness, we look at the set of accounts that EvilCohort detects as malicious in **T**, but that were missed by the anti-spam systems deployed by the webmail provider. These are malicious accounts *not* in **L**. We refer to this number as *additional detections*. This value gives us an estimate on the overall effectiveness of EvilCohort. Ideally, we want this number to be high, so that if EvilCohort were to be deployed in conjunction with the defenses that are already in place on the online service, it would increase the number of malicious accounts that can be detected and blocked.

The third element that we consider is the *confidence* that the communities detected by EvilCohort are indeed malicious. To this end, we look at the fraction of accounts in **L** that are present in each detected community. We consider a community as malicious (i.e., a true positive) if at least 10% of the accounts belonging to it are part of our labeled dataset of malicious accounts. Otherwise, we consider it as a false positive of EvilCohort. We empirically found that this 10% fraction of vetted bad accounts gives us a good confidence that the communities are indeed malicious. Recall that **L** is a dataset composed of "repeated offenders." In other words it contains accounts that have a consistent history of sending spam, therefore having a small fraction of accounts from this set in a community is a strong indicator of the entire community being malicious. As we show in Section 5.3, if we relax the method that we use to assess true positives (for example we consider an account as malicious if it sent a single email flagged as spam by the webmail provider) then the majority of the accounts in communities detected by EvilCohort are confirmed as malicious. In Section 6 we show that by observing additional properties of the communities detected by EvilCohort we are able to confirm almost the totality of them as malicious.

Table 2 provides a summary of the results that we obtained when running EvilCohort on **T**, based on different values of the threshold s. As one can see, the fraction of accounts in **L** that our system detects decreases quickly as we increase s. With a threshold of 2, EvilCohort only detects 58% of the labeled accounts.With a threshold of 10 the fraction of accounts in **L** that are covered is only 10%. Once we reach higher thresholds, the fraction of detected accounts that are part of **L** becomes very small. The additional detections performed by EvilCohort over the webmail provider's detection system also decrease as we increase s. With a threshold of 2 we detect 23% malicious accounts that existing approaches miss. A threshold of 10 still ensures 5.5% additional detections over the dataset **L**. False positives decrease rapidly as we increase s as well. Setting s to 2 results in 9% false positives. A threshold of 10 reduces false positives to 1.7%. By setting s to 65, EvilCohort does not mistakenly flag any legitimate account as malicious. Unfortunately, the number of detections at this threshold is quite low.

Given the results reported in this section, we decided to use 10 as a value of s for our experiments. At this threshold false positives are low (1.7%), but the system is still able to significantly improve the detections performed by the existing countermeasures deployed by the webmail provider, and detects 116 communities.

It is interesting to notice that although the number of accounts misclassified by EVILCOHORT is generally low, percentages are higher when looking at communities: with a threshold of 10, for example, 40% of the detected communities are considered to be false positive accounts. Interestingly, however, the size of true positive and false positive communities varies consistently: false positive communities are composed of 9 accounts on average, while malicious ones are composed of 370 or more accounts. An explanation for this is that small communities could be a side effect of computers behind a NAT which constantly change their IP address through DHCP. As previous research showed, some ISPs change the IP address of their customers very often [26]. The small size of such communities, however, shows that filtering on the number of accounts in a community could be an effective filter to further reduce false positives. We did not include a threshold on the size of a community in EVILCOHORT because we wanted to keep the system general. However, in a production setting an additional threshold on the community size could be a straight-forward way to reduce false positives even further. In addition, Section 6 illustrates that false positive communities expose distinct behavioral characteristics. These characteristics can be leveraged to further reduce false positives.

5.2 Comparison between EVILCOHORT and the Single Account Method.

As we discussed in Section 2, EVILCOHORT outperforms detection approaches that look at single accounts accessed by a high number of IP addresses by orders of magnitude. At a threshold of 10, where EVILCOHORT reports a false positive rate of 1.7%, the single-account method has a false positive rate of 32%. Even by dramatically increasing the threshold, the number of false positives of the single-account method remains high. At a threshold of 65, at which EVILCOHORT reports no wrong detections, the single-account method has a false positive rate of 1.2%. Even at a threshold of 100, the single-account method has a small number of false positives.

The last question to answer is whether accounts that are accessed by a high number of IP addresses do form communities, in other words whether EVILCOHORT is able to detect most malicious accounts that were accessed by a number of IP addresses s. To answer this question, we looked at single accounts accessed by a number of IP addresses n (from one to 100), and labeled them as malicious or benign in the same way we labelled the communities in the previous experiment. We then proceeded as follows: for each value of n, we considered the single-account method to have perfect recall (i.e., no false negatives). We then looked at how many of the accounts detected by this method would have formed communities, and therefore be detected by EVILCOHORT. The fraction of malicious accounts that form communities is generally very high. With a threshold of 10, EVILCOHORT detected 93% of the malicious accounts detected by the single-account method. With a threshold of 20 this fraction becomes 95%, while with a threshold of 50 it becomes 98%. We conclude that the vast majority of accounts accessed by a high number of IP addresses form communities, and that therefore EVILCOHORT is a suitable alternative to the single-account method for what concerns false negatives, and it reduces false positives by orders of magnitude compared to the single account method.

5.3 Detection in the Wild

We applied EVILCOHORT to the datasets $\mathbf{D}_1$ and $\mathbf{D}_2$. In the following, we show that EVILCOHORT is able to detect a large number of malicious online service accounts, regardless of the type of online service that it is run on.

Detection on the webmail activity dataset. The dataset $\mathbf{D}_1$ is composed of email-sending activities logged on a large webmail provider. Over a period of 5 months, EVILCOHORT detected **M** = 1,217,830 accounts as malicious. In total, these accounts were part of 17,803 malicious communities.

We first wanted to understand the number of false positives generated by EVILCOHORT in the wild. We tried to answer this question from two vantage points. First, we performed the same false positive analysis explained in Section 5.1. That is, we considered an account to be vetted as malicious if the majority of the emails sent by it during the day of observation were detected as spam by the webmail operator. We then considered a community as a false positive by EVILCOHORT if less than 10% of the accounts in the community belonged to our set of vetted malicious accounts. In total, we found 23,269 accounts to be potential false positives (1.9% of the total accounts in **M**). This is in line with the validation results from Section 5.1, in which we reported 1.7% false positives by using the same threshold. As a second method of assessment, we manually analyzed 100 randomly picked communities among the ones detected by EVILCOHORT. For each of these communities we could identify signs of automated activity and possible maliciousness. For example, the user IDs of the accounts used by some communities had been clearly automatically generated: in one case, all the accounts belonging to the community were composed of two dictionary words concatenated with a four-digit number. In another case, all the user IDs were 20-letter random alphanumeric characters. In an-

Day	1	2	3	4	5	6	7	8
OSN_1	24	30	6	4	4	4	5	2
OSN_2	1	0	0	0	0	0	0	0
OSN_3	0	0	0	0	1	1	1	0
OSN_4	0	0	0	0	0	0	0	0

Table 3: Number of malicious communities detected per day by EvilCohort on the dataset $\mathbf{D}_2$.

Social Network	OSN_1	OSN_2	OSN_3	OSN_4
Accounts (Avg)	3,662	2	2	0
Accounts (Med)	13	2	2	0
Accounts (Max)	66,764	2	2	0
IPs (Avg)	2,381	14	10	0
IPs (Med)	19	14	10	0
IPs (Max)	3,9884	14	10	0

Table 4: Size of the malicious communities detected by EvilCohort on the dataset $\mathbf{D}_2$. Numbers (Average, Median and Maximum) are expressed per community.

other case, the accounts belonging to a community were accounts with a long history of legitimate activity, which suddenly started being accessed by a large, common set of IP addresses. We highly suspect that this community of accounts was composed of legitimate accounts that had been compromised by cybercriminals.

We then wanted to evaluate the false negatives reported by EvilCohort. 94.6% of the vetted malicious accounts used in the false positive analysis at the operating threshold formed communities, and were therefore detected by EvilCohort. The false negatives would therefore account for 5.4% of the total accounts in **M**. This shows that malicious accounts accessed by a large number of IP addresses are typically accessed by botnets, and confirms the usefulness of our approach.

We then looked at how many accounts detected by EvilCohort sent at least one email that was flagged as spam by the webmail provider during the day in which EvilCohort detected them. In total, 715,671 accounts fall in this category (i.e., 59% of **M**). This also shows that relaxing our ground truth assumptions and looking at accounts that sent a single spam email as malicious instead of a vetted dataset results in having the majority of the accounts detected by EvilCohort confirmed by the defenses already in place at the webmail provider. Conversely, EvilCohort proves to be able to grow the set of malicious accounts detected by the webmail provider consistently, since it detected 502,159 additional accounts as malicious (41% of the total).

On average, our prototype implementation of EvilCohort processes one day of data in about ten minutes using a COTS server with 16GB of RAM.

Detection on the social network login dataset. The dataset $\mathbf{D}_2$ is composed of login events from online social networks. In the following, we applied EvilCohort to this second dataset to demonstrate the viability of the approach for different types of services. We used the same threshold as selected in Section 5.1 on this dataset too. Unfortunately, no labeled dataset was available for these experiments. To confirm that the accounts belonging to the identified communities were indeed malicious, we performed a post-processing analysis. We discuss the results of these experiments in Section 6.

Over eight days, EvilCohort was able to detect a total of 83 communities, which represents a total of 111,647 unique accounts. The number of detected communities and the size of these communities evolve daily, as one can see in Table 3 and Table 4. Unsurprisingly, our numbers heavily depend on the size of the social network. OSN_1 is by far the largest network; consequently, this is where we observed the highest number of communities, as well as the largest communities. Interestingly, we observe an important drop in the number of communities on the third day. This might indicate that accounts were taken down by the network. The remaining communities tend to be of smaller size. In OSN_2 and OSN_3, we only detect isolated communities of very small size: two accounts accessed by ten different IP addresses. The activity for OSN_4 was too little to detect any interesting community with the selected threshold.

To understand the evolution of the detected communities, we studied their similarity over time. A community remains stable over time if it is found similar to a community detected the day before. We qualitatively consider two communities as similar if they share more than 50% of their accounts. In OSN_1, one of the largest communities, with more than 66,000 accounts, was stable over five days, with a similarity ranging between 53% to 85%. Two communities of smaller size, with about 10,000 accounts each, were found stable over the two first days but they disappeared on the third day as previously observed. The community detected in OSN_3 is only made up of two accounts and it is stable over three days before being brought down.

6 Application of Post-processing Techniques

As we mentioned in Section 2, EvilCohort was motivated by observations performed on a webmail service. Given the generality of our approach, however, we can apply it to any online service that makes use of accounts. For this reason, we tested EvilCohort on the OSN dataset $\mathbf{D}_2$. The question remains on how well EvilCohort works on such a different dataset. Unfortunately, the dataset $\mathbf{D}_2$ came without ground truth. For this reason, we used the techniques described in Section 3.4 to assess the maliciousness of the detected communities. In a nutshell, we analyze the communities detected by

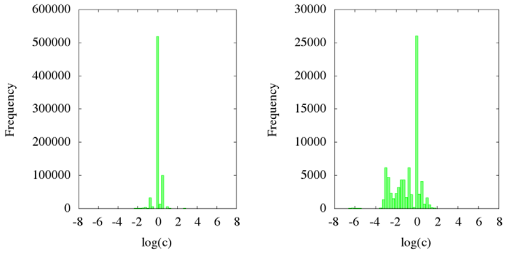


Figure 4: Correlation between user agents and IPs: legitimate accounts (left) and malicious accounts (right).

EvilCohort on the dataset $\mathbf{D}_2$, and consider them as true positives if the accounts belonging to it show patterns that are indicative of automated activity or of botnet use. These techniques are not part of EvilCohort's core detection, but as we will show they are helpful in assessing how well our approach performs.

The results show that most communities of accounts detected by EvilCohort show very different characteristics than legitimate accounts, and are, therefore, very likely malicious. Specifically, of the 83 malicious communities detected by EvilCohort on $\mathbf{D}_2$ only 5 showed characteristics that are similar to regular accounts and are therefore possibly false positives. In particular, we observed a very large community of 5,727 accounts, and four smaller ones of 7, 3, 3, and 2 accounts respectively. Given the size of the largest community, we wanted to understand whether it was really a false positive. Looking at its characteristics, we observed a mix of IP address accessing a single account and a large population of IP addresses accessing many different accounts, which makes us believe that such accounts might have been compromised and being accessed by both their legitimate owners and the hijackers. As such, this community is not a false positive, as it was actually accessed by a botnet. We provide further evidence that this is the case in the following sections. The other communities, on the other hand, are very likely to be false positives by EvilCohort, because they show a consistent human like behavior. Given their small size (15 accounts in total, out of 111,647 total detected accounts), however, we can conclude that the false positives generated by EvilCohort on the dataset $\mathbf{D}_2$ are minimal. In the following, we describe our analysis in detail.

User-agent correlation. Information about user agents was only available for OSN_1 and OSN_2 in $\mathbf{D}_2$. Consequently, we excluded OSN_3 and OSN_4 from this analysis. We also excluded all the account singletons, because the notion of ratio then becomes meaningless.

Based on the description in Section 3.4, we plot the correlation between user agents and IP addresses in Figure 4. The left distribution is generated for legitimate accounts that do not form communities, whereas the right distribution corresponds to accounts in identified malicious communities. The distribution for malicious communities is shifted and no longer aligned on the origin. For legitimate accounts, the average of $log(c)$ was 0.08, which is close to zero, as expected (with a standard deviation of 0.43). For malicious communities, the average shifts to -0.85 with a standard deviation of 1.24.

For the accounts in two of the potential false positive communities described before, the correlation index $log(c)$ was very close to zero, making them very similar to what is expected for regular accounts. For the remaining potential false positive communities this metric did not reveal any anomalous behavior.

Event-based time series. Time series become only significant if the amount of data is sufficiently large to make a measure statistically meaningful. For this reason, we only computed the event-based time series (introduced in Section 3.4) for OSN_1. Unfortunately, the volume of login events observed for OSN_2, OSN_3 and OSN_4 made this approach impractical for these networks.

The assumption behind the time series analysis is that part of the events observed in malicious communities are the result of automation. This results in a distinct shape of activity from communities of legitimate users where events are triggered by humans [17]. To verify this assumption, we plotted the time series associated with the activity of the biggest malicious communities detected in OSN_1. The experiments show that the time series generated for malicious communities differ fundamentally in shape from regular user activity, even when users are grouped behind a NAT.

Concrete examples are plotted in Figure 5 The left time series represents the activity of all users from OSN_1 over 8 days. The reader can clearly see the daily patterns in the activity. The middle time series represents the activity generated by the largest community detected in OSN_1. As can be seen, there are fundamental differences: disappearance of the daily patterns and higher stability on the long term. The right time series is representative of most of the time series obtained for smaller communities of OSN_1: the volume of events remains low but one can clearly observe regular bursts of activity. This bursty shape is also observed for the potential false positive community of 5,727 accounts mentioned previously, which supports our assumption that this community might be composed of compromised accounts that alternate legitimate and malicious activity. The smaller false positive communities, on the other hand, show diurnal patterns similar to the ones observed for legitimate accounts, which support the conclusions that these communities are false positives.

IP addresses and account usage. An alternative representation of account activity over time is to plot the usage graphs for IP addresses and accounts as detailed

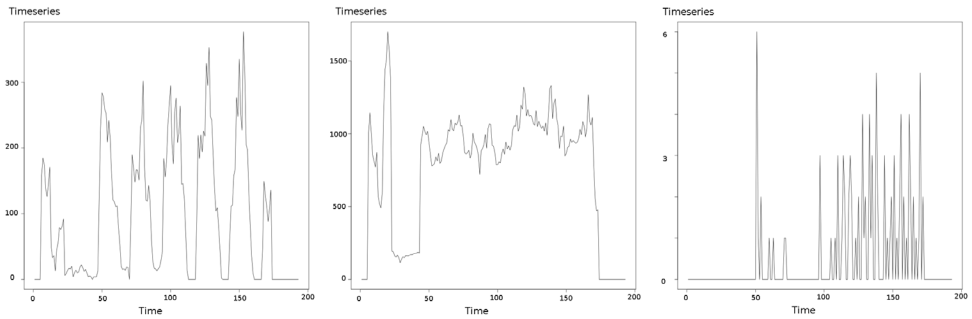


Figure 5: Time series plotting login event over time: legitimate accounts behind a NAT (left plot) and malicious communities (center and right plots).

in Section 3.4. For malicious communities, the usage graphs will exhibit suspicious patterns, indicating synchronization across accounts and IP addresses. For reference, Figure 6 presents the usage graphs for the users behind a NAT. IP address usage is not really relevant for a single IP address, but one can clearly see the daily interruptions over night, as well as the randomness of the events during day time. If we look at malicious communities as plotted in Figure 8, one can see the suspicious vertical patterns appearing. Looking at the IP addresses usage graphs, IP addresses are active in synchronized groups. Previous research already observed that malicious accounts are often used in a synchronized fashion by cybercriminals, and leveraged this property for detection [7]. This gives us additional confidence that the detection performed by EVILCOHORT on the dataset $\mathbf{D}_2$ identifies accounts that are indeed malicious. This anomalous synchronization can be observed for all detected communities of OSN$_1$, OSN$_2$, and OSN$_3$ with the exception of the five potential false positive communities previously mentioned, for which the usage resembles the one of legitimate accounts.

If we look at the large false positive community, however, one can see in Figure 7 that the usage graphs are overall similar to the behavior shown by regular users behind a NAT. However, looking more closely, one can observe multiple darker vertical patterns in the graphs. The mix of legitimate and malicious activities makes us even more confident that such community is indeed composed of compromised accounts accessed by a botnet, and is therefore a true positive detected by EVILCOHORT.

7 Discussion

We showed that EVILCOHORT can be applied to a variety of online services and to any type of activity on these services. This versatility, together with the fact that it complements detections by state-of-the-art systems, makes EVILCOHORT a useful tool in the fight against malicious activity on online services. We hope that, in the future, other researchers will be able to apply the techniques presented in this paper to other online services and types of activity.

As any detection system, EVILCOHORT has some limitations. The main limitation of EVILCOHORT, as we already mentioned, is that it can only detect malicious accounts that are accessed by communities of IP addresses. As we showed in Section 2, however, such accounts are more dangerous than the ones that are accessed by single IP addresses, and existing countermeasures are able to shut down this second type of accounts much quicker.

Another shortcoming is that EVILCOHORT relies on a threshold to limit the number of false positives. An online service that decided to use our approach would have to select a value of s that suits their needs. In this paper we showed that the number of false positives decreases rapidly as we increase s. Applied to our dataset, consisting of millions of events every day, this observation allowed us to reduce false positives to practically zero. Operators can easily tune the value of s by performing sensitivity analysis similar to what we did in Section 5.1.

A last shortcoming is that the accounts used by cybercriminals are not necessarily fake accounts, but could be legitimate accounts that have been compromised. In our current implementation, EVILCOHORT cannot distinguish between the two types of accounts. Dealing with compromised accounts is more difficult, because the online service cannot just suspend them, but has to go through expensive password-reset operations. As we showed in Section 6, it is possible detect whether the detected accounts are fake or compromised by using the postprocessing techniques. A human operator could then decide how to deal with the malicious accounts, depending on their nature.

As with any detection system, a cybercriminal who is aware of EVILCOHORT could attempt to evade it. A straightforward way of doing this would be to have each of the online accounts under his control accessed by a

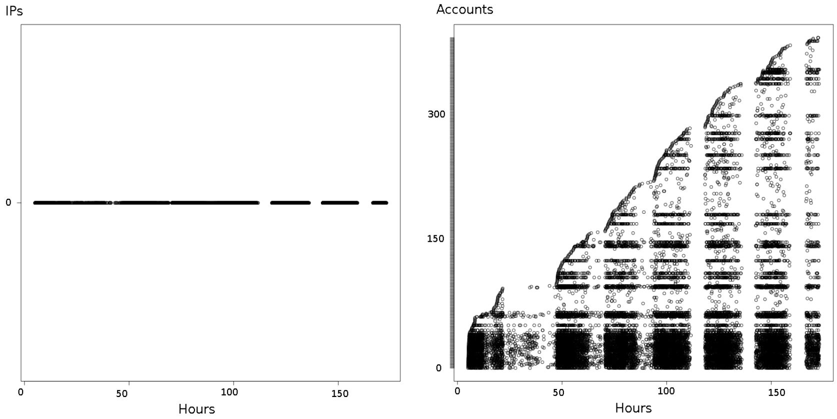


Figure 6: Activity of legitimate users behind a NAT: IP address usage (left) and account usage (right).

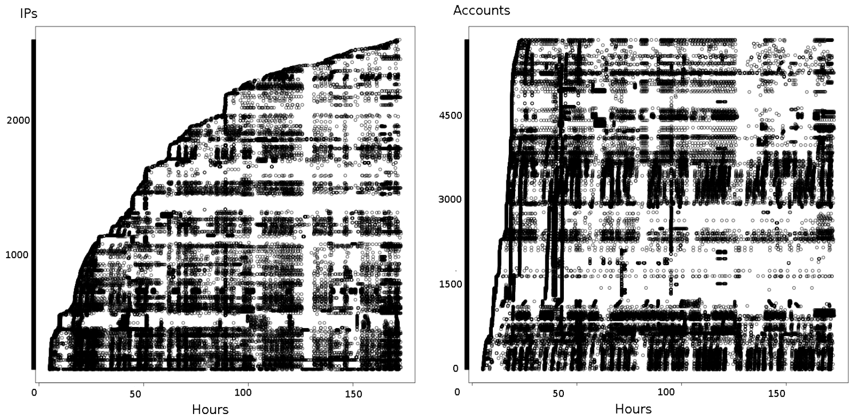


Figure 7: Activity of false positive community: IP address usage (left) and account usage (right). The fact that synchronized activity is interleaved to regular user activity makes us believe that this community is made of compromised accounts.

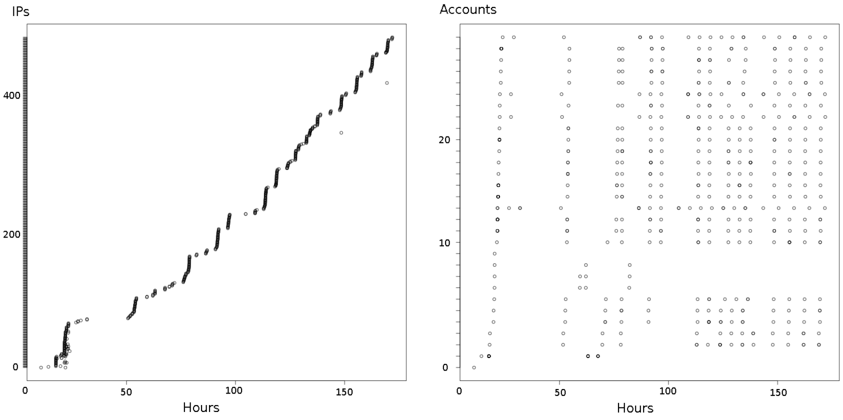


Figure 8: Activity of malicious communities: IP address usage (left) and accounts usage (right).

single IP address. Although this would evade detection by our system, it would make the cybercriminal operation less robust: every time the online service detects and shuts down an account as malicious one of the bots in the botnet becomes useless, while conversely every time a bot is taken offline there is an account on the online service that is not used any more. For this reason EVILCOHORT can be evaded, but by doing so a cybercriminal makes his operation less efficient and profitable.

Since EVILCOHORT can work on any type of activity events, including login events, it has the potential of detecting an account as malicious and blocking it before it performs any malicious activity. In our current implementation, the system works in batches, on time intervals of one day. In the future, we plan to extend it to handle a stream of data, and operate in real time. This way, the system could continuously build communities, and flag an account as malicious as soon as it joins a community.

A source of false alarms for EVILCOHORT could be users who access a service via an IP anonymization service such as Tor. In this case, the set of exit nodes would appear as a community. To mitigate these false positives, we could use the post-processing techniques that we described in Section 3.4. Also, Tor-related communities can be easily identified by comparing the set of IP addresses to the known list of Tor exit nodes. As future work, we plan to explore different tradeoffs between the number of IP addresses accessing a community and the characteristics of such communities.

8 Related Work

A wealth of research has been conducted on detecting malicious activity on online services. Previous work falls into three categories: *content analysis*, *detection of malicious hosts*, and *detection of malicious accounts*.

Content analysis. A corpus of work focuses on detecting malicious content that is shared on online services. Multiple papers dealt with detecting email content that is typical of spam by using machine learning [10, 25]. Other works check whether the URLs posted on an online service are malicious [29,34]. Pitsillidis et al. developed a system that extracts templates from spam [22].

Content analysis systems are effective in detecting and blocking malicious content posted on online services. However, they suffer from two major limitations. The first limitation is that such techniques are typically resource-intensive, and this limits their applicability on busy online services [28]. The second limitation is that such systems can make a detection only when the malicious party tries to posts their content. On the other hand, EVILCOHORT can detect an account (or an IP address) as malicious even if the account does not post any content on the online service.

Detection of malicious hosts (bots). Online services can check in real time if an IP address is known to be a bot by querying DNS blacklists [1]. DNS blacklists are heavily used in anti-spam systems for emails because, unlike content analysis systems, they are lightweight. However, previous research showed that DNS blacklists have a very high number of false negatives [23]. To improve the coverage offered by DNS blacklists, several methods have been proposed. Sinha et al. [24] propose to determine the reputation of an IP address on a global scale, instead of doing it on a local (provider-scale) one. Hao et al. presented SNARE, a system that establishes the reputation of an IP address based on a number of behavioral features (such as the geographical distance between the sender and the recipient) [16].

Detection of malicious accounts. To perform malicious activity on online services, miscreants have to get access to accounts on such services. To this end, they can pay workers to create account for them [33], purchase mass-created fake accounts [30], or buy credentials to compromised accounts on such services [26]. Given the magnitude of the problem, numerous approaches have been proposed to detect accounts that perform malicious activities on online services.

A number of systems analyze the characteristics of accounts on online services, looking for indicators that are typical of mass-created fake accounts (such as the number of people that an account follows) [3, 14, 19, 27, 36, 38]. Yu et al. [37] proposed a system to detect fake social network accounts; the system looks at the network structure, and flags accounts that are not well-connected with their peers in the network as possibly malicious. Benevenuto et al. presented a system to detect accounts that leverage the Youtube service to spread malicious content [4]. Gao et al. [13] developed a system that clusters messages posted on social networks, looking for large-scale spam campaigns. Other approaches look at how messages propagate on social networks, looking for messages that spread anomalously, such as worms [8, 35]. Wang et al. [32] proposed a technique to detect malicious accounts on social networks, based on the sequence of clicks that the people (or the programs) controlling such accounts perform. Jacob et al. [17] presented PUBCRAWL, a system that detects accounts that are used to crawl online services. Egele et al. [11] developed COMPA, a system that learns they typical behavior of social network accounts, and considers any change in behavior as the sign of a possible compromise. Cao et al. presented SynchroTrap [7], a system that detects malicious activity by grouping together social network accounts that performed similar actions during the same period of time. EVILCOHORT improves over SynchroTrap, because accounts do not have to act synchronously to be detected.

Although systems that detect malicious accounts are useful to detect and block malicious activity on online services, they typically rely on a single threat model, and can only detect malicious accounts that operate following that threat model. Conversely, EVILCOHORT leverages the observation that cybercriminals use a set of (compromised) IP addresses and a set of online accounts. For this reason, EVILCOHORT can detect online accounts controlled by cybercriminals, regardless of the purpose for which these accounts are used.

Studying communities of interest. A large corpus of research has been conducted over the years to study communities of interest in networks [2, 20]. Such communities are collections of hosts that share the same goal. Studying communities of interest is useful to model the typical behavior of related hosts, and detect anomalies in their behavior that can be indicative of malicious activity. The communities that we study in this paper are different in nature, because they are composed of online accounts instead of hosts and are consistently used for malicious purposes by miscreants.

The closest work to EVILCOHORT is BOTGRAPH [39]. Similar to EVILCOHORT, this system looks at accounts that are accessed by a common set of IP addresses. However, BOTGRAPH relies on heuristics that are dependent on the email-sending habits of accounts to perform detection, and therefore limit its applicability to the spam-fighting domain. Conversely, EVILCOHORT is principled, and can be applied on any online service without pre-existing domain knowledge. In addition, the fact that EVILCOHORT can be applied on activity other than email-sending events (for example login events) allows us to detect malicious activity other than sending malicious content (e.g., online accounts used as a C&C channel). Another difference is that BOTGRAPH calculates its threshold over a 30-day period, and therefore is not suited to perform detection on freshly-created accounts. In this paper, on the other hand, we showed that EVILCOHORT can work in one-day batches, and detect as malicious accounts that were created during that same day.

9 Conclusions

We presented EVILCOHORT, a system that detects malicious accounts on online services by identifying communities of accounts that are accessed by a common set of computers. Our results show that the vast majority of the accounts that form such communities are used for malicious purposes. In the rare cases in which legitimate communities of accounts form, we show that such communities present characteristics that are very different than the ones of malicious communities. These differences can be used to perform more accurate detection. We ran EVILCOHORT on two real-world datasets, and detected more than one million malicious accounts.

Acknowledgments

This work was supported by the Office of Naval Research (ONR) under grant N00014-12-1-0165, the Army Research Office (ARO) under grant W911NF-09-1-0553, the Department of Homeland Security (DHS) under grant 2009-ST-061-CI0001, the National Science Foundation (NSF) under grant CNS-1408632, and SBA Research.

References

[1] Spamhaus dbl. http://www.spamhaus.org.

[2] AIELLO, W., KALMANEK, C., MCDANIEL, P., SEN, S., SPATSCHECK, O., AND VAN DER MERWE, J. Analysis of communities of interest in data networks. In *Passive and Active Network Measurement*. 2005.

[3] BENEVENUTO, F., MAGNO, G., RODRIGUES, T., AND ALMEIDA, V. Detecting Spammers on Twitter. In *Conference on Email and Anti-Spam (CEAS)* (2010).

[4] BENEVENUTO, F., RODRIGUES, T., ALMEIDA, V., ALMEIDA, J., AND GONÇALVES, M. Detecting spammers and content promoters in online video social networks. In *ACM SIGIR conference on Research and development in information retrieval* (2009).

[5] BILGE, L., STRUFE, T., BALZAROTTI, D., AND KIRDA, E. All Your Contacts Are Belong to Us: Automated Identity Theft Attacks on Social Networks. In *World Wide Web Conference (WWW)* (2009).

[6] BLONDEL, V. D., GUILLAUME, J.-L., LAMBIOTTE, R., AND LEFEBVRE, E. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment* (2008).

[7] CAO, Q., YANG, X., YU, J., AND PALOW, C. Uncovering Large Groups of Active Malicious Accounts in Online Social Networks. In *ACM Conference on Computer and Communications Security (CCS)* (2014).

[8] CAO, Y., YEGNESWARAN, V., PORRAS, P., AND CHEN, Y. PathCutter: Severing the Self-Propagation Path of XSS Javascript Worms in Social Web Networks. In *Symposium on Network and Distributed System Security (NDSS)* (2012).

[9] COOKE, E., JAHANIAN, F., AND MCPHERSON, D. The zombie roundup: Understanding, detecting, and disrupting botnets. In *Proceedings of the USENIX SRUTI Workshop* (2005).

[10] DRUCKER, H., WU, D., AND VAPNIK, V. N. Support vector machines for spam categorization. In *IEEE transactions on neural networks* (1999).

[11] EGELE, M., STRINGHINI, G., KRUEGEL, C., AND VIGNA, G. Compa: Detecting compromised accounts on social networks. In *Symposium on Network and Distributed System Security (NDSS)* (2013).

[12] FARKAS, I. J., DERÉNYI, I., BARABÁSI, A.-L., AND VICSEK, T. Spectra of real-world graphs: Beyond the semicircle law. *Physical Review E* (2001).

[13] GAO, H., CHEN, Y., LEE, K., PALSETIA, D., AND CHOUDHARY, A. Towards Online Spam Filtering in Social Networks. In *Symposium on Network and Distributed System Security (NDSS)* (2012).

[14] GAO, H., HU, J., WILSON, C., LI, Z., CHEN, Y., AND ZHAO, B. Detecting and Characterizing Social Spam Campaigns. In *Internet Measurement Conference (IMC)* (2010).

[15] GRIER, C., THOMAS, K., PAXSON, V., AND ZHANG, M. @spam: the underground on 140 characters or less. In *ACM Conference on Computer and Communications Security (CCS)* (2010).

[16] HAO, S., SYED, N. A., FEAMSTER, N., GRAY, A. G., AND KRASSER, S. Detecting Spammers with SNARE: Spatio-temporal Network-level Automatic Reputation Engine. In *USENIX Security Symposium* (2009).

[17] JACOB, G., KIRDA, E., KRUEGEL, C., AND VIGNA, G. Pubcrawl: protecting users and businesses from crawlers. In *USENIX Security Symposium* (2012).

[18] JAGATIC, T., JOHNSON, N., JAKOBSSON, M., AND JAGATIF, T. Social Phishing. *Comm. ACM 50*, 10 (2007), 94–100.

[19] LEE, K., CAVERLEE, J., AND WEBB, S. Uncovering social spammers: social honeypots + machine learning. In *International ACM SIGIR Conference on Research and Development in Information Retrieval* (2010).

[20] MCDANIEL, P. D., SEN, S., SPATSCHECK, O., VAN DER MERWE, J. E., AIELLO, W., AND KALMANEK, C. R. Enterprise Security: A Community of Interest Based Approach. In *Symposium on Network and Distributed System Security (NDSS)* (2006).

[21] NIU, Y., WANG, Y., CHEN, H., MA, M., AND HSU, F. A Quantitative Study of Forum Spamming Using Context-based Analysys. In *Symposium on Network and Distributed System Security (NDSS)* (2007).

[22] PITSILLIDIS, A., LEVCHENKO, K., KREIBICH, C., KANICH, C., VOELKER, G. M., PAXSON, V., WEAVER, N., AND SAVAGE, S. botnet Judo: Fighting Spam with Itself. In *Symposium on Network and Distributed System Security (NDSS)* (2010).

[23] RAMACHANDRAN, A., DAGON, D., AND FEAMSTER, N. Can DNS-based blacklists keep up with bots? In *Conference on Email and Anti-Spam (CEAS)* (2006).

[24] S. SINHA, M. B., AND JAHANIAN, F. Improving Spam Blacklisting Through Dynamic Thresholding and Speculative Aggregation. In *Symposium on Network and Distributed System Security (NDSS)* (2010).

[25] SAHAMI, M., DUMAIS, S., HECKERMANN, D., AND HORVITZ, E. A Bayesian approach to filtering junk e-mail. *Learning for Text Categorization* (1998).

[26] STONE-GROSS, B., COVA, M., CAVALLARO, L., GILBERT, B., SZYDLOWSKI, M., KEMMERER, R., KRUEGEL, C., AND VIGNA, G. Your Botnet is My Botnet: Analysis of a Botnet Takeover. In *ACM Conference on Computer and Communications Security (CCS)* (2009).

[27] STRINGHINI, G., KRUEGEL, C., AND VIGNA, G. Detecting Spammers on Social Networks. In *Annual Computer Security Applications Conference (ACSAC)* (2010).

[28] TAYLOR, B. Sender reputation in a large webmail service. In *Conference on Email and Anti-Spam (CEAS)* (2006).

[29] THOMAS, K., GRIER, C., MA, J., PAXSON, V., AND SONG, D. Design and Evaluation of a Real-Time URL Spam Filtering Service. In *IEEE Symposium on Security and Privacy* (2011).

[30] THOMAS, K. AND MCCOY, D. AND GRIER, C. AND KOLCZ, A. AND PAXSON, V. Trafficking Fraudulent Accounts: The Role of the Underground Market in Twitter Spam and Abuse. In *USENIX Security Symposium* (2013).

[31] THOMASON, A. Blog Spam: A Review. In *Conference on Email and Anti-Spam (CEAS)* (2007).

[32] WANG, G., KONOLIGE, T., WILSON, C., WANG, X., ZHENG, H., AND ZHAO, B. Y. You are how you click: Clickstream analysis for sybil detection. *USENIX Security Symposium* (2013).

[33] WANG, G., WILSON, C., ZHAO, X., ZHU, Y., MOHANLAL, M., ZHENG, H., AND ZHAO, B. Y. Serf and turf: crowdturfing for fun and profit. In *World Wide Web Conference (WWW)* (2012).

[34] XIE, Y., YU, F., ACHAN, K., PANIGRAHY, R., HULTEN, G., AND OSIPKOV, I. Spamming Botnets: Signatures and Characteristics. *SIGCOMM Comput. Commun. Rev.* (2008).

[35] XU, W., ZHANG, F., AND ZHU, S. Toward worm detection in online social networks. In *Annual Computer Security Applications Conference (ACSAC)* (2010).

[36] YANG, C., HARKREADER, R., AND GU, G. Die Free or Live Hard? Empirical Evaluation and New Design for Fighting Evolving Twitter Spammers. In *Symposium on Recent Advances in Intrusion Detection (RAID)* (2011).

[37] YU, H., KAMINSKY, M., GIBBONS, P. B., AND FLAXMAN, A. Sybilguard: defending against sybil attacks via social networks. *ACM SIGCOMM Computer Communication Review* (2006).

[38] ZHANG, C. M., AND PAXSON, V. Detecting and Analyzing Automated Activity on Twitter. In *Passive and Active Measurement Conference* (2011).

[39] ZHAO, Y., XIE, Y., YU, F., KE, Q., YU, Y., CHEN, Y., AND GILLUM, E. Botgraph: Large scale spamming botnet detection. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2009).

[40] ZHONG, Z., HUANG, K., AND LI, K. Throttling Outgoing SPAM for Webmail Services. In *Conference on Email and Anti-Spam (CEAS)* (2005).

Trends and Lessons from Three Years Fighting Malicious Extensions

Nav Jagpal Eric Dingle Jean-Philippe Gravel Panayiotis Mavrommatis
Niels Provos Moheeb Abu Rajab Kurt Thomas
Google
{*nav, ericdingle, jpgravel, panayiotis, niels, moheeb, kurtthomas*}*@google.com*

Abstract

In this work we expose wide-spread efforts by criminals to abuse the Chrome Web Store as a platform for distributing malicious extensions. A central component of our study is the design and implementation of WebEval, the first system that broadly identifies malicious extensions with a concrete, measurable detection rate of 96.5%. Over the last three years we detected 9,523 malicious extensions: nearly 10% of every extension submitted to the store. Despite a short window of operation—we removed 50% of malware within 25 minutes of creation— a handful of under 100 extensions escaped immediate detection and infected over 50 million Chrome users. Our results highlight that the extension abuse ecosystem is drastically different from malicious binaries: miscreants profit from *web traffic* and *user tracking* rather than email spam or banking theft.

1 Introduction

Browsers have evolved over recent years to mediate a wealth of user interactions with sensitive data. Part of this rich engagement includes extensions: add-ons that allow clients to customize their browsing experience by altering the core functionality of Chrome, Firefox, and Internet Explorer. Canonical examples include search toolbars, password managers, and ad blockers that once installed intercept webpage content through well-defined APIs to modify every page a user visits.

Criminals have responded in kind by developing malicious extensions that interpose on a victim's browsing sessions to steal information, forge authenticated requests, or otherwise tamper with page content for financial gain. Poignant malware strains include Facebook account hijackers, ad injectors, and password stealers that exist purely as *man-in-the-browser* attacks [21, 25, 37, 41]. While many of these threats have binary-based equivalents—for instance the Torpig banking trojan that injected rogue phishing forms into banking webpages or the ZeroAccess bot that tampered with page advertisements [27, 34]—extensions bridge the semantic gap between binaries and browsers, trivializing broad access to complex web interactions.

In this paper we expose wide-spread efforts by criminals to abuse the Chrome Web Store as a platform for distributing malicious extensions. Our evaluation covers roughly 100,000 unique extensions submitted to the Chrome Web Store over a three year span from January 2012–2015. Of these, we deem nearly *one in ten* to be malicious. This threat is part of a larger movement among malware authors to pollute official marketplaces provided by Chrome, Firefox, iOS, and Android with malware [7, 10, 42].

A central component of our study is the design and implementation of WebEval, the first system that broadly identifies malicious extensions with a concrete, measurable detection rate of 96.5%. We arrive at a verdict by classifying an extension's behaviors, code base, and developer reputation. In the process, we incorporate existing techniques that detect specific malware strains and suspicious extension behaviors and evaluate each of their effectiveness in comparison to our own [21, 37, 41]. WebEval also faces a unique challenge: live deployment protecting the Chrome Web Store where attackers have a strong incentive to adapt to our infrastructure. We explore the impact that evasive threats have on our overall accuracy throughout our deployment and the necessity of human experts to correct for model drift.

In total, we removed 9,523 malicious extensions from the Chrome Web Store. The most prominent threats included social network session hijackers that generated synthetic likes, friend requests, and fans; ad injectors that rewrote DOM content to laden pages with additional advertisements; and information stealers that injected rogue tracking pixels and covertly siphoned search keywords. Despite a short window of operation—we reported 50% of malware within 25 minutes of creation—a handful of under 100 malicious extensions distributed via binary payloads were able to infect nearly 50 million users before removal. We distill these observations into a number of key challenges facing app marketplaces that extend beyond just the Chrome Web Store.

In summary, we frame our contributions as follows:

- We present a comprehensive view of how malicious extensions in the Chrome Web Store have evolved and monetized victims over the last three years.
- We detail the design and implementation of our security framework that combines dynamic analysis, static analysis, and reputation tracking to detect 96.5% of all known malicious extensions.
- We highlight the importance of human experts in operating any large-scale, live deployment of a security scanner to address evasive malware strains.
- We explore the virulent impact of malicious extensions that garner over 50 million installs; the single largest threat infecting 10.7 million Chrome users.

2 Background

We provide a brief background on how users obtain extensions and the control extensions have over the Chrome browser that simplify malware development.

2.1 Chrome Web Store

The Chrome Web Store is the central repository of all Chrome extensions. While initially the store was an optional ecosystem, rampant abuse outside of the store lead to Chrome locking down all Windows machines in May 2014 [13]. With this policy decision, Chrome now automatically blocks all extensions not present in the store from installation.

The Chrome Web Store relies on built in protections against malware that subject every extension to an abuse review. This approach is not unique to Chrome: Firefox, iOS, and Android all rely on application reviews to protect their user base from malware [1, 14, 17]. Malicious extensions detected during preliminary review are never exposed to the public. In the event the Chrome Web Store retroactively detects a malicious extension, the store can take down the offending code and signal for all Chrome clients to expunge the extension [16]. This defense layer provides a homogeneous enforcement policy for all Chrome users compared to the heterogeneous security environments of their desktop systems that may have no recourse against malicious extensions.[1]

2.2 Chrome Extension Architecture

Developers author extensions much like websites using a combination of JavaScript, CSS, and HTML. Unlike websites, extensions are exempt from same origin protections and are afforded a range of Chrome and document-level controls that allow customizing how users interact with the Internet.

Permissions: Extensions may interact with privileged Chrome resources such as tabs, cookies, and network traffic through a Chrome-specific API. Chrome mediates these sensitive capabilities through a coarsely defined permission model where a permission consists of a resource (e.g., cookies) and a scope (e.g., *https://mail.google.com*) [4]. When a developer authors an extension, she lists all desired permissions in a static manifest. As discussed by Carlini *et al.*, this design favors "benign but buggy" extensions where the author adheres to a principle of least privilege [9]. The model provides no protection against malicious extensions beyond explicitly signaling broad capabilities (e.g., intercepting all network traffic).

Background Page & Content Scripts: Chrome loads an extension's core logic into a long running process called a background page. This privileged process obtains access to all of the Chrome API resources specified in the extension's permission manifest. To prevent permission re-delegation attacks, Chrome isolates background pages from all other extensions and web sites. Chrome eases this isolation by allowing extensions to register content scripts that run directly in the context of a web page as though part of the same origin (and thus with access to all of the origin's DOM content, DOM methods, and session cookies). Background pages communicate with content scripts through a thin message passing layer provided by Chrome. As with the Chrome API, extensions must specify content scripts and the targeted domains in an extension's manifest.

3 System Overview

We develop our system called WebEval to protect the Chrome Web Store from malicious extensions. The challenge is messy and fraught with evasive malware strains that adapt to our detection techniques. We rely on a blend of automated systems and human experts who work in conjunction to identify threats and correct for failures surfaced by user reports of abuse. Before diving into detailed system operations, we highlight the design principles that guided our development and offer a birds eye view of the entire architecture's operation.

[1] Chrome extensions are not supported on Android or other mobile platforms. As such, we limit our discussion of malicious extensions to desktop environments.

3.1 Design Goals

At its heart, WebEval is designed to return a verdict for whether an extension is malicious. If the extension is pending publication, the Chrome Web Store should *block* the extension from release. Previously published extensions must be *taken down* and uninstalled from all affected Chrome instances. Arriving at a malware verdict is constrained by multiple requirements:

1. *Minimize malware installs.* Our foremost goal with WebEval is to minimize the number of users exposed to malicious extensions. However, near-zero false positives are imperative as Chrome expunges an extension's entire user base if we return a malware verdict incorrectly. We design our system such that human experts vet every verdict prior to actioning.

2. *Simplify human verification.* Whenever possible, our system should be fully automated to minimize the time required from human experts to confirm an extension is malicious.

3. *Time-constrained.* Our system embargoes extensions from public release until we reach a verdict. Its critical that we return a decision within one hour. Relatedly, our system must scale to the throughput of newly submitted items to the Chrome Web Store and weekly re-evaluated extensions that we estimate at roughly 19,000 reviews/day.

4. *Comprehensible, historical reports.* Any automated reports produced by our system must be comprehensible to human analysts, including machine learning verdicts. Similarly, all reports should contain some annotation to allow a historical perspective on the evolution of malicious extensions.

5. *Tolerant to feature drift.* Finally, our system must keep pace with the evasive nature of malware and adaptations in monetization strategies. This includes allowing experts to easily deploy new rules to capture emerging threats that are then automatically incorporated into long-running detection modules.

3.2 System Flow

WebEval is a living system that has evolved over the last three years in response to threats facing the Chrome Web Store. We describe our current pipeline for classifying an extension as malicious in Figure 1. The system consists of four stages: (❶) a scheduler that submits extensions for evaluation; (❷) our extension execution framework that captures behavioral signals; (❸) an annotation phase that incorporates content similarity, domain reputation, and anti-virus signatures; and finally (❹) scoring where manually curated rules, an automated classifier, and human experts reach a verdict for whether an extension is malicious.

Scheduler: We feed every extension uploaded to the Chrome Web Store, either new or updated, into our system and analyze it within one hour of submission. In total, we analyzed 99,818 Chrome extensions submitted over the course of January 2012–January 2015. This set includes extensions that were blocked prior to public release.[2] Furthermore, we have access to each revision of the extension's code base: over 472,978 unique variants (measured by SHA1 sums). Each revision triggers a rescan in addition to a weekly re-scan aimed at extensions that fetch dynamic, remote resources that can become malicious.

Evaluation: We subject every extension to an evaluation phase that extracts behavioral signals for classification. This includes a reputation scan of the publisher, static analysis of the extension's code base, and dynamic analysis that emulates common tasks performed in Chrome: querying search engines, visiting social media, and browsing popular news sites. We store all raw features for posterity, totaling over 45 TB. Our philosophy is to retain everything (even packet contents) in order to enable offline analysis in the event an extension becomes defunct due to dead or broken remotely fetched resources. This storage simultaneously enables tracking trends in malware behavior over time and retroactively applying new malware signatures. We present the full details of our evaluation framework in Section 4.

Annotation: We practice a defense in depth strategy that incorporates domain blacklists, anti-virus engines, and content similarity that contextualizes an extension's behaviors against the larger ecosystem of malicious developers and extensions. We include these signals as annotations to an extension's evaluation in the event our own behavioral suites fail to surface any malicious logic. We present the annotation process in greater detail at the end of Section 4.

Scoring: The final step of WebEval returns a verdict for whether to expunge a malicious extension. We use a combination of manually curated rules and a logistic regression classifier re-trained daily over all previously detected malicious extensions to generate a score. A human expert then confirms our automated verdict before passing our decision on to the Chrome Web Store to take action. We present our technique for training, regulariza-

[2] We note that any extensions blocked prior to release are absent from the previous work by Kapravelos *et al.* [21] that studied malicious extensions found in the Chrome Web Store.

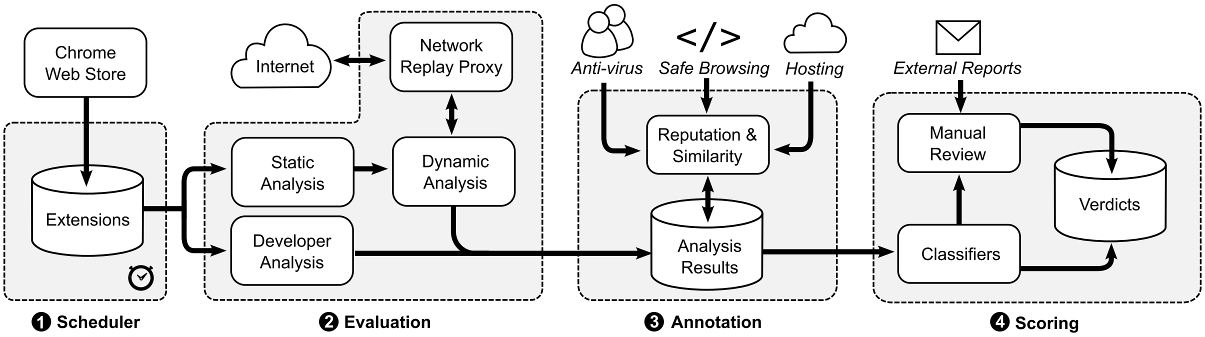


Figure 1: Our pipeline for detecting malicious extensions. WebEval's architecture consists of a scheduler, extension execution framework, an annotator that incorporates third-party security intelligence, and finally scoring where we return a malware verdict.

tion, and manual decision rules in Section 5. We discuss our approach for obtaining labeled data and evaluating our system's accuracy later in Section 6.

4 Evaluating Extensions

WebEval's core automation systems generate a report that surfaces signals about an extension's code base, the impact it has a user's browsing experience, and who developed the extension. With a few exceptions, none of these signals in isolation indicate outright malice: we leave it up to our classifier and human experts to determine which combinations of features clearly distinguish malware.

4.1 Static Analysis

Apart from remotely fetched resources, all of an extension's HTML, CSS, JavaScript, and manifest are self-contained and available to the Chrome Web Store upon submission. We scan through these components to identify potential threats.

Permissions & Content Scripts: We enumerate all an extension's permissions, content scripts, and contexts. Permissions in particular offer some indication of an extension's capabilities such as intercepting and modifying traffic (*proxy, webRequest*), triggering on a page load (*tabs*), introspecting on all cookies (*cookies*), and uninstalling or disabling other extensions (*management*). As part of this process we also identify broad contexts (e.g., <all_urls>, *https://**) that allow an extension to interact with every page.

Code Obfuscation: We scan for the presence of three types of code obfuscation: minification, encoding, and packing. We build on the detection strategy of Kaplan *et al.* that identifies common character substrings found in obfuscated vs. unobfuscated code [20]. Instead of detecting individual characters, we develop a set of regular expressions that identifies boilerplate initialization tied to the most prominent packers (e.g., *jsmini.com*, *jscompress.com*, and */packer/*). We employ a similar approach for detecting long encoded character strings. Finally, we detect minification by measuring the distance between a prettified version of an extension's JavaScript against the original supplied by the developer.

Files and Directory Structure: We extract the file names and directory structure of an extension as well as text shingles of the contents of every file. We rely on these features for detecting near-duplicate extensions (discussed in Section 4.4) as well as identifying commonly imported libraries and malicious files.

4.2 Dynamic Analysis

We collect the majority of our malware signals by black-box testing each extension with a barrage of behavioral suites that simulate common browsing experiences as well as custom tailored detection modules that trigger malicious logic. While more exhaustive approaches such as symbolic JavaScript execution exist [32], in practice we obtain sufficient enough behavioral coverage to reach accurate malware verdicts as discussed in Section 6.

Sandbox Environment: Our testing environment consists of a Windows virtual machine outfitted with two logging components: (1) a *system monitor* that captures low-level environment changes such as Windows settings, Chrome settings, and file creation; and (2) an *in-browser activity logger* that interposes on and logs all DOM events and Chrome API calls. This activity logger is natively built into Chrome explicitly for monitoring the behavior of extensions [28]. We note that Chrome isolates extensions from this logging infrastructure. Extensions cannot tamper with our results unless they compromise Chrome itself.

We supplement our monitoring infrastructure by routing all network traffic through a *network logging proxy*. This proxy also serves as a replay cache. For each test suite we develop, we first record the network events produced by Chrome absent any extension installed. During dynamic evaluation we replay any network events from the cache that match. If a cache miss occurs, we route requests out of our sandbox to the Internet. This proxy allows us to minimize page dynamism, guarantee that test suites are consistent across executions absent new dynamic behavior, and reduce overall network round trip time. Similarly, we can easily flag network requests produced by our test actions (e.g., a click or fetching ad content) versus those produced by an extension.

The output of our dynamic analysis is a list of all network requests, DOM operations, and Chrome API calls made by an extension. Each of these include the page the event occurred on as well as the remote target of XHR requests and injected scripts. We supply all of these as raw features to our classifier as well as to human experts who can generate manual rules that capture sequences of events tied to known malware strains.

Behavioral Suites: The event driven nature of extensions requires that we replay realistic browsing scenarios to trigger malware. Our system allows human experts to record complex interactions (e.g., clicks, text entry, etc) with webpages that we then replay against every extension to detect malicious behaviors. These simulations, called behavioral suites, cover querying *google.com* with multiple searches; logging into *facebook.com* via a test account and viewing the account's news feed; shopping on *amazon.com* and *walmart.com*; and lastly browsing popular media sites including *nytimes.com* and *youtube.com*. As new threats arise, analysts can easily deploy new behavioral suites to trigger a malicious extension's logic.

Generic Suites: Our replay suites are by no means exhaustive; we rely on a generic set of test suites to simulate a wider variety of browser events. These tests are duplicates of the techniques previously discussed by Kapravelos *et al.* for Hulk [21] and include simulating network requests to popular news, video, shopping, and banking sites to trigger an extension's *webRequest* handler as well as using HoneyPages that create dummy elements on the fly to satisfy JavaScript requests from extensions.

Malicious Logic Suites: We supplement our browsing actions by explicitly testing an extension's logic against known threats: uninstalling other extensions (e.g., anti-virus, Facebook malware remover); preventing uninstallation by terminating or redirecting tabs opening *chrome://extensions*; and stripping or modifying Content Security Policy headers. We explicitly flag each of these activities in addition to the log signals produced throughout the extension's evaluation.

4.3 Developer Analysis

The closed-garden nature of the Chrome Web Store enables tracking fine-grained reputation about developers and the extensions they author. We monitor where developers log in from, the email domain they use to register, the age of the developer account, and the total number of extensions authored thus far. These signals help us detect fake developer accounts that miscreants register via commonly abused email providers and proxies, staples of abusive account creation [39]. We note that newly registered developers must pay a nominal one-time fee of $5 that increases the overhead of churning out fake accounts [15].

In the event a malicious extension escapes initial detection, we also incorporate signals generated from users interacting with the Chrome Web Store. These includes the number of installs an extension receives, the number of users who have rated the extension, and the average rating. Our intuition is that highly used extensions that never receive any feedback are suspicious as are extensions that receive many low ratings.

4.4 Annotation

In the event the signals we collect during evaluation are insufficient, we rely on a defense in depth strategy that incorporates intelligence from the broader security community. In particular, we scan all of the files included in an extension with multiple anti-virus engines similar to VirusTotal.[3] If any single anti-virus vendor reports a file as malicious we flag the file in our report. We extend a similar strategy to all of the outgoing network requests produced by an extension where we scan the domains contacted against Google Safe Browsing and a collection of domain blacklists.

We also evaluate an extension in the context of all previously scanned extensions. We take the text shingles of an extension's code base computed during static analysis and identify near-duplicate extensions that share 80% of the same code. This approach allows us to detect extension developers that routinely re-upload previously detected malicious extensions. We extend this clustering logic to group extensions based on common embedded strings such as Google Analytics UIDs, Facebook App IDs, and Amazon Affiliate IDs. Finally, for extensions that evade initial detection and are released to the Chrome Web Store, we cluster the extensions based on the referrer of all incoming install requests to identify common websites involved in social engineering. We surface these clusters to human experts along with the ratio of known malware in each cluster.

[3] Due to licensing agreements, we are unable to disclose which anti-virus software we scan with.

5 Scoring Extensions

We reach a concrete verdict of an extension's malice by flagging any extension caught by our automated classifier or manually constructed heuristics. A human expert then verifies our decision and removes the offending extension from the Chrome Web Store if appropriate.

5.1 Automated Detection

Our automated detection uses a proprietary implementation of an online gradient descent logistic regression with L1 regularization to reduce the size of our feature space [6]. We believe similar accuracy can be achieved in a distributed fashion with the open source machine learning libraries provided by Spark [33]. We train a model daily over all previously scanned extensions with labeled training data originating from human experts (discussed shortly in Section 6).

Our feature set for classification consist of a collection of over 20 million signals. For each extension we construct a sparse string feature vector that contains every requested permission, the contexts the extension operates on, whether obfuscation was present, and a string representation of all of the extension's file names and directory structure. From dynamic analysis we include a feature for every DOM operation, Chrome API call, XHR request, remotely contacted domain, and a bit for whether the extension uninstalled a security related extension, prevented uninstallation, or modified CSP headers. From the developer analysis we include the email domain, last login geolocation, and a discretized bucket of the developer account's age.

We exclude annotation signals from learning; they are only used by human experts for manually curating rules and analyzing clusters of badness. We also exclude text shingles both to limit our feature space and retain meaningful signals. Our philosophy is that any input to the classifier should have a direct translation to an activity that analysts can recognize rather than loosely contextualized text blobs.

As part of the learning stage, we assign each feature a weight which we optimize using a gradient descent on a logistic regression model. In particular, we use L1 regularization to reduce our feature set to roughly 1,000 of the most impactful features. These features become decision rules, which we use to classify new extensions. Because human reviewers cannot look at every single extensions in the Web Store, we have variable confidence in the malware or benign labels assigned to training instances. To compensate for this, we multiply the gradient descent learning rate with a correction factor that is proportional to an approximate confidence level. Every known malware items gets a correction factor of 1.0 due to prior vetting by a human expert. On the other hand, the learning rate for benign items is scaled down by the following factor:

$$f = \frac{min(\frac{P}{P_t}, 1.0) + min(\frac{A}{A_t}, 1.0)}{2}$$

We represent the popularity of an extension P as the number of existing installs and the age of an extension A as the number of days since the extension was published. P_t and A_t represent thresholds above which we omit any penalty. This correction factor captures the risk that a new extensions with no user base is malicious and yet to be identified, while seasoned extensions with tens of thousands of users are likely benign.

For the sake of tuning the learning pipeline, we use 5-folds cross validation to confirm we do not overfit the model. The final model we use in production is trained on 100% of the data available. For the purposes of our study, we evaluate our model based on its accuracy the next day rather than relying on a holdout golden dataset.

5.2 Manual Rules

We supplement our automated detection with manually curated rules generated by human experts that address many of the most prominent threats facing the Chrome Web Store (discussed later in Section 7). While these rules are fall backs in the event our automated classifier fails, they are immensely helpful in contextualizing the monetization strategy of malicious extensions that we track over time. We note that all extensions surfaced by these rules are still subject to expert verification.

Facebook Hijacking: Initial reports of malicious extensions hijacking a victim's Facebook account to post status updates, send chat messages, befriend users, or "like" content without consent first emerged in 2012 and have persisted ever since [29]. We detect these extensions by scanning network logs produced during dynamic evaluation for outgoing network POSTs to resources (e.g., *ajax/follow/follow_profile.php*) that may indicate unauthorized account behavior.

Ad Injection: Ad injection extensions insert or replace web advertisements. We identify this behavior by comparing the origin of inserted DOM elements and injected scripts against a list of known advertisers derived from third party ad block software, previous reports on ad injection affiliate programs [37, 41], and domains surfaced during manual review. We also scan for DOM operations that replace existing advertisements on any of the pages visited during our behavioral suites where we know ad positions *a priori*.

Search Leakage: Search leakage broadly refers to any extension that funnels search queries to third parties, typically for modifying search results, injecting advertisements, or tracking user interests. We detect search leakage by scanning outgoing network requests to determine whether they contain the same keywords our behavioral suite supplies to *google.com*. This module may potentially miss term leakage in the event of encrypted or obfuscated network parameters.

User Tracking: We rely on a heuristic to detect user tracking that involves scanning all DOM operations for the insertion of 0×0, 1×1, or hidden image during dynamic analysis. We consider any such operation a likely indicator of inserting a tracking pixel.

6 Evaluation

We evaluate WebEval under a live deployment and track daily accuracy as vetted by human experts. As part of our analysis we offer insights into the most important features for classification and the role of human experts in correcting for evasive strains.

6.1 Dataset

Our evaluation dataset consists of 99,818 extensions scored by WebEval between January 2012–2015. Human experts provided our ground truth labels. Due to the possibility of delayed detection we continue to update labels one month after the cut off for our dataset. In total, experts identified 9,523 malicious extensions (9.4% of all extensions created during the same window). For the purposes of our evaluation, we define WebEval's verdict as a *false positive* if WebEval returned a malware label that was either rejected as incorrect by human experts or later refuted by the extension's developer and overturned upon secondary review. Similarly, we define a *false negative* as any extensions surfaced by human experts or external reports despite our system returning a benign verdict. We likely underestimate false negatives as some threats are bound to escape both automated and external review.

6.2 Overall Accuracy

We measure the precision and recall of WebEval as a function of all scored extensions over the last three years. In total, our machine learning pipeline and manually curated rule sets surfaced 93.3% of all known malicious extensions to human experts (recall). Of the extension's that WebEval flagged as potentially malicious, human experts agreed 73.7% of the time (precision). If we restrict our calculation to the last year, WebEval had a recall of 96.5% and a precision of 81%. We find that accuracy is a living process that we detail in Figure 2.

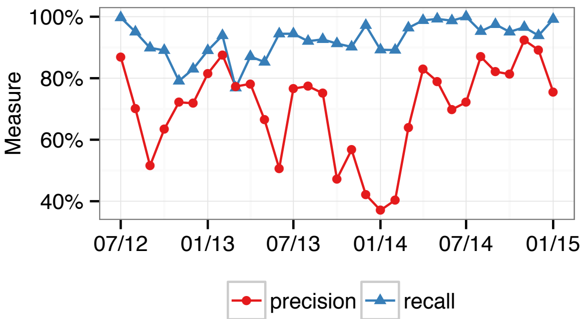


Figure 2: Monthly precision and recall of all scoring systems in aggregate from 2012–2015.

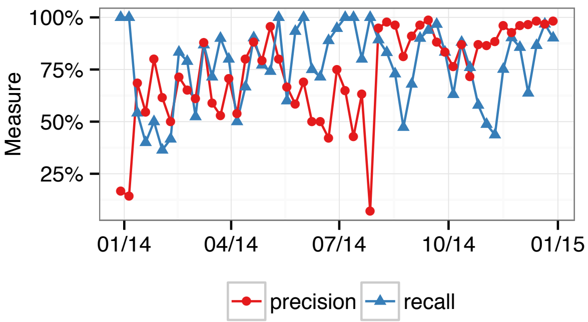


Figure 3: Weekly precision and recall of logistic regression classifier from 2014–2015.

We experience drops in recall when new threats emerge which we subsequently recover from via updated rules and daily retraining of our classifier with new samples. We consistently value recall over precision: we would rather burden human experts with more reviews rather than expose users to malicious extensions. Nevertheless, our precision is reasonable enough as to not force human experts to review every extension in our dataset.

6.3 Automated Classifier Accuracy

Ideally WebEval can run in a purely automated fashion without curated rules or expert verification. We evaluate this possibility by calculating the precision and recall of our logistic model, shown in Figure 3. Over the last year our classifier surfaced 77% of known threats. Human experts agreed with our model's verdict 86% of the time. Overall performance has steadily increased over time with the addition of new features, an increasing training corpus, and increasingly frequent model retraining. Accuracy of the classifier during the final two weeks of our evaluation boasted 98% precision and 91% recall—on par with human experts. However, new threats always require human intervention as indicated by consistent drops in recall throughout time: while the model can quickly recover with daily retraining, we maintain that

Requested Permission	Precision	Recall
tabs	12%	84%
webRequest	23%	39%
webRequestBlocking	22%	27%
notifications	14%	27%
contextMenus	15%	26%
storage	9%	25%
webNavigation	21%	19%
cookies	10%	14%
unlimitedStorage	14%	13%
idle	27%	10%

Table 1: Top 10 permissions requested in extension manifest.

Chrome API	Precision	Recall
runtime.onInstalled	12%	79%
tabs.onUpdated	29%	61%
runtime.connect	21%	50%
extension.getURL	25%	34%
tabs.executeScript	47%	31%
tabs.query	31%	27%
runtime.onConnect	46%	25%
tabs.get	43%	24%
browserAction.setBadgeText	28%	23%
browserAction.setBadgeBackgroundCol...	39%	21%

Table 2: Top 10 Chrome API calls performed during dynamic execution.

DOM Operation	Precision	Recall
eval	10%	76%
Window.navigator	19%	59%
XMLHttpRequest.onreadystatechange	31%	56%
XMLHttpRequest.open	21%	53%
Document.createElement	20%	47%
Window.setTimeout	18%	46%
Node.appendChild	20%	45%
HTMLElement.onload	25%	30%
HTMLScriptElement.src	51%	25%
Window.location	23%	12%

Table 3: Top 10 DOM operations performed during dynamic execution.

Behavioral Signal	Precision	Recall
XHR Request	30%	52%
Code Obfucsation	21%	25%
Script Injected	50%	19%
HTTP 400 Error	41%	9%
Modifies CSP Headers	86%	2%
Uninstalls Extension	96%	0.5%
Prevents Uninstallation	100%	0.1%

Table 4: Precision and recall of individual behavioral signatures.

experts must always be part of our pipeline to minimize both false positives and false negatives. This is an immediate consequence of a centralized market for extensions where there are limited external sources of labeled training data. In contrast, email and telephony spam systems can rely on honeypots and informed users to readily generate representative daily training data. While our human throughput currently scales to the size of the Chrome Web Store, larger ecosystems face a significant challenge for sustainable accuracy.

6.4 Relevance of Individual Signals

WebEval is an amalgam of behavioral signals where no single feature captures the majority of malicious extensions. We examine assumptions we had of certain behaviors, whether they are unique to malware, and which signals are the most important to classification.

Requested Permissions: We list the most popular permissions used by malware and benign extensions in Table 1. These permissions include allowing an extension to trigger when Chrome creates a new tab (84% of all malware) or when Chrome generates a network request (39%). While these behaviors appear fundamental to malware they are equally prevalent in benign applications. This observation captures a significant limitation of the current Chrome permission model as applied towards security judgments: coarse permissions required by all extensions provide no indication that an extension is malicious. Similarly, 93% of all malicious extensions request to interact with every URL as do 57% of all other extensions. These broad contexts make it difficult to determine the pages an extension interacts with, further complicating dynamic analysis.

Chrome API Calls & DOM Operations: We find the strongest features for detecting malware originate from a mixture of Chrome API calls and DOM operations. We provide a list of the most common operations in Table 2 and Table 3. The majority of malware (and benign extensions) rely on injecting scripts, generating XHR requests, and adding new DOM elements that target newly created tabs. What distinguishes the two are the aggregate set of events triggered as well as the domains of remote resources loaded into a page (e.g., injected scripts or content). Our model effectively learns which resources are commonly fetched by malware in addition to common strategies for tampering with pages.

Malicious Logic: Recent work by Kapravelos *et al.* proposed a number of behavioral flags they deemed “suspicious” for extensions. We evaluate the effectiveness of

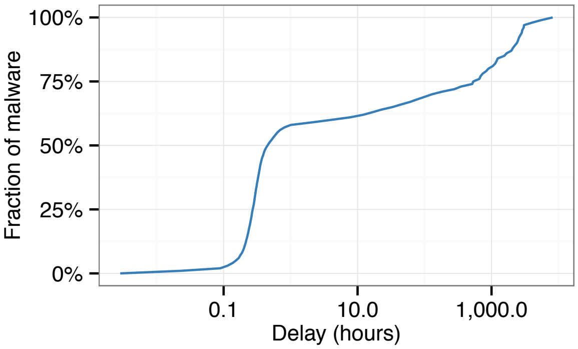


Figure 4: CDF of the delay before catching a malicious extension after it is first submitted to the Chrome Web Store. We catch malicious extensions within a median of 25 minutes.

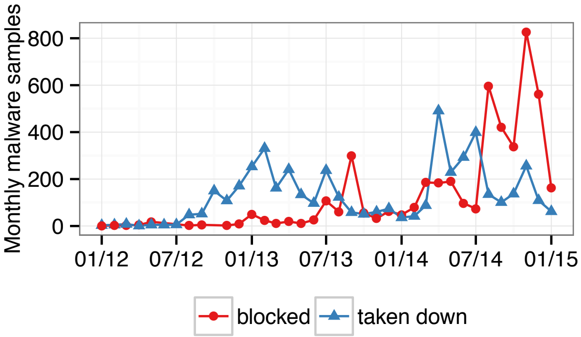


Figure 5: Actions taken against malicious extensions in the Chrome Web Store over time. Our systems are becoming increasingly proactive at blocking malware rather than reactive.

these signals in Table 4. We find that behaviors such as modifying CSP headers, uninstalling extensions, or preventing uninstallation are exclusive to malware, though rare. Contrastingly, Hulk's decision to surface extensions that produce network request errors or inject scripts would overly burden human experts due to low precision. These signals still have value, but they must be combined with the other features collected by our system in order to generate a precise verdict.

6.5 Detection Latency

A critical metric of WebEval's performance is our vulnerability window: the time between when a developer submits a malicious extension to the Chrome Web Store until its detection. This metric represents a worst case scenario where we assume an extension is malicious from its onset rather than after an update or a remote resource begins including malicious functions. Over the last year it took a median of 25 minutes before we flagged an extension as malicious—within the one hour window an extension is embargoed from public access. However, this delay has a long tail as shown in Figure 4. We catch 70% of malicious extensions within 5 days and 90% within 3 months. During this period, users are exposed to malicious content, the impact of which we evaluate in Section 7. Over time, our verdicts have become increasingly proactive rather than reactive as shown in Figure 5. Blocked extensions never reach the public, while extensions taken down by the Chrome Web Store leave users vulnerable for a short period. As we discuss shortly, proactive blocking has a substantial impact on reducing the number of known victims exposed to malware.

6.6 Manual Review Effort

WebEval relies heavily on human experts to validate the verdicts of automated classification and manual rules to guarantee high precision. In the last year, we surfaced 10,120 suspicious extensions for review, entailing a total of 464 hours of analysis—an average of 2.75 minutes per extension. This process is simplified by access to all of WebEval's dynamic and static analysis and concrete training features as previously discussed in Section 4 and Section 5. We recognize that manual review by experts represents a scarce resource that is challenging to scale. Consequently, we continuously look for ways to improve automated verdicts to achieve a precision on par with human experts.

7 Trends in Malicious Extensions

Consistently high recall over the last three years allows us to provide a retrospective on how malicious extensions have evolved over time. This includes the monetization vectors used, the breadth of users impacted, and the developers responsible.

7.1 Abuse Vectors

Despite hundreds of new monthly malicious extensions, we find the strategies for abusing Chrome users have remained largely constant. Figure 6 shows a breakdown of abuse strategies of extensions per month where a manually curated label is available;[4] we categorize extensions flagged by automated systems that provide no context on abuse vectors as "other". Noticeably absent from the top threats are banking trojans, password theft, and email spam. While these are all within the realm of a malicious extension's capabilities—and have cropped up from time to time—such threats are dwarfed by Facebook hijacking, ad injection, and information theft.

[4] Labels are not guaranteed to be unique; an extension can simultaneously hijack Facebook credentials, inject ads, and insert tracking pixels.

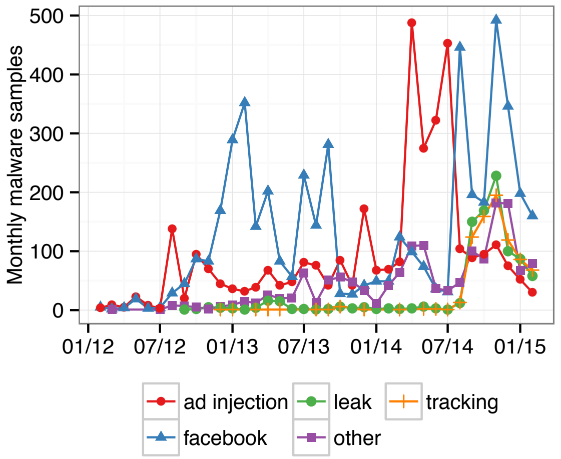


Figure 6: Malware varietals detected each month from 2012–2015.

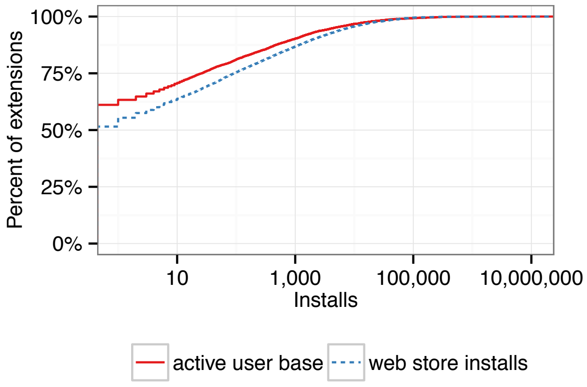


Figure 7: CDF of installs broken down by those originating from the Chrome Web Store and an extension's active pings that capture both store-based and sideloaded installs.

Facebook Hijacking: Our findings show that Facebook malware remains a persistent threat with over 4,809 variants in the last three years. These malicious extensions purport to offer enhancements such as removing the Facebook Timeline, adding a "dislike" button, or changing the theme of the Facebook interface. The hook has evolved over time. The latest rendition tricks users into installing an extension masquerading as a video codec required to view a video posted by a friend. Once installed, the extension hijacks the victim's credentials to post status updates that propagate the malware. How the extensions monetize Facebook accounts is not entirely clear, but appears to involve inflating likes, fans, and friend counts much like previously studied fake engagement contagions on Twitter [36, 38].

Ad Injection: Ad injection is the second most prevalent threat in the Chrome Web Store comprising 3,496 extensions. These extensions rely on content scripts that run on every page that allow the extension to scan DOMs for common banners to replace with rogue advertisements or simply insert new ads into pages. We note that ad injection is not expressly prohibited by the Chrome Web Store: the extensions flagged also performed some other malicious behavior or violated one of the store's policies as determined by a human expert.

Other Variants: In recent months we have witnessed a larger variety of abuse vectors. In depth investigations of a sample of these extensions reveal malware tampering with bitcoin wallets, injecting into banking sessions for Brazilian institutions, and modifying Amazon affiliate URLs. While we lack manual rules for these specific abuse vectors, we are nevertheless able to catch them via our classifier.

7.2 Installs

Malicious extensions obtain installs in one of two fashions: (1) via binaries that modify Chrome's user profile to sideload extensions,[5] or (2) via social engineering where miscreants direct users to the Chrome Web Store or prompt users with an install dialogue on a third-party site. We measure both approaches using two metrics. We define an extension's *active user base* as the total number of Chrome clients who ping the Chrome Web Store with update requests (sent by all extensions, including sideloaded extensions). This value changes each day as users install or uninstall extensions, so we select the all-time maximum. We define an extension's *web store installs* as the total number of install dialogues Chrome clients initiate with the Chrome Web Store. We note that a third option exists for miscreants to obtain installs: paying an existing, legitimate extension developer to hand over their app. In practice, we found only 6 malicious extensions (0.06%) that involved an ownership transfer.

Evidence of Side Loading: We provide a breakdown of both install metrics in Figure 7. We find that 51% of malicious extensions never received any active user base or Web Store installs due to early detection. Evidence of sideloading is relatively rare: only 290 extensions had a larger active user base than Web Store installs. However, these extensions were immensely popular with over 43.5 million combined active users. In contrast, all malicious extensions combined received 29.6 million installs via the Chrome Web Store. As such, it would appear that binary distribution of malicious extensions contributed substantially to user infections. This allows malware authors to rely on the same distribution models of the past (e.g., drive-by downloads [30], exploit packs [18], pay-per-install [8]) while tapping into the extension API as a means for simplifying exploitation.

[5] The extension still must be in the Chrome Web Store due to the lockdown policy discussed previously in Section 2

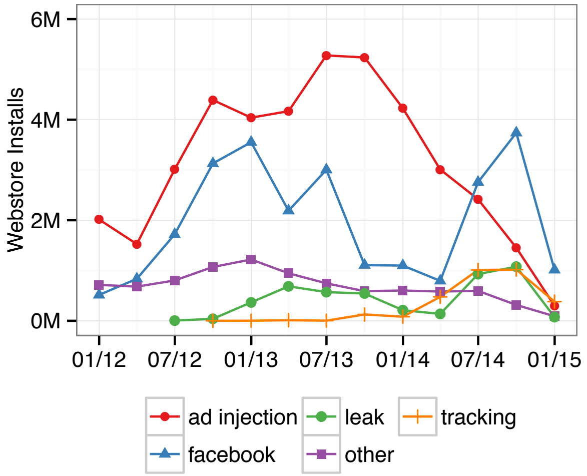


Figure 8: Malware installations via the Chrome Web Store for the past three years broken down by abuse vector. This excludes binaries sideloading extensions.

Country	Infected Users	Popularity
United States	2,375,363	8%
Brazil	1,982,570	7%
Mexico	1,942,288	6%
Colombia	1,634,933	5%
Turkey	1,569,949	5%
Argentina	1,525,364	5%
India	1,475,228	5%
Russia	1,244,932	4%
Peru	955,695	3%
Vietnam	806,625	3%

Table 5: Top 10 regions impacted by malicious extensions downloaded via the Chrome Web Store.

Equally problematic, installs follow a long tail distribution. We find that 64 extensions (1% of all malware) attained an aggregate 46.6 million active users, 83% of all installations. The top two most popular threats were ad injectors and search hijackers that each garnered over 10 million active users. Miscreants distributed each extension solely via binaries flagged as malware by Google Safe Browsing. Our results emphasize that seemingly small false negative detection rates can have substantial negative impact on Chrome users. This drastically differs from email and telephony spam where an incorrectly labeled message typically impacts only a single user—not millions.

Popular Social Engineering Campaigns: Focusing exclusively on installs mediated by the Chrome Web Store, we investigate which abuse vectors achieved the most new installs per month and the country of origin of installs. Figure 8 tracks the rise and fall of various monetization strategies over time. Despite a short window before catching malicious extensions we still find that social engineering campaigns enticed millions of new users each month to install Facebook malware, ad injection software, and information stealers. The downward trend in recent months is the result of proactive blocking rather than retroactive takedowns that expose users to malware for a short window. We find no single country is disproportionately represented as the source of installs, as shown in Table 5. Our results highlight the global scale and negative impact that malicious extensions have on users and the need for greater research attention to the problem.

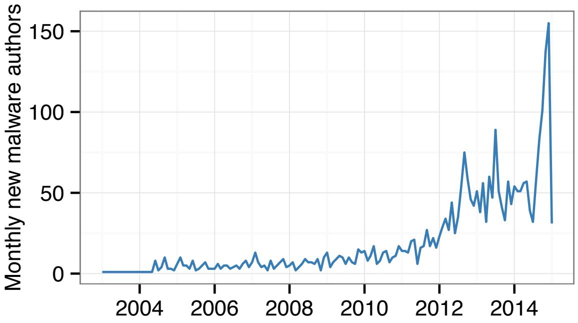


Figure 9: Registration time of malware authors. Most authors rely on accounts created in the last three years.

7.3 Malicious Developers

We identify 2,339 malicious extension developers throughout the course of our study. While 50% of developers authored their malicious extension within 3 to 4 months of registering, there is a long tail of potentially compromised accounts used to interact with the Chrome Web Store as shown in Figure 9. We find miscreants access 31% of developer accounts from IPs within Turkey followed in popularity by range of other countries detailed in Table 6. Many of the countries with the highest infection counts were also prominent locations for malicious developers indicating threats were likely localized. Re-use of malicious developer accounts was fairly limited: 50% of accounts authored fewer than 2 malicious extensions while 90% authored fewer than 10.

8 Discussion

With multiple years spent fighting malicious extensions, we reflect on some of the lessons we have learned, limitations of our approach, and potential technical and policy improvements that can help prevent malicious extensions in the future.

Country	Developers	Popularity
Turkey	552	31%
United States	164	9%
Dominican Republic	126	7%
Brazil	106	6%
Vietnam	83	4%
Russia	60	3%
Germany	43	2%
Peru	43	2%
India	43	2%
Israel	39	2%

Table 6: Top 10 login geolocations of malicious developers.

8.1 Lessons Learned

When we first set out to identify malicious extensions our expectation was to find banking trojans and password stealers that duplicated the strategies pioneered by Zeus and SpyEye. In practice, the abusive extension ecosystem is drastically different from malicious binaries. Monetization hinges on direct or indirect relationships with syndicated search partners and ad injection affiliate programs, some of which earn millions of dollars from infected users [37]. Miscreants derive wealth from *traffic* and *user targeting* rather than the computing resources or privileged access mediated via the browser. It may simply be that the authors of malicious binaries have little incentive (or external pressure) to change, leaving extensions to a distinct set of actors. This uncertainty is a strong motivation for exploring the extension ecosystem further.

A second lesson is the importance of equipping an abuse prevention team with the tools necessary to rapidly respond to new, unforeseen threats. As we have shown, even momentary lapses in protection have drastic consequences on Chrome users. This is especially true in the case of social engineering campaigns like those used to distribute malicious Facebook extensions that spread exponentially. We argue that evaluating a detection system purely on precision and recall is not effective when the ultimate goal is to protect users from malware installations. Instead, we must weigh false negatives by their consequences—the number of victims exposed to malware. In this light, our system has continuously improved over the last three years.

In the long term we believe the Chrome Web Store must extricate itself from the current fire-fighting approach to malicious extensions and outright disrupt the malicious actors involved. This reflects a nascent strategy within the research community to pursue criminal relationships such as those underpinning spammed pharmaceuticals [26]. However, to arrive at this point we must first lay a foundation for how to study the extension ecosystem. As the research community develops the necessary understanding this abuse space—and in particular the ad and search relationships involved—there must be a system to both protect users as well as generate longitudinal data on abuse strategies and their support infrastructure. WebEval satisfies both of these requirements.

8.2 Role of Policy

Research primarily considers technical solutions to abuse, but we argue that policy decisions prove equally effective at protecting users. When Chrome first released extensions there was no requirement of developers uploading their code to the Chrome Web Store. This enabled malicious developers to side-load extensions via binaries and left Chrome users with little room for discovering the installation or recourse. The subsequent Chrome lockdown forced all malicious extensions to at least be surfaced to the Chrome Web Store and created a homogeneous enforcement policy for all Chrome users. While binaries can still side-load extensions in the Chrome Web Store, WebEval now incorporates signals to detect organic versus silent installs.

It is worth noting the Chrome lockdown policy has some limitations. Anecdotally, we have observed binaries distributing payloads that overwrite the local content of legitimate extensions previously installed by a user. Because only the legitimate extension is in the store, WebEval cannot offer any protection. Chrome has since responded to this threat by introducing extension content verification, but this is just a single stage in an increasing arms race.

8.3 Limitations

Dynamic analysis and security crawlers consistently run the risk of overlooking malicious behaviors due to cloaking [2, 23, 31]. Extension analysis is equally vulnerable. Potential threats include malware delaying execution until after WebEval's evaluation; supplying benign versions of remotely fetched JavaScript until after evaluation; or malware developers fingerprinting our evaluation environment and IP addresses. A separate issue is code coverage: our behavioral suites are not guaranteed to trigger all of an extension's logic during evaluation. Worse, we face an intractably broad threat surface that we must test as the majority of malware requests access to every page a user visits. While symbolic execution systems exist for Javascript [32], they rely on fuzzing that is not guaranteed to trigger malicious behavior due to the implicit event-driven nature of extensions where activation requires a specific sequence of listeners to fire. Solutions to these challenges remain elusive; we currently rely on human experts and abuse reports to surface false negatives so we can adapt our detection framework.

8.4 Improving Detection

Fundamentally improving WebEval (and by proxy other security scanners) requires we break from evaluating extensions in a sandboxed environment vulnerable to cloaking and instead move to *in situ* monitoring of Chrome users. This strategy, previously considered by researchers to improve drive-by download detection [35], applies equally to malicious extensions. However, such a move creates a new challenge of balancing early infections of clients, user privacy, and anonymous but feature-rich reporting of an extension's behaviors with enough details to detect malice.

Furthermore, while we can retrain our a model of malicious extensions to incorporate client logs, the process would be immensely aided by the cooperation of website developers who label DOM resources as sensitive. We should take these labels as hints, not facts, to account for overzealous developers who label every DOM element as sensitive in an effort to dissuade extension modifications, even when desired by users. We believe this combined approach strikes the best balance between Chrome's current philosophy of allowing users to alter their browsing experience in any way with the necessity of early detection of malicious modifications.

9 Related Work

Security Sandboxes & Malware Detection: WebEval borrows heavily from a history of malware analysis sandboxes that capture system calls and network traffic. Examples include Anubis [5], CWSandbox [40], and GQ [24] among a breadth other architectures [12]. However, malicious extensions pose a unique set of challenges that limit the effectiveness of these sandboxes without modification. Unlike standalone applications, Chrome extensions run in the context of a webpage making it harder for traditional system-wide malware monitoring techniques to isolate malware activity from that of the browser. Our system manages to achieve this isolation by comparing extension activity to baseline activity captured while the extension was not running as well as by tapping natively into Chrome's JavaScript and API modules.

The closest system to our own is Hulk which captures in-browser activity logs [21]. Unlike Hulk, our system goes beyond identifying suspicious behaviors to return a concrete verdict of malice. This is imperative as the signals proposed by Hulk are insufficient at detecting most malicious extensions as we showed in Section 6. Research has also explored competing strategies such as information flow tracking in JavaScript with tainted inputs [11] or tracking common API calls made by Browser Helper Objects installed by adware [22]. These techniques influence our design but only capture a subset of the malicious extensions we identify.

Buggy & Malicious Extensions: Most research into browser extensions has focused on their security and permission model in light of the possible vulnerabilities [3, 4, 9, 19]. Only recently has research shifted towards the threat of outright malicious extensions. This includes re-imagining application-based attacks as man-in-the-browser threats [25]; examining the role of extensions in the ad injection ecosystem [37, 41]; and characterizing malicious extensions found in the Chrome Web Store [21]. Our observations agree with many of these former studies. We expand upon these works by offering a complete perspective of how malicious extension monetization techniques have evolved over the last three years and the techniques malware developers use to distribute extensions.

10 Conclusion

In this work we exposed wide-spread efforts by criminals to abuse the Chrome Web Store as a platform for distributing malicious extensions. As part of our study, we presented the design and implementation of a framework that automatically classifies an extension's behaviors, code base, and author reputation to surface malware. Due to our live deployment, this system cannot run in a fully automated fashion: we required regular inputs from human experts to correct for false negatives surfaced via Chrome user reports and manual investigations. Our unique combination of automated and human systems yielded a framework that identified 96.5% of all known malware submitted to the Chrome Web Store between January 2012–2015.

In total, we detected 9,523 malicious extensions that hijacked social networking sessions to generate synthetic likes, friend requests, and fans; ad injectors and affiliate fraudsters that rewrote DOM content to laden pages with additional advertisements; and information stealers that injected rogue tracking pixels and covertly siphoned search keywords. Despite a short window of operation—we disabled 50% of malware within 25 minutes of creation—a handful of under 100 malicious extensions were able to infect over 50 million users before removal. Our results highlight key challenges of protecting app marketplaces that are broadly applicable beyond the Chrome Web Store.

References

[1] Apple. App Review. `https://developer.apple.com/app-store/review/`, 2015.

[2] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Efficient de-

tection of split personalities in malware. In *Proceedings of the Network and Distributed System Security Conference*, 2010.

[3] Sruthi Bandhakavi, Samuel T King, Parthasarathy Madhusudan, and Marianne Winslett. Vex: Vetting browser extensions for security vulnerabilities. In *Proceedings of the USENIX Security Symposium*, 2010.

[4] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting browsers from extension vulnerabilities. In *Proceedings of the Network and Distributed System Security Conference*, 2010.

[5] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. In *Proceedings of the Network and Distributed System Security Conference*, 2009.

[6] Jeremy Bem, Georges R Harik, Joshua L Levenberg, Noam Shazeer, and Simon Tong. Large scale machine learning systems and methods, 2007. US Patent 7,222,127.

[7] Christina Bonnington. First instance of ios app store malware detected, removed. `http://www.wired.com/2012/07/first-ios-malware-found/`, 2012.

[8] Juan Caballero, Chris Grier, Christian Kreibich, and Vern Paxson. Measuring pay-per-install: The commoditization of malware distribution. In *Proceedings of the USENIX Security Symposium*, 2011.

[9] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. An evaluation of the google chrome extension security architecture. In *Proceedings of the USENIX Security Symposium*, 2012.

[10] Lucian Constantin. Malicious browser extensions pose a serious threat and defenses are lacking. `http://www.pcworld.com/article/2049540/malicious-browser-extensions-pose-a-serious-threat-and-defenses-are-lacking.html`, 2014.

[11] Mohan Dhawan and Vinod Ganapathy. Analyzing information flow in javascript-based browser extensions. In *Proceedings of the Annual Computer Security Applications Conference*, 2009.

[12] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys*, 2012.

[13] Erik Kay. Protecting Chrome users from malicious extensions. `http://chrome.blogspot.com/2014/05/protecting-chrome-users-from-malicious.html`, 2014.

[14] Firefox. Review Process. `https://addons.mozilla.org/en-US/developers/docs/policies/reviews`, 2015.

[15] Google. Developer registration fee. `https://support.google.com/chrome_webstore/answer/187591?hl=en`, 2015.

[16] Google. Google Chrome Web Store Developer Agreement. `https://developer.chrome.com/webstore/terms`, 2015.

[17] Google. Google Play Developer Program Policies. `https://play.google.com/about/developer-content-policy.html`, 2015.

[18] Chris Grier, Lucas Ballard, Juan Caballero, Neha Chachra, Christian J Dietrich, Kirill Levchenko, Panayiotis Mavrommatis, Damon McCoy, Antonio Nappa, Andreas Pitsillidis, et al. Manufacturing compromise: the emergence of exploit-as-a-service. In *Proceedings of the Conference on Computer and Communications Security*, 2012.

[19] Arjun Guha, Matthew Fredrikson, Benjamin Livshits, and Nikhil Swamy. Verified security for browser extensions. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2011.

[20] Scott Kaplan, Benjamin Livshits, Benjamin Zorn, Christian Siefert, and Charlie Curtsinger. " nofus: Automatically detecting"+ string. fromcharcode (32)+" obfuscated". tolowercase ()+" javascript code. In *Technical Report, Microsoft*, 2011.

[21] Alexandros Kapravelos, Chris Grier, Neha Chachra, Chris Kruegel, Giovanni Vigna, and Vern Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *Proceedings of the USENIX Security Symposium*, 2014.

[22] Engin Kirda, Christopher Kruegel, Greg Banks, Giovanni Vigna, and Richard Kemmerer. Behavior-based spyware detection. In *Proceedings of the USENIX Security Symposium*, 2006.

[23] Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Rozzle: De-cloaking internet malware. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.

[24] Christian Kreibich, Nicholas Weaver, Chris Kanich, Weidong Cui, and Vern Paxson. Gq: Practical containment for measuring modern malware systems. In *Proceedings of the ACM SIGCOM Internet Measurement Conference*, 2011.

[25] Lei Liu, Xinwen Zhang, Guanhua Yan, and Songqing Chen. Chrome extensions: Threat analysis and countermeasures. In *Proceedings of the Network and Distributed System Security Conference*, 2012.

[26] Damon McCoy, Hitesh Dharmdasani, Christian Kreibich, Geoffrey M. Voelker, and Stefan Savage. Priceless: The role of payments in abuse-advertised goods. In *Proceedings of the Conference on Computer and Communications Security*, 2012.

[27] Paul Pearce, Vacha Dave, Chris Grier, Kirill Levchenko, Saikat Guha, Damon McCoy, Vern Paxson, Stefan Savage, and Geoffrey M. Voelker. Characterizing large-scale click fraud in zeroaccess. In *Proceedings of the Conference on Computer and Communications Security*, 2014.

[28] Adrienne Porter Felt. See what your apps & extensions have been up to. `http://blog.chromium.org/2014/06/see-what-your-apps-extensions-have-been.html`, 2015.

[29] Emil Protalinski. Malicious Chrome extensions hijack Facebook accounts. `http://www.zdnet.com/article/malicious-chrome-extensions-hijack-facebook-accounts/`, 2012.

[30] Niels Provos, Panayiotis Mavrommatis, Moheeb Abu Rajab, and Fabian Monrose. All your iFRAMEs point to us. In *Proceedings of the USENIX Security Symposium*, 2008.

[31] M Rajab, Lucas Ballard, Nav Jagpal, Panayiotis Mavrommatis, Daisuke Nojiri, Niels Provos, and Ludwig Schmidt. Trends in circumventing web-malware detection. In *Google Technical Report*, 2011.

[32] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for JavaScript. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[33] Spark. Machine learning library (mllib) programming guide. `http://spark.apache.org/docs/1.4.0/mllib-guide.html`, 2015.

[34] Brett Stone-Gross, Marco Cova, Lorenzo Cavallaro, Bob Gilbert, Martin Szydlowski, Richard Kemmerer, Christopher Kruegel, and Giovanni Vigna. Your botnet is my botnet: Analysis of a botnet takeover. In *Proceedings of the Conference on Computer and Communications Security*, 2009.

[35] Gianluca Stringhini, Christopher Kruegel, and Giovanni Vigna. Shady paths: Leveraging surfing crowds to detect malicious web pages. In *Proceedings of the Conference on Computer and Communications Security*, 2013.

[36] Gianluca Stringhini, Gang Wang, Manuel Egele, Christopher Kruegel, Giovanni Vigna, Haitao Zheng, and Ben Y Zhao. Follow the green: Growth and dynamics in twitter follower markets. In *Proceedings of the ACM SIGCOM Internet Measurement Conference*, 2013.

[37] Kurt Thomas, Elie Bursztein, Chris Grier, Grant Ho, Nav Jagpal, Alexandros Kapravelos, Damon McCoy, Antonio Nappa, Vern Paxson, Paul Pearce, Niels Provos, and Moheeb Abu Rajab. Ad injection at scale: Assessing deceptive advertisement modifications. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[38] Kurt Thomas, Frank Li, Chris Grier, and Vern Paxson. Consequences of connectivity: Characterizing account hijacking on twitter. In *Proceedings of the Conference on Computer and Communications Security*, 2014.

[39] Kurt Thomas, Damon McCoy, Chris Grier, Alek Kolcz, and Vern Paxson. Trafficking fraudulent accounts: The role of the underground market in twitter spam and abuse. In *Proceedings of the USENIX Security Symposium*, 2013.

[40] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2007.

[41] Xinyu Xing, Wei Meng, Udi Weinsberg, Anmol Sheth, Byoungyoung Lee, Wenke Lee, and Roberto Perdisci. Unraveling the relationship between ad-injecting browser extensions and malvertising. In *Proceedings of the International Conference on the World Wide Web*, 2015.

[42] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of the Network and Distributed System Security Conference*, 2012.

Meerkat: Detecting Website Defacements through Image-based Object Recognition

Kevin Borgolte, Christopher Kruegel, Giovanni Vigna
University of California, Santa Barbara
{kevinbo,chris,vigna}@cs.ucsb.edu

Abstract

Website defacements and website vandalism can inflict significant harm on the website owner through the loss of sales, the loss in reputation, or because of legal ramifications.

Prior work on website defacements detection focused on detecting unauthorized changes to the web server, e.g., via host-based intrusion detection systems or file-based integrity checks. However, most prior approaches lack the capabilities to detect the most prevailing defacement techniques used today: code and/or data injection attacks, and DNS hijacking. This is because these attacks do not actually modify the code or configuration of the website, but instead they introduce new content or redirect the user to a different website.

In this paper, we approach the problem of defacement detection from a different angle: we use computer vision techniques to recognize if a website was defaced, similarly to how a human analyst decides if a website was defaced when viewing it in a web browser. We introduce MEERKAT, a defacement detection system that requires no prior knowledge about the website's content or its structure, but only its URL. Upon detection of a defacement, the system notifies the website operator that his website is defaced, who can then take appropriate action. To detect defacements, MEERKAT automatically learns high-level features from screenshots of defaced websites by combining recent advances in machine learning, like stacked autoencoders and deep neural networks, with techniques from computer vision. These features are then used to create models that allow for the detection of newly-defaced websites.

We show the practicality of MEERKAT on the largest website defacement dataset to date, comprising of 10,053,772 defacements observed between January 1998 and May 2014, and 2,554,905 legitimate websites. Overall, MEERKAT achieves true positive rates between 97.422% and 98.816%, false positive rates between 0.547% and 1.528%, and Bayesian detection rates[1] between 98.583% and 99.845%, thus significantly outperforming existing approaches.

1 Introduction

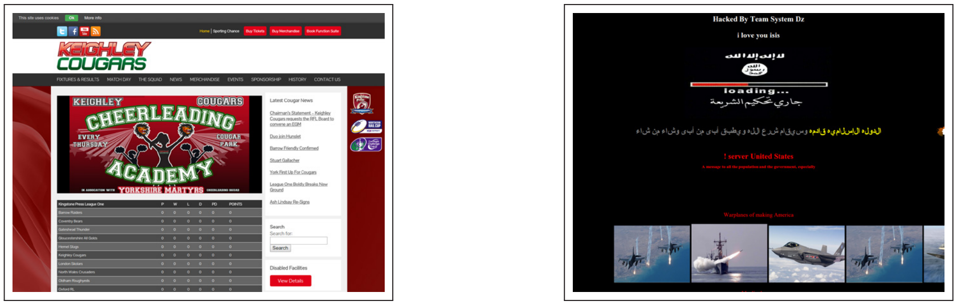
The defacement and vandalism of websites is an attack that disrupts the operation of companies and organizations, tarnishes their brand, and plagues websites of all sizes, from those of large corporations to the websites of single individuals [1–3].

In a website defacement, an attacker replaces the content of a legitimate website with some of his/her own content. A website might be defaced for many different reasons and in many different ways: For example, an attacker might deface the website by brute-forcing the administrator's credentials, by leveraging a SQL injection to introduce content or code, or by hijacking the domain name; however, all defaced websites share one characteristic: the defacer leaves a message that is shown to the visitors of the website instead of the legitimate content, changing the visual appearance of the website.

Although nearly all defacers vandalize websites for their "15 minutes of fame," and to get a platform to publicize their message, their messages vary: some embarrass the website's operator, others make a political or religious point, and others again do it simply for "bragging rights." For instance, in the beginning of November 2014, as reported by the BBC [4], attackers defaced the website of the Keighley Cougars, a professional rugby club from England competing in League 1. The defacers modified the website so that visitors were greeted with a message in support of the terrorist organization "Islamic State of Iraq and the Levant/Syria" (ISIL/ISIS). Similarly, in late 2012, defacers close to the Syrian regime defaced the homepage of the prominent Qatari television network Al Jazeera, and instead of being shown news articles, visitors were greeted by a message alleging Al Jazeera of "spreading false fabricated news." Reliably detecting such website defacements is challenging, as there are many ways in which an attacker can tamper with the website's appearance, including re-routing the traffic to a different website, which does not affect the legitimate website's content directly in any way.

In this paper, we introduce MEERKAT, a website monitoring system that automatically detects if a website has been defaced. MEERKAT detects website defacements by rendering the website in a browser, like a normal visitor

[1] The Bayesian detection rate is the likelihood that if we detect a website as defaced, it actually is defaced, i.e., P(true positive|positive).



Kingstone Press League One	P	W	L	D	PD	POINTS
Barrow Raiders	0	0	0	0	0	0
Coventry Bears	0	0	0	0	0	0
Gateshead Thunder	0	0	0	0	0	0
Gloucestershire All Golds	0	0	0	0	0	0
Hemel Stags	0	0	0	0	0	0
Keighley Cougars	0	0	0	0	0	0
London Skolars	0	0	0	0	0	0
North Wales Crusaders	0	0	0	0	0	0
Oldham Roughyeds	0	0	0	0	0	0
Oxford RL	0	0	0	0	0	0

Chairman's Statement – Keighley Cougars requests the RFL Board to convene an EGM

Duo join Hunslet

Barrow Friendly Confirmed

Stuart Gallacher

York First Up For Cougars

League One Boldly Breaks New Ground

Ash Lindsay Re-Signs

Hacked By Team System Dz

i love you isis

الدوله الإسلاميه قادمه وس يقام شرع الله و يطبق أبى من أبى وشاء من شاء

! server United States

A message to all the population and the government, especially

Warplanes of making America

(a) Normal, non-defaced version.

(b) Defaced version.

Figure 1: Screenshots of the Keighley Cougars homepage, non-defaced and defaced in an attack on November 2, 2014. (a) shows how the website looks normally, (b) shows how the defaced website looked like after being defaced by *Team System Dz*, a defacer group close to the terrorist organization Islamic State of Iraq and the Levant/Syria (ISIL/ISIS).

would, and deciding, based on features learned exclusively from screenshots of defacements and legitimate websites observed in the past, if the website's *look and feel* is that of a defaced or a legitimate website. If the website is detected as being defaced, the system notifies the operator, who, in turn, can, depending on the confidence in MEERKAT's decision, put the website (automatically) in maintenance mode or restore a known good state to reduce the damage.

Our technical contributions in this paper are:

- We introduce MEERKAT, a website defacement detection system that learns a high-level feature set from the visual representation of the website, i.e., it learns a compressed representation of the *look and feel* of website defacements and legitimate websites. Based on the learned features, the system then produces a model to differentiate between defaced and legitimate websites, which it uses to detect website defacements in the wild. In addition, the system notifies the website's operator upon detection (Section 3).
- We evaluate MEERKAT on the largest website defacement dataset to date, comprising of 10,053,772 website defacements observed between January 1998 to May 2014, and 2,554,905 legitimate and (supposedly) not defaced websites from Alexa's, MajesticSEO's, and QuantCast's top 1 million lists (Section 4).

In the remainder of this paper, we make a compelling case for the need of an accurate and lightweight website monitoring system that detects website defacements (Section 2), discuss how MEERKAT works in detail (Section 3), evaluate our system on the largest defacement dataset to date (Section 4), discuss some limitations of website defacement detection systems (Section 5), compare MEERKAT to related work (Section 6), and, finally, we conclude (Section 7).

2 Motivation

Lately, the detection of website defacements as a research topic has not received much attention from the scientific community, while, at the same time, defacements became more prominent than they have ever been. The number of reported defacements has been exceeding the number of reported phishing pages since October 2006 by a factor of 7 on average, and reached up to 33.39 defacements being reported to Zone-H[2] per phishing page reported to PhishTank[3] (see Figure 2). Yet, website vandalism is often played down as a problem instead of being acknowledged and addressed.

The increase in defacements is evident (see Figure 2): while a mere 783 verified defacements were reported on average each day to Zone-H in 2003, the number of reports increased to 3,258 verified defacements per day for the year 2012, to an all-time high of over 4,785 verified defacements being reported each day to Zone-H in 2014. This corresponds to an increase of websites being defaced by 46.87% from 2012 to 2014 [5].

Similarly, according to the Malaysian Computer Emergency Response Team (CERT), 26.04% of all reported incidents in 2013 were website defacements, but only 1.5% of the reported incidents were defacements in 2003, and 10.81% were website defacements in 2007 [7, 8].

Furthermore, in 2014, attackers confirmedly defaced over 53,000 websites ranked on Alexa's, MajesticSEO's, and QuantCast's top 1 million lists. Corroborating that not only websites that are "low-hanging fruit" are being defaced, but that high-profile ones are being attacked alike (see Table 1).

This recent resurgence and the increase in defacements and "cyber-vandalism" is generally attributed to the rise of hacktivist groups, like *anonymous* or *LulzSec* [9, 10], but also gained traction through the escalation of international conflicts [11, 12]. Although the scientific consensus is that the attacks employed to deface a website are usually rather primitive in nature [9], hacktivist groups and other politically- and religiously-motivated defacers have been extremely successful in the past: in February 2015, Google Vietnam was defaced by *Lizard Squad* for several hours [13]; in January 2015, the website of Malaysia Airlines was defaced by *Cyber Caliphate* [3]; in late 2014, the defacer group *Team System Dz* defaced over 1,700 websites to speak out against the actions of the US in the Syrian civil war and to advocate for ISIS/ISIL [2]; in April 2014, over 100 websites be-

[2] Zone-H [5] is an archive containing only defaced websites, all reported defacements are mirrored locally and manually verified [6]. Upon manual inspection, a reported defacement is removed from the archive if it does not constitute a defacement, or it is marked as verified.

[3] PhishTank is the largest public clearinghouse of data about phishing scams, users report potential phishing scams and other users agree or disagree with the submitter, resulting in a user-assigned phishing score. Phishing pages are not being verified by expert analysts.

Month	Website	Alexa		MajesticSEO		QuantCast	Page Views per Month ▼
		US	Global	TLD[1]	Global	US	
Nov 2014	princeton.edu	999	3,412	17	273	3,444	796,000
	volvo.com	54,607	57,046	3,757	7,323	568,058	-
	cca.gov.in[2]	146,039[3]	780,660	-	-	-	-
Aug 2014	openelec.tv	7,226[4]	48,754	184	93,894	-	-
	omicsonline.org	7,561[3]	42,030	5,068	63,924	-	-
Jul 2014	ct.gov	2,454	10,976	72	2,054	3,548	809,000
	us.to	2,846[5]	28,100	18	11,061	-	-
	sunnewsonline.com	68[6]	9,958	31,315	58,277	236,740	-
	newsmoments.in	3,725	39,262	-	-	-	-
Jun 2014	wordpress.net	3,522[7]	41,295	1,410	28,021	321,317	-
May 2014	arynews.tv	72[8]	5,308	949	536,436	-	-
	sundaytimes.lk	120[9]	38,591	6	39,866	209,083	-
Mar 2014	taylorswift.com	3,560	23,425	12,161	23,608	15,678	1.2 million
	gbjobs.com	798[10]	9,181	-	-	-	-
Dec 2013	openssl.net	5,994	16,409	80	933	-	-
Oct 2013	avg.com	117	155	471	854	-	37 million
	aljazeera.net	25[11]	1,831	37	920	2,196	28 million
	bitdefender.com	5,934	5,898	1,132	2,094	3,963	1.4 million
	avira.com	24[12]	1,108	1,275	2,361	6,081	480,000
	leaseweb.com	359[4]	4,035	23,585	44,451	230,626	-
	metasploit.com	124,365	175,570	33,537	59,816	120,839	-
2011-2013[14]	telegraph.co.uk	21[13]	225	3	107	6[13]	125 million
	ups.com	71	231	319	549	101	40 million
	nationalgeographic.com	483	1,006	94	139	125	37 million
	acer.com	4,060	6,042	-	-	1,995	2.9 million
	theregister.co.uk	2,737	3,457	443	14	11,327	1 million
	vodafone.com	7,052[13]	20,625	5,833	2,980	101,624	-

Table 1: Recent high-profile websites that were defaced, with their respective page rank according to Alexa, MajesticSEO, and QuantCast, and their monthly page impressions. These defacements were reported to Zone-H and include a major logistics company (UPS), computer and information security vendors (BitDefender, Avira, AVG, MetaSploit), news websites (Al Jazeera, Ary News, News Moments, Sunday Times, Sun News Online, Telegraph, The Register), a scientific society (National Geographic), a hardware vendor (Acer), the world's second largest telecommunications provider (Vodafone), a singer-songwriter/actress (Taylor Swift), the state of Connecticut (ct.gov), an Indian federal ministry (cca.gov.in), an auto-mobile company (Volvo), an ivy-league university (Princeton), well-known open source projects (OpenSSL, OpenELEC), and a hosting provider (Leaseweb). Missing fields represent unavailable data, data is unavailable due to being kept secret by the website operators or requiring subscriptions to Alexa, MajesticSEO or QuantCast.

[1] Top-level domain rank. [2] Government of India, Ministry of Communications & Information Technology. [3] Rank in India. [4] Rank in Netherlands. [5] Rank in Indonesia. [6] Rank in Nigeria. [7] Rank in Bulgaria. [8] Rank in Pakistan. [9] Rank in Sri Lanka. [10] Rank in China. [11] Rank in Yemen. [12] Rank in Iran. [13] Rank in United Kingdom. [14] Selected high-profile website defacements from Fortune 50 and Global 500 companies between 2011 to 2013.

longing to the government and major companies in Zambia were defaced by Syrian and Saudi Arabian defacers to voice against the Western world's *meddling* in the Syrian civil war [14]; in January 2014, the website of the popular mobile game Angry Birds was defaced in protest of governmental spying by the NSA and GHCQ [15]; and, in October 2013, a Pakistani defacer group gained access to the domain registrars of Suriname, Antigua & Barbados, and Saint Lucia and defaced the regional websites of Audi, AVG, BlackBerry, BMW, Canon, Coca-Cola, Fujitsu, Hitachi, Honda, IBM, Intel, Microsoft, Samsung, Symantec, Rolls-Royce, Vodafone, and other companies simply for "bragging rights" [16].

A prime example that quantifies the impact of defacements is the case of the Telegraph, a major UK daily newspaper, which was defaced in September 2011. The Telegraph is the third most-visited website in the United Kingdom, according to MajesticSEO, and it is the 21st most visited website in the United States, according to Alexa. Each month, the homepage of the Telegraph is visited over 125 million times (48 times per second), and, since reports state that the defacement lasted around three hours, we can estimate that more than 500,000 people saw the defacement instead of the legitimate website.[4]

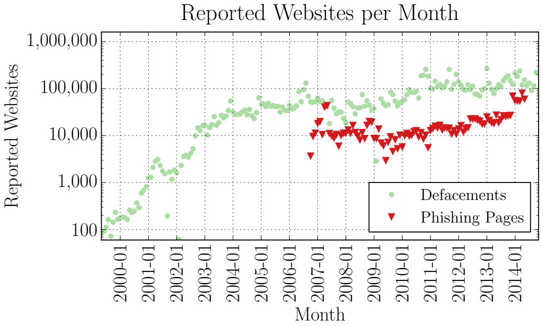


Figure 2: Defacements reported to Zone-H and phishing pages reported to PhishTank, per month from January 2000 to including October 2014. The drops in reported defacements in February 2002, February 2009, and March 2009 are because Zone-H was under maintenance during that time and did not accept any new reports. No data is available from PhishTank earlier than October 2006, when the website was launched. The trend of an increasing number of defacements per month, as well as the gap in the number of defacements to the number of phishing pages of a factor of up to 33x are evident.

While the list of prominent defacements goes on [4, 17–25], it is important to note that most techniques to deface a website, like code and data injection attacks (such as SQL injections), improper access control, or DNS hijacking and poisoning, have been well-studied and protection mechanisms have been proposed by prior

[4] Since the website was defaced on a Sunday afternoon local time in the United Kingdom, the number of visitors is likely much higher.

work [26, 27]. However, it is extremely hard to protect against all defacement attacks simultaneously and at scale.

Even worse, organizations are often responsible for hundreds (or thousands) of different websites, with different levels of security [9]. A single insecure website that is defaced, however, can inflict significant harm on the organization: in qualitative terms because of the loss of reputation, and in quantitative terms because of the cost of having to investigate and remove the defacement.

Although defacements can inflict serious harm on the website operator, a two-month study by Bartoli et al. [28] shows that many website operators still react slowly to defacements with an average response time of over 72 hours. Moreover, their study finds that mere 24% of the defaced websites were restored within one day, about 50% defacements were removed within the first week, while more than 37% of the websites remained defaced for over two weeks. Overall, their findings suggest that prior website defacement protection techniques and detection methods have not been widely adopted.

We argue that the logical first step is to reduce the harm inflicted on the website operator by quickly detecting if his/her website has been defaced, so that the operator can put the website in maintenance mode or restore its content to a known good state. As such, an automatic, accurate, and lightweight defacement detection system that monitors websites, notifies the website's operator, and acts as an early warning system is desired. In this paper, we propose one such system, MEERKAT.

3 Meerkat

The approach MEERKAT takes to detect website defacements is fundamentally different from prior work for three reasons. First, while the system does leverage machine learning for classification, it does not rely on handpicked features that were selected based on prior domain knowledge, i.e., it requires no feature engineering. Instead, MEERKAT relies on recent advances in machine learning, stacked autoencoders, to learn high-level features directly from data. Second, MEERKAT does not require the website operator to supply any information other than the domain name at which his/her website can be accessed. We designed our system in this way because other defacement detection systems that require the operator to define keywords and other metadata, provide a reference version of his/her website, or describe the website's legitimate content, have been rarely adopted in the past. By reducing the effort required from the website operator to actually use a defacement detection system, we hope to improve on this situation. Finally, MEERKAT approaches defacement detection visually: the system analyzes the *look and feel* of the website and how a user would experience it by rendering it in a web browser and analyzing a screenshot of the website, instead of analyzing its source code or content.

Approaching the problem of detecting website defacements visually has several advantages over analyzing the source code or content of a website: some defacements rely heavily on JavaScript and Cascading Style Sheets (CSS) to stylize the defacement, which all must be analyzed in an overarching browser context, and others again rely heavily on images. In fact, similar to spam, phishing, and many scams, defacements often do not contain much textual content, but include images to display text instead [29], thus they trivially evade text-based detection approaches. Furthermore, the source code of two websites can be vastly different, yet they appear the same to the human eye when rendered in a browser. Therefore, leveraging prior work, such as DELTA [30], to analyze the DOM-tree, the website's code, or parts thereof, is unlikely to be successful when trying to detect website defacements accurately, which is why we opted for a perceptual approach that does not suffer from the aforementioned problems.

Following, we describe how MEERKAT learns from defacements and legitimate websites, and how it detects defacements in the wild. Next, we motivate the structure of our deep neural network briefly, then, we discuss the concept and motivation of fine-tuning the network, then, we provide some notes on our implementation, and, last, we briefly recap how MEERKAT can be deployed in practice.

3.1 Training and Detection

Before MEERKAT can be trained, two crucial parameters must be selected that determine how and from what data the system learns the *look and feel* of defacements:

Window Size. MEERKAT is not trained on whole screenshots of websites, but on a window "into" each website (i.e., only a part of the screenshot), thus we must select the size of these *representative windows*. Some important considerations must be made before picking the size of the windows that we extract.

A small window can be more accurate because it might only contain the exact representative part of the defacement but not any noise, like an irrelevant background color. However, if the windows are too small, the system will also have more false positives because the windows are not representative of defacements; instead, they are representative for only parts of the defacements, which might also occur in legitimate websites.

On the other hand, when using larger windows, it will take significantly longer to train the network initially, but the network might learn a more accurate model. However, if the windows are too large, then the system will learn about specific kinds of defacements in-detail and overfit; e.g., the system might learn that two defacements are different, while the two defacements are actually the same but have a slightly different, dynamically-generated background image.

Considering the trade-offs for different window sizes, for our implementation, we decided to extract windows that are 160×160 pixels in size. Our evaluation later shows that this window size works well in practice to detect website defacements (see Section 4). We briefly explored other window sizes, like 30×30, that fared worse.

Window Extraction Strategy. The strategy to extract the representative window from a screenshot is fundamental to learn the *look and feel* of defacements and legitimate websites. If the windows are extracted according to some poorly chosen strategy, then we expect the classification accuracy to be poor as well. For instance, if the strategy always extracts the part of a website that is just a plain background, the system will only detect plain backgrounds. Therefore, it is crucial that the window extraction strategy is chosen well, and we compare some suitable strategies, like extracting the window always from the center or at random, later (see Section 3.1.2).

After selecting these parameters carefully, the system can be trained. This is where most of the complexity of MEERKAT lies. The training phase works as follows:

1. We collect a considerable amount of labeled website defacements and legitimate websites, and we extract their graphic representation (i.e., a screenshot of the browser window; Section 3.1.1)

2. For each sample, we extract the `160×160` representative window from each screenshot according to the selected extraction strategy (Section 3.1.2).

3. The representative windows are first used to learn the features of our approach, and then to learn the model for classification, for which we use a neural network (Section 3.2).

Once the neural network is trained, MEERKAT detects defacements in the wild. Its detection phase consists of only two steps, on which we expand later:

1. The website is visited with a browser to retrieve a representative screenshot (Section 3.1.1).

2. A sliding window approach is used to check if the website is defaced and, if so, an alert is raised (Section 3.1.3).

3.1.1 Screenshot Collection

The first step to detect if a website has been defaced based on its *look and feel* is to collect a screenshot of how the website looks for a normal visitor. MEERKAT visits the website with a browser that renders the website like any other browser would, and takes a screenshot once the browser finished rendering the website. In our implementation, we use PhantomJS to collect the screenshots of the websites. PhantomJS is a headless browser based on the Webkit layout engine that renders websites (nearly) identical to Safari or Google Chrome. PhantomJS also executes included JavaScript code, renders Cascading Style Sheets (CSS), and includes dynamic content, such as advertisements, like a browser that a human would use.

Another important aspect in collecting a representative screenshot of a website with a headless browser is the resolution of the *simulated* screen. The resolution of the display is important when collecting screenshots because many websites render differently for different screen sizes, such as for mobile devices, tablets, small notebooks, or large displays. In our case, we decided to fix the resolution to `1600×900` pixels, which is a display resolution often found in budget and mid-range notebooks.

3.1.2 Window Extraction Techniques

For training the system, after collecting the screenshots, we need to extract a representative window from each screenshot so that we can train the neural network to detect defacements. Various techniques can be used to extract the representative window, which can be grouped into deterministic and non-deterministic techniques. Hereinafter, we discuss the trade-offs for four possible techniques: (i) selecting the center window, (ii) selecting n non-overlapping windows according to some measure (explained later), (iii) uniformly selecting the window at random, and (iv) randomly sampling the window's center from a Gaussian distribution for the x and y dimension separately.

Deterministic Window Extraction

The most straightforward deterministic technique is to always extracts the window from the center of the screenshot of the website. However, this makes evading the system trivial. Generally, if an attacker can accurately predict the window that will be extracted, he can force the system to learn about defacements poorly, and, in turn, deteriorate classification performance drastically. Therefore, such a simple technique is unsuitable for a detection system in an adversarial context.

Alternatively, one can extract the window according to some measure. Identifying the most representative window according to a measure (e.g., the Shannon entropy), however, forces us to compute it for all possible windows and then pick the top ranking one. In turn, for a `1600×900` screenshot and a `160×160` window, we would need to evaluate over 1 million candidate windows for each sample in the dataset. In total, for our dataset, this would require over 13 *trillion* computations of the measure just to extract the representative windows. Clearly, this is impractical.

Nonetheless, a deterministic selection strategy based on a clever measure can increase the accuracy of the system, and it can also be extended trivially to extract multiple top-ranking windows at no additional cost. However, using more than one window per sample increases the dataset size by a factor of n and prolongs training time. Therefore, n would have to be chosen carefully.

Taking into account the trade-offs the different deterministic extraction strategies bear (increased training/detection time, ease of evasion, or computationally impractical) and considering that a comprehensive evaluation of them would require at least an order of magnitude of additional experiments,[5] we decided to select a non-deterministic extraction strategy that follows intuition and is based on user interface and user experience design principles instead. This selection makes our classification performance a lower bound: other window extraction strategies might be more accurate and/or robust, but (at the same time) they also incur significant additional cost at training and/or detection time.

[5] Performing these additional experiments would require at least 6 months just in computational time on our current GPU infrastructure, which is why we decided against performing them.

Non-deterministic Window Extraction

A straightforward non-deterministic strategy to extract a window from a screenshot is to select it uniformly at random. However, one cannot simply take any point from the website's screenshot as the center of the window. Instead, it must be sampled so that the whole window contains only valid data, forcing us to sample its center from the interval $[80, 1520]$ for x and $[80, 820]$ for y (these intervals are specific to the screenshot (`1600×900`) and window size (`160×160`)). Therefore, pixels at the border have a slightly lower probability to occur in a window than those in the center. Although this is an unintended side effect, it has negligible impact in practice because the center of a website is more likely to be descriptive anyways. Alternatively, we could create an "infinite" image by wrapping the screenshot at its borders, which would, however, yield artifacts because we would combine parts of the top of the website with parts of the bottom (and left and right, respectively), resulting in windows that do not occur on the real website, which, in turn, might disturb or confuse detection.

Alternatively to selecting the window's center uniformly at random, one can sample it from any other distribution, discretizing the sampled point. For instance, from a Gaussian distribution to extract windows from mostly the center of the screenshot, but not extracting from it exclusively. A focus on the center of the website is often desirable because it is likely to be more descriptive of the website's *look and feel*. For robustness, however, we also want to the system to not learn exclusively from the center but to also learn about defacements that occur at the border of the website. Therefore, for our implementation, we extract a single window per website with a Gaussian extraction strategy with $\mu_x = 800$ and $\sigma_x = 134.63975$ for x and $\mu_y = 450$ and $\sigma_y = 61.00864$ for y, so that the windows at the border of the screenshot have a lower probability to be sampled but are not ignored completely. If x and y values outside of the screenshot are sampled, we simply resample the value for x or y respectively. We selected the μ and σ values this specifically so that we sample values outside of the screenshot only with likelihood 0.0001%.

3.1.3 Defacement Detection

After MEERKAT has been trained on a set of extracted windows, it can detect if a website has been defaced. Detecting website defacements with MEERKAT is conceptually extremely simple:

1. We visit the website that we want to check with our browser and we take a screenshot of the rendered website (Section 3.1.1).

2. We apply a standard sliding window detection approach on the screenshot we took to check if a part of the screenshot is detected as being defaced, similarly to prior work in image classification [31].

3. If a window is detected to be a defacement by MEERKAT, we raise an alert and inform the website operator that his/her website has been defaced.

Note that MEERKAT does not compare a possibly-defaced website to an older, legitimate version of it, and, thus, does not need to analyze or store an older version. Instead, it detects defacements solely by examining how the current version looks like.

Exclusively to improve performance, instead of starting in a corner of the screenshot, our system starts in the center and moves outward. This behavior is motivated by the fact that the center of the website is likely more descriptive, and our training set was focused on the center region of the screenshots. This does not mean, however, that MEERKAT misses defacements that are at the border of a website, they will be detected when the sliding window reaches the actually-defaced part, the border. The same is also true if a website is only partially defaced: once the sliding window reaches the defaced area, MEERKAT detects that the website is defaced.

Additionally, a special case worth mentioning is that a legitimate website might show a large promotional screen or an advertisement with the same intention of a website defacer: attracting attention. In turn, such a promotional screen might be similar in its *look and feel* to that of a website defacement. While MEERKAT might currently (theoretically) mislabel them as defacements, our evaluation shows that they do not matter much (see Section 4). Furthermore, if they start to matter at one point in the future, it is straightforward to consider them: the defacement engine can make use of an advertisement blocker, and the website operator could whitelist the system to not be shown any promotional screens.

3.2 Neural Network Structure

In this section, we briefly discuss the design of our deep neural network and how the different layers of the network interact with the input image. The structure of our deep neural network was notably inspired by prior work by Le at al. [32], Krizhevsky et al. [33], Sermanet et al. [31], and Girshick et al. [34]. We refer to them for further details.

The main components of our deep neural network are autoencoders, which we stack on top of each other, and a standard feed-forward neural network. Autoencoders are a special type of neural network that are used for unsupervised learning. The goal of an autoencoder is to find a compressed, possibly approximated encoding/representation of the input, which can be used to remove noise from the input, or, when autoencoders are stacked, they can learn high-level features directly from the input, like where edges in an image are, or if cats or human faces are part of an image [32].

Overall, the structure of our deep neural network is based on the following idea: first, we use a stacked autoencoder to denoise the input image and learn a compressed representation of both defaced and legitimate websites, i.e., we leverage the stacked autoencoder to learn high-level features, similar to Le et al. [32]; second, we utilize a feed-forward neural network with dropout for classification, similar to Krizhevsky et al. [33].

The initial layer of our stacked autoencoder is comprised of local receptive fields. This layer is motivated by the need to scale the autoencoders to large images [32, 35–38], this layer groups parts of the image to connect to the next layer

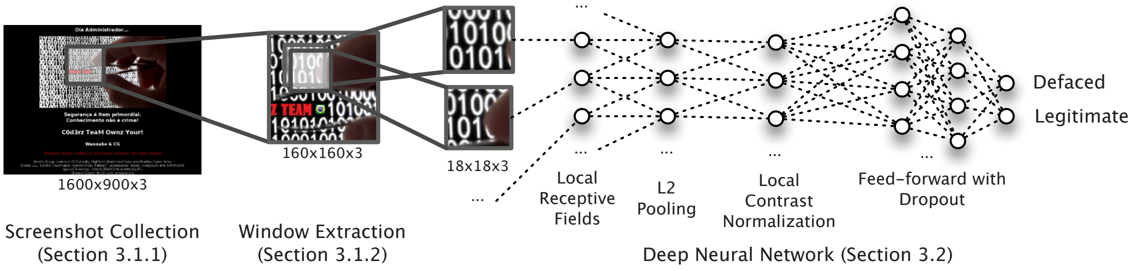

1600x900x3
160x160x3
18x18x3
Local Receptive Fields
L2 Pooling
Local Contrast Normalization
Feed–forward with Dropout
Defaced
Legitimate
Screenshot Collection (Section 3.1.1)
Window Extraction (Section 3.1.2)
Deep Neural Network (Section 3.2)

Figure 3: Architecture of our deep neural network.

of the autoencoder, instead of allowing the whole image to be used as input to each node of the following layer. It takes 20,164 (142^2) sub-images of size 18×18 as input, extracted at a stride of 1 from the 160×160 representative window (see Figure 3; note that each pixel in each sub-image has three dimensions for the three colors: red, green, and blue). The second layer of our stacked autoencoder employs L2 pooling to denoise local deformations of the image and to learn invariant features [37, 39–41]. Finally, the last layer of our autoencoder performs local contrast normalization for robustness [42].

The output of the stacked autoencoder is then used as the input to a feed-forward neural network with dropout that provides a 2-way softmax output. The 2-way softmax output corresponds to the two classes that we want to detect: defaced websites and legitimate websites. We use dropout in our deep neural network to prevent overfitting of the network, and to force it to learn more robust features by preventing neurons to rely that other neurons of the network are available (i.e., to prevent the co-adaptation of neurons) [43].

3.3 Fine-Tuning the Network's Parameters

In an adversarial context, such as when trying to detect if an attacker defaced a website, concept drift can be introduced intentionally by the attacker and impede the accuracy of the detection system drastically. Furthermore, concept drift also occurs naturally, such as when the style of defacements evolves over time in such a way that the features cannot distinguish between legitimate and defacement anymore. Therefore, concept drift can be a severe limitation of any detection system, if it is not taken into account and addressed properly (see Section 5.1).

MEERKAT can deal with concept drift in two different, fully-automatic ways: fine-tuning the network's parameters (adjusting feature weights), and retraining the entire network on new data. While the latter is conceptually straightforward and addresses all kinds of concept drift, it is computationally very expensive. The former, on the other hand, allows us to deal with some forms of concept drift gracefully and is computationally much less expensive. However, it requires some further attention: when fine-tuning the neural network, MEERKAT does not learn new features, but adjusts how important the already learned features are. Therefore, fine-tuning cannot address major concept drift for which the already learned features do not model defacements accurately anymore. Instead, when we fine-tune the network's parameters, we adjust the already learned weights of the deeper layers of the neural network so that new observations of defacements and legitimate websites are classified properly. As such, fine-tuning the network to maintain an accurate detection performance requires no additional information about the websites at all, but only defacements and legitimate websites that were not part of the training set before.

Conceptually speaking, when fine-tuning the network given new defacements and legitimate websites, we search for a better and, given the new data, *more optimal* set of weights in the space of all possible weights. To do so more efficiently, instead of initializing the weights at random, we initialize them based on the previously-learned weights.

3.4 Implementation

For this paper, we implemented a prototype of MEERKAT using Python and the "Convolutional Architecture for Fast Feature Embedding" (Caffe) framework by Jia et al. [44]. Caffe was used because of its high-performance and ease of use, however, it does not offer all functionality that our neural network requires and some modifications were made.

Overall, the general architecture of MEERKAT is embarrassingly parallel: the screenshot collection engine is completely separate from the detection engine except for providing its input. For instance, to quickly collect the screenshots of all websites, we utilized 125 machines (with 2 cores and 2 GiB memory each), and collection peaked at about 300 screenshots per second. Similarly, once the neural network has been trained, the learned parameters can be distributed to multiple machines and detection can be scaled out horizontally, and, although the system is trained on a GPU, once trained, the detection engine does not require a GPU and can run on common CPUs instead.

Training the system, on the other hand, is not parallelized to multiple machines yet, but some clever tricks can be used to reduce training time significantly [33], which we leave for future work.

3.5 Real-world Deployment

MEERKAT's main deployment is as a monitoring service, acting as an early warning system for website defacements, to which a website operator subscribes with only the URL at which his website can be reached. For each monitored website, the system regularly checks, such as every few minutes (or even seconds), that the website is not defaced. If it detects it as being defaced, it notifies the website's operator, who, in turn, depending on the confidence in the warning, manually investigates, or automatically puts the website in maintenance mode or restores a known good state. Acting as an early warning system, MEERKAT reduces the reaction time to defacements from hours, days, and even weeks (see Section 2) down to minutes (or even seconds), and, therefore, it reduces the damage inflicted on the website's operator by the defacement significantly.

Furthermore, MEERKAT can also reduce human labor: currently, Zone-H manually vets all submissions for defacements [6], of which nearly two thirds are invalid. MEERKAT automates this significant amount of work.

4 Evaluation

We evaluate our implementation of MEERKAT in various settings. However, first, we provide details on what data our dataset is comprised of, and how we partition it to simulate various defacement scenarios.

Our evaluation scenarios are traditional and simulations of real-world events, such as a new defacer group emerging, or how the system's accuracy changes over time, with and without fine-tuning the neural network.

In our experiments, a true positive is a website defacement being detected as a defacement and a true negative is a legitimate website being detected as legitimate. Correspondingly, a false positive is a legitimate website that is being detected as being defaced, and a false negative is a defacement being detected as being legitimate.

4.1 Dataset

The dataset on which we evaluate MEERKAT contains data from two different sources. First, it includes a comprehensive dataset of 10,053,772 defacements observed from January 1998 to May 9, 2014; we obtained this data through a subscription from Zone-H, but it is also freely available from `http://zone-h.org` under a more restrictive license. From those defacements, 9,258,176 defacements were verified manually by Zone-H [6]; the remaining 795,596 website defacements were pending verification and we do not include them in our dataset. Second, our dataset contains 2,554,905 unique (supposedly) undefaced websites from the top 1 million lists from Alexa, MajesticSEO, and QuantCast.[6] Note that we cannot be certain that the legitimate websites in our dataset are not defaced, and since manual verification is impractical at such a large scale, the true negative rate is actually a lower bound and the false positive rate is an upper bound, correspondingly. In layman's terms: the system might be more accurate than our results suggest.[7]

To accurately evaluate the classification performance of MEERKAT in a real-world deployment, we report its accuracy in three different scenarios:

- Traditional, to compare to prior work, i.e., by performing 10-fold cross-validation by sampling from all data uniformly at random, so that each bin contains 925,817 defacements and 255,490 legitimate websites.
- Reporter, to simulate a new defacer emerging, i.e., by performing 10-fold cross-validation on the reporters of a defacement and including only their defacements in their respective bin; legitimate website are sampled from the legitimate data uniformly at random.
- Time-wise, to evaluate the practicality of our approach in a real-world setting, i.e., we start by training the system on all data from December 2012 to December 2013, and, then, we detect defacements from January to May 2014. We report the system's detection accuracy for each month.

We evaluate our system in these settings to prevent a positive skew of our results that might be the result of the different evaluation method and how the dataset is composed. For instance, a reporter of a defacement might introduce an inherit bias to the distribution of the defacement by only reporting the defacements of one specific defacer (such as themselves), or there might be a bias in how defacements and how the web evolved. Those potential pitfalls might skew the results positively or negatively and must be considered for an accurate comparison to prior work.[8]

Finally, to account for the difference in the number of samples of the legitimate websites (2,554,905) and defaced websites (10,053,772), we report the Bayesian detection rate [45]. The Bayesian detection rate is normalized to the number of samples and corresponds to the likelihood if we detect a website as being defaced, it is actually defaced (the likelihood of a positive prediction being correct, that is a true positive; i.e., P(true positive|positive)).

4.2 Features Learned

The features that MEERKAT learns depend on the data it is being trained on. Although one can treat the system as a black-box and not worry about its internal details, understanding how it comes to its final decision helps one to reason about its robustness and to understand how difficult the system is to evade or to estimate when the system must be retrained to retain its accuracy. In our experiments, MEERKAT learned various features automatically and directly from image data, of which we manually grouped some on a higher, more conceptual level together. We manually identified the learned features by evaluating which representative windows activate the same neuron of the neural network, i.e., which windows trigger the same feature to be recognized by MEERKAT. Note that all the features we discuss hereinafter have been learned *automatically* from data and no domain knowledge whatsoever was required to learn and use these features; yet, the overlap with features that an analyst with domain knowledge would use confirms the prospects of feature/representational learning for website defacement detection. Some of the learned features can be best described as:

Defacement group logos. MEERKAT learned to recognize the individual logos of some of the most prolific defacement groups directly (see Figure 4). Clearly, the

[6] We made a list of all 2,554,905 legitimate websites included in our dataset available at `http://cs.ucsb.edu/~kevinbo/sec15-meerkat/legitimate.txt.bz2`.

[7] Over 191,000 website in our legitimate dataset have been defaced at one point in the past, thus, it is likely that some of them are actually defaced and therefore mislabeled; thus, if classified correctly as a defacement by MEERKAT, they appear as false positives in our results.

[8] We cannot compare prior work on our dataset directly as they do not scale to its size, and we cannot compare on their datasets because they are too small to train MEERKAT accurately (see Section 6.1).

logos of the defacer groups themselves are extremely descriptive of website defacements because they are very unlikely to be included in legitimate websites.

Color combinations. MEERKAT also learned to recognize unique or specific color combinations indicative of legitimate and defaced websites, including but not limited to one of the most prominent combinations: bright red or green text on a black background, which is an often used color combination by defacers, but rarely seen on legitimate websites. On the other hand, small black text on a white or brightly colored background is being consulted as a non-definitive indicator for a legitimate, non-defaced website.

Letter combinations. Interestingly, defacers often not only mix colors, but also mix characters from different alphabets right next to each other, such as Arabic or Cyrillic script being mixed with Latin script, to promote their message in both their native language and also in English as the web's *lingua franca*. Additionally, sometimes the defacement contains characters in a character set encoding specific to the defacer's native language, like ISO-8859-13 for Baltic languages or Windows-1256 for Arabic. As such, characters appear differently or are replaced by special characters if the browser does not support it, or if the website does not specify the character set and if the browser's fallback is different (like in our case, as we fall back to UTF-8), resulting in a *look and feel* that is descriptive of defacements, and, correspondingly, it was automatically learned by MEERKAT.

Leetspeak. Similarly to letter combinations, MEERKAT learned that defacers often use "leetspeak," an English alphabet in which some characters are replaced by numbers or special characters (e.g., "leetspeak" as "1337sp34k") and in which some words are deliberately misspelled ("owned" as "pwned," "the" as "teh," or "hax0red" instead of "hacked"). Defacers often use leetspeak to discern themselves from "common folks," and to show that they are "elite" and special, which, in turn, makes it often a good indicator that a website has indeed been defaced.

Typographical and grammatical errors. While some typographical mistakes are deliberate (as in the case of leetspeak, see above), many defacers make other unintentional typographical and grammatical mistakes, which rarely occurred on the legitimate websites in our dataset. Many defacers make these mistakes most likely because they are not native English speakers (the country of the reporter of the defacement, part of the meta-data in our dataset, suggests that most defacers do not speak English as their first language). MEERKAT learned to detect some of these mistakes at training and values them as a supporting indicator of a website defacement. Some of the examples of (supposedly) unintentional typographical and grammatical errors include "greats to" (instead of "greets to"), "goals is" (instead of "goals are"), or "visit us in our website" ("visit us at our website" or just "visit our website").

Note that, since MEERKAT works on image data, the system is unaware that it analyzes text and the textual features, such as unique letter combinations, leetspeak, or typographical and grammatical errors, are actually being evaluated on rendered text. As such, it seems likely that the textual features are specific to the font, possibly overfitting on the specific font type. However, we manually confirmed that the system actually learned a more robust feature and is not overfitting: it combines slight variances in the font family and size in a single high-level feature. Furthermore, given the sliding window approach MEERKAT employs for detection, the features are also completely independent of the position of the text in the representative window and website.

While some of the learned features can be evaded theoretically, evading them almost always contradicts the defacer's goal: making a name for themselves in the most "stylish" and personalized way possible, thus, it is unlikely that these features will change drastically in the near future. Furthermore, MEERKAT also consults features that were not as easy to discern into high-level feature groups manually, such as artifacts unique to legitimate or defaced websites, or features that are indicative for one group but are not definitive because they might appear more often in defaced websites, but also sometimes legitimately. MEERKAT can also be retrained easily and new features are learned *automatically* once the old features do not model defacements accurately anymore (i.e., if the concept of a defacement drifted significantly). Finally, since MEERKAT uses a non-linear classifier to combine those features, it can learn more complex models about defacements and legitimate websites, and simply evading only some features will not be sufficient to evade detection.

Interestingly, some of the high-level features (letter and color combinations) that MEERKAT learned automatically from data have been leveraged to a smaller degree by prior work [46, 47] (through manual feature engineering), while others (logos, leetspeak, and typographical mistakes) had not been utilized yet. Further suggesting that representation learning and inspection of the learned features can yield important insight into security challenges that were dominated by feature engineering in the past, such as intrusion, malware, or phishing detection.

4.3 Traditional Split

First, for an accurate comparison to prior work, we evaluate MEERKAT on our dataset using 10-fold cross-validation, i.e., we split the dataset into 10 bins that contain 925,817 website defacements and 255,490 legitimate websites each. Note that we discard 6 website defacements and 5 legitimate websites from our dataset at random to have the same number of samples in each bin. Next, for each bin, we train the system on the other 9 bins (training bins) and measure its classification performance on the 10^{th} bin (test bin). Considering the 10 different 90% training and 10% test-set partitions of our dataset separately, MEERKAT achieves true positive rates between 97.422% and 98.375%, and false positive rates ranging from 0.547% to 1.419%. The Bayesian detection rate is between 99.603% and 99.845%.

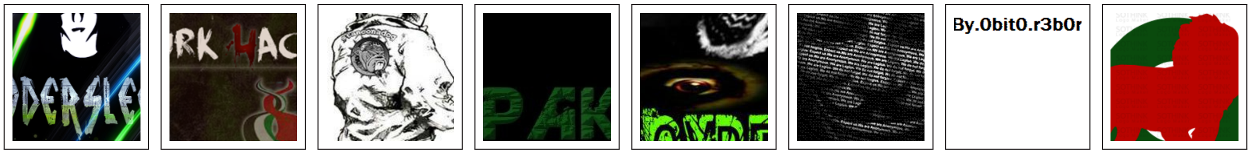


Figure 4: Defacement Group Logos. Example representative windows of logos of defacement groups that MEERKAT learned to recognized to be a significant indicator for defacements. Note that MEERKAT also recognizes variations and that there are many other features used for classification.

More interestingly, as a partition-independent measure of the system's classification performance, the average true positive rate is 97.878%, the average false positive rate is 1.012%, and the average Bayesian detection rate is 99.716%. If MEERKAT detects a defacement and raises an alert, with likelihood 99.716% it is a website defacement. Therefore, MEERKAT is significantly outperforming current state-of-the-art approaches.

4.4 Reporter Split

For the reporter split, we partition our dataset by the reporter of the defaced website. We deliberately designed the experiment this way to show that MEERKAT is not overfitting on specific defacements, which our results verify.

While a partition by reporter might seem counter-intuitive at first, it becomes clear that such a split is meaningful and that it can be used to evaluate that a new defacer group emerges once it is taken into account that these groups often have unique defacement designs and that defaced websites are most often reported by the defacers themselves. Therefore, if we split by reporter, we are practically splitting by defacer group; meaning, we create the most difficult scenario for a defacement detection system: detecting a defacer and his/her defacement style although we have never seen defacements from him/her before.

In the same way as for the traditional split, we employ 10-fold cross-validation. However, we do so slightly differently: first, we separate the reporters of the defacements into 10 bins uniformly at random (each bin containing 7,602 reporters). Second, we construct the corresponding defacement bins, i.e., we construct a defacement bin for each reporter bin so that it contains only the defacements reported by these reporters. For each bin, we then train MEERKAT on the remaining 9 bins and use the 10^{th} bin for testing. Note that the defacement bins contain a different number of samples, simply because the number of reported defacements varies per reporter (see Appendix A). We account for the uneven distribution of defacements by reporting the average true positive and false positive rate weighted by the number of samples.

Overall, when simulating the emergence of a new defacer, MEERKAT achieves a true positive rate of 97.882% and a false positive rate of 1.528% if bins are weighted, and 97.933% and 1.546% if they are not (see Figure 5; the true positive rate is between 97.061% and 98.465%, the false positive rate is between 0.661% and 2.564%). The Bayesian detection rates for the reporter split are 99.567% (unweighted) and 99.571% (weighted) respectively (per split, the Bayesian detection rate is between 99.286% and 99.814%).

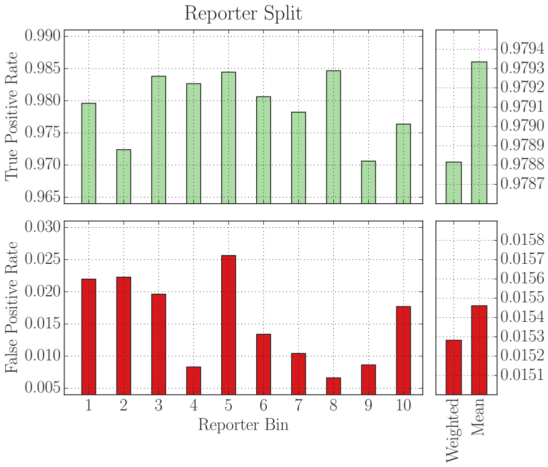


Figure 5: True positive and false positive rates for the reporter split, per bin of the 10-fold cross-validation set. Note that the scales for true positives and false negatives are the same, but that the *y*-axis goes from 0.965 to 0.99 for the true positive rate and 0.005 to 0.03 for the false positive rate. The weighted mean true positive rate is 97.882% and its false positive rate is 1.528% (weighted by samples per bin). The unweighted mean true positive rate is 97.933% and its false positive rate is 1.546%.

4.5 Time-wise Split

The time-wise experiment evaluates how well MEERKAT detects website defacements in the wild, i.e., in a real-world deployment. Here, we train the system on defacements seen in the past, and we detect defacements in the present. Similarly to the reporter split, the time-wise experiment shows that MEERKAT does not overfit on past defacements, and that it successfully detects present defacements.

Our training set selection follows a simple argument: it is extremely unlikely that websites today will be defaced in the same way as they were defaced in 2005 or even 1998. Including those defacements in our training set would then very likely decrease classification performance for defacement detection in 2014. Equivalently, one would not include this data to train the system in practice.

We train MEERKAT on all defacements that were reported between December 2012 and December 2013 (including, i.e., 13 months with 1,778,660 defacements observed in total), and 1,762,966 legitimate websites that we sample from all legitimate websites uniformly at random. We then detect defacements over a five months time frame, from January to May 2014, and we report the classification performance for each month. The test data from January to May 2014 spans a total of 1,538,878 unique samples that are distributed as follows: 421,758 samples from January 2014, 364,168 samples from February 2014, 474,758 samples from March 2014, 241,926 samples from April 2014, and 81,268 samples from the beginning of May 2014.

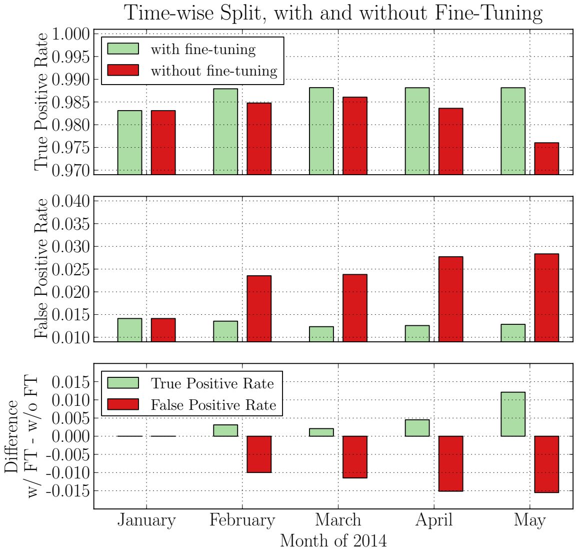


Figure 6: True positive and false positive rates, and the difference with and without fine-tuning, for the time-wise split. Note that the scales for true positives and false negatives are the same, but that the *y*-axis goes from 0.97 to 1 for the true positive rate and 0.01 to 0.04 for the false positive rate. No significant change is visible for the true positive rate in the beginning regardless if the network is fine-tuned regularly or not, however, a non-negligible difference is observable for May 2014. A difference is observable for the false positive rate starting in February 2014, after the network was first fine-tuned.

In detail, MEERKAT achieves a true positive rate between 98.310% and 98.816% when the system is fine-tuned after each month on the data observed in that month, and 97.603% to 98.606% when it is not. Although there is no significant difference in its accuracy from January to March when the neural network is fine-tuned and when it is not (see Figure 6), a non-negligible difference between their accuracy can be observed for April and the beginning of May (increase in 0.452 percentage points (pp) and 1.211 pp for the true positive rate; decrease of 1.513 pp and 1.550 pp for the false positive rate). The Bayesian detection rate if no fine-tuning is used decreases from 98.583% in January 2014 to 97.666% in February (0.917 pp decrease) to 97.177% in May (1.406 pp decrease to January). If fine-tuning is utilized, the Bayesian detection rate increases from 98.583% in January 2014 to 98.717% in May (0.134 pp).

Unsurprisingly, the regularly fine-tuned system performs better over time, probably because some defacers became significantly more active in 2014, like *Team System Dz*, who started to deface websites just in January 2014 and who were not active before at all, and because some defacers changed their defacements to spread a different message as opposed to the one they spread the year before. When the system is not fine-tuned, however, these minor changes to the defacements allow attackers to evade detection without actively trying to evade it, with a minor accuracy deterioration already visible after just four to five months, confirming that detection systems need to be able to tackle even minor concept drift adequately and gracefully to maintain accurate detection capabilities over time, like MEERKAT does with fine-tuning.

5 Limitations

Similar to other systems leveraging machine learning, our system has some limitations that can be used to evade detection. We discuss some of these limitations and show how they can be addressed for a real-world deployment. First, we discuss *concept drift*, a problem all systems leveraging machine learning have to deal with; second, we remark on *browser fingerprinting* and *delayed defacement*, an issue all client-based detection approaches have to address; and, lastly, we introduce the concept of *tiny defacements*, a limitation specific to defacement detection systems.

5.1 Concept Drift

Concept drift is the problem of predictive analysis approaches, such as detection systems, that the statistical properties of the input used to train the models change. In turn, a direct result of concept drift is often a heavy deterioration of the classification performance, up to the point where the system cannot differentiate between good and bad behavior anymore. For instance, prior work [48–55] has shown that concept drift (in the sense of adversarial learning) can actually be leveraged to evade detection systems and classifiers in practice. Therefore, a detection system must address it.

While concept drift is a major issue for all systems using machine learning, it can generally be addressed, due to its nature, by adopting a new feature space or retraining the machine learning model on new data, or with an increased weight on new data. However, often, old instances do not follow the statistical properties of the new feature models, and, therefore, they are classified less accurately than before. This has little impact in practice, because old instances are less likely to occur in the future anyways; yet, it is important to realize that this approach allows attackers to evade the system by oscillating their attack strategy regularly.

For MEERKAT, those shortcomings can be addressed more easily than for traditional systems: for minor concept drift, the system's accuracy can be maintained by fine-tuning the parameters of the network. Here, the system simply needs to learn minor adjustments to the weights of existing features from new data, because some features have become more important and others have become less important (they differ now more from other features than they did previously, relatively speaking; since we start with already-initialized weights, fine-tuning requires much less time than training the whole system again). Here, the features still model the differences between defacements and legitimate websites, however, the weights are not optimal anymore and need to be adjusted. Once the new weights are learned, classification performance is restored. Therefore, to address minor concept drift adequately, we recommend fine-tuning the model regularly, e.g., every month (see Section 4.5).

While fine-tuning the system's parameters can theoretically address major concept drift similar to retraining the system on new data, in practice, we expect prediction accuracy to decrease, since different or more features must be modeled with the same amount of resources. Instead, for major concept drift, increasing the number of hidden nodes of the neural network that learn the compressed representation (the features) and their weights, and then retraining the system

Split	True Positive Rate	False Positive Rate	Bayesian Detection Rate
Traditional	97.878%	1.012%	99.716%
Reporter (weighted)	97.882%	1.528%	99.571%
Reporter (unweighted)	97.933%	1.546%	99.567%
Time-wise with fine-tuning	98.310% - 98.816%	1.233% - 1.413%	98.583% - 98.767%
Time-wise without fine-tuning	97.603% - 98.606%	1.413% - 2.835%	97.177% - 98.583%

Table 2: Average true positive, false positive, and Bayesian detection rates for traditional and reporter split. Lower and upper bound of true positive, false positive, and Bayesian detection rate for time-wise split from January to May 2014.

can maintain the system's accuracy. Simply adding nodes to the hidden layers of the neural network can counteract the issue of major concept drift because we increase the number of features that MEERKAT learns from data directly. Therefore, introducing more hidden units allows the system to learn additional and different internal representations about the *look and feel* of defacements, while, at the same time, maintaining a model of how the old defacements look like. However, it requires computationally-costly retraining of the network (previously, having those additional hidden units in the network would result in overfitting because the system would learn more complex representations than necessary, and each would only differ little from one another; the system would then be prone to missing minor variations of defacements).

It is important to note that in both cases, for minor and major concept drift, MEERKAT requires no additional feature engineering because the features are learned *automatically* from data. In turn, this allows MEERKAT to handle any form of concept drift much more gracefully than approaches introduced by prior approaches, which require manual feature engineering.

5.2 Fingerprinting and Delayed Defacement

A second limitation of detection systems is fingerprinting. Since we are leveraging a web browser to collect the data that we are analyzing, in our case fingerprinting corresponds to IP-based and browser fingerprinting. For IP-based fingerprinting, a set of VPNs and proxies can be used to cloak and regularly change the browser's IP address. In case of browser fingerprinting, the server or some client-side JavaScript code detects what browser is rendering the website, and then displays the website differently for different browsers. In its current form, the screenshot engine from MEERKAT might be detectable (to some degree) by browser fingerprinting. It is theoretically possible to detect it because it is currently built on the headless browser PhantomJS rather than a "headful" browser typically used by a normal user, like Google Chrome. However, since PhantomJS is built from the same components as Google Chrome, fingerprinting the current screenshot engine is not trivial and requires intimate knowledge of the differences between the different versions of the components and their interaction. Therefore, we argue that the evasion through browser fingerprinting is unlikely. If, however, the screenshot engine is evaded this way in the future, only some minor engineering effort is required to utilize a browser extension to retrieve the websites' screenshots instead.[9]

Additionally, the issue of delaying the defacement emerges, also referred to as the snapshot problem [30]. With the increased popularity and use of JavaScript, client-side rendering, and asynchronous requests to backends by websites to provide a seamless and "reload-free" user experience, it is uncertain at what point in time a website is representative of how a user would experience it. This then bears the issue of when a detection system can take a representative snapshot of the website and stop executing client-side scripts. For instance, if a detection system takes a snapshot always after five seconds, to evade detection, defacers could simply inject JavaScript that only defaces the website if a user interacts with it for at least six seconds.

While delayed defacements are currently scarce, it is likely that they will gain some traction once more detection systems have been put in place, in a way similar to mimicry attacks and the evasions of malware detection systems [56, 57]. However, prior work can be leveraged to detect evasions [58] or trigger the functionality [59] to force the defacement to be shown. Both approaches are complementary to MEERKAT and we leave their adoption to defacement detection for future work, once delayed defacements are actually occurring in the wild.

5.3 Tiny Defacements

A third limitation of all current defacement detection systems, including MEERKAT, is the lack of detection capabilities for tiny defacements. Tiny defacements describe a class of defacements in which only a very minor modification is made to part of the content of the defaced website. For instance, a defacer might be dissatisfied by an article published by a newspaper. Instead of defacing the website as a whole, the attacker modifies (or deletes) the news article. It is clear that such defacements are very hard to differentiate from the original content because they might only have minor semantic changes to text or images. Thus, to detect tiny defacements, the detection system must understand the semantics of the website's content, its language, and its general behavior to derive a meaningful model for the website.

However, while those defacements exist, they are extremely scarce in numbers, or they are rarely noticed. In fact, it is seldom the case that a defacer wants to modify a website without embarrassing the operator more publicly. Most often, the goal of the defacer is to expose the insecurity

[9] In fact, we are migrating our screenshot engine to Chrome, eliminating the problem that PhantomJS might be fingerprinted.

of the website, ridicule the operator publicly, show their own "superiority," and place their opinion and beliefs in the most public space possible. Therefore, tiny defacements are currently of little interest to the defacers themselves, and, hence, also of little impact for detection systems. However, we acknowledge that tiny defacements must be addressed once they increase in numbers, possibly leveraging recent work to extract relevant changes from websites [60], and advances in natural language processing.

6 Related Work

Hereinafter, we discuss related work that detects defacement, and image-based detection used in computer security.

6.1 Defacement Detection

Similar to MEERKAT, Davanzo et al. [46] introduce a system that acts a monitoring service for website defacements. Their system utilizes the website's HTML source code for detection, and its features were selected manually based on domain knowledge acquired *a priori*, making the system prone to concept drift. On their, comparatively, very small dataset containing only 300 legitimate websites and 320 defacements, they achieve false positive rates ranging from 3.56% to 100% (depending on the machine learning algorithm used; suggesting extreme under- and over-fitting with some algorithms), and true positive rates between 70.07% to 100% (in the case of simply classifying everything as a defacement; i.e., extremely under-fitting the dataset). Overall, these results are significantly worse than MEERKAT, both in terms of false positives (1.012%) and true positives (97.878%).

Bartoli et al. [47] propose Goldrake, a website defacement monitoring tool that is very similar to the tool proposed by Davanzo et al. and leverages a superset of their features. To learn an accurate model, Goldrake requires knowledge about the monitored website to learn website-specific parameters. However, it is unclear how well Goldrake detects defacements in practice because it is evaluated on a small and (likely) non-diverse dataset comprised of only 11 legitimate websites and 20 defacements, on which it performs poorly with a high false negative rate (27%).

Medvet et al. [61] introduce a defacement detection system based on work by Bartoli et al. and Davanzo et al., but the detection engine is replaced by a set of functions that are learned through genetic programming. The learned functions take the features by Bartoli et al. and Davanzo et al. as input, but classification is more accurate on a dataset comprised of 15 websites (between 0.71% and 23.38% false positives, and about 97.52% true positives). It is, again, unclear how the system would fare in a real-world deployment because of the small and (likely) non-diverse dataset.

Note that all text-based approaches have major shortcomings, similar as those encountered in spam and phishing detection, such as using images to show text to evade detection. MEERKAT does not suffer from these shortcomings.

Lastly, most commercial products that detect website defacements are built upon host-based intrusion detection systems to monitor modifications of the files on the web server, e.g. via file integrity checks (checksums) [62, 63]. Therefore, those approaches bear the major shortcoming that they can only detect the subset of defacements that modify files on disk, and that they cannot detect other defacement attacks, such as through SQL injections; even when the defacements look exactly the same to the website's visitors. MEERKAT does not suffer from this shortcoming.

6.2 Image-based Detection in Security

Since, to the best of our knowledge, no prior work applies image-based methods to detect defacements, we compare prior work to defacement detection that visually detects phishing pages, or leverages image-based techniques as part of a larger system.

Medvet et al. [64] propose a system to detect if a potential phishing page is similar to a legitimate website. The system leverages features such as parts of the visible text, the images embedded in the website, and the overall appearance of the website as rendered by the browser for detection. Similarity is measured by comparing the 2-dimensional Haar wavelet transformations of the screenshots. Their system achieves a 92.6% true positive rate and a 0% false positive rate on a dataset comprised of 41 real-world phishing pages.

Similarly, Liu et al. [65] present an antiphishing solution that is deployed at an email server and detects linked phishing pages by assessing the visual similarity to the legitimate page, but only when analysis is triggered on keyword detection. The system detects phishing pages by comparing the suspicious website to the legitimate website by measuring similarity between text and image properties, like the font size and family used, or source of an image.

While detecting phishing pages by comparing the similarity of two websites makes sense, for defacements the difference between them is more interesting. Instead of creating a visually-similar page to trick users into submitting their credentials, a defacer wants to promote his message. Adopting existing phishing detection systems to detect defacements instead, i.e., by comparing if the website looks different from its usual representation, however, bears two problems: (a) the usual representation must be known and/or stored for comparison, and (b) false positives are much more likely if the website is dynamic or if it shows regularly-changing ads.

Anderson et al. [29] introduce image shingling, a technique similar to w-shingling, to cluster screenshots of scams into campaigns. However, in its current form, image shingling cannot be used to detect defacements as it is trivial to evade the clustering step with only minor modifications that are invisible to the human eye, and, thus, the technique is unsuitable for a detection system in an adversarial context.[10]

Nappa et al. [66] leverage perceptual hashing to group visually similar icons of malicious executables under the assumption that a similar icon suggests that the two executables are part of the same malware distribution campaign. While it is theoretically possible to detect defacements through perceptual hashing-based techniques and comparing the distance of the hashes, it is impractical to do so on a large scale and in an adversarial context. For

[10] The authors acknowledge the shortcomings in an adversarial context in Section 4.2, but they do not discuss any remediation techniques.

once, one must have a ground-truth screenshot that is close enough to the screenshot that one wants to classify; if ground-truth is not available or slightly too different, a system based on perceptual hashing will be unable to detect the defacement. Furthermore, classification is not constant in the number of defacements the system has seen in the past: for each new screenshot we would want to classify, we would need to compute the distance to the hashes of at least some (or all) of the previously-seen defacements.[11]

Grier et al. [67] introduce their own image similarity measure to cluster malicious executables that have similar looking user-interface components after being executed in a dynamic analysis environment. Two images are considered similar if the root mean squared deviation between the images' histograms is below some manually-determined threshold. Clearly, a defacement system based on this technique is not suitable in an adversarial context: an attacker can (and eventually will) simply change the colors slightly or add dynamic content, so that the root mean squared deviation is above the threshold, but remains visually the same to the human eye. Furthermore, exactly as for Nappa et al. [66], one needs to pair-wise compare the histogram of the screenshot one wants to classify to some or all of the already-seen defacements.[11]

MEERKAT does not suffer from these shortcomings: first, it learns high-level features on the defacements' general *look and feel* to detect also previously unseen defacements, and, second, its classification time is constant in the number of already-seen defacements.

7 Conclusions

In this paper, we introduced MEERKAT, a monitoring system to detect website defacements, which utilizes a novel approach based on the *look and feel* of a website to identify if the website has been defaced. To accurately identify website defacements, MEERKAT leverages recent advances in machine learning, like stacked autoencoders and deep neural networks, and combines them with computer vision techniques. Different from prior work, MEERKAT does not rely on additional information supplied by the website's operator, or on manually-engineered features based on domain knowledge acquired *a priori*, such as how defacements look. Instead, MEERKAT automatically learns high-level features from data directly. By deciding if a website has been defaced based on a region of the screenshot of the website instead of the whole screenshot, the system is robust to the normal evolution of websites and defacements and can be used at scale. Additionally, to prevent the evasion of the system through changes to the *look and feel* of defacements and to be robust against defacement variants, MEERKAT employs various techniques, such as dropout and fine-tuning.

We showed the practicality of MEERKAT on the largest website defacement dataset to date, spanning 10,053,772 defacements observed between January 1998 and May 2014, and 2,554,905 legitimate websites. On this dataset, in different scenarios, the system accurately detects defacements with a true positive rate between 97.422% and 98.816%, a false positive rate between 0.547% and 1.528%, and a Bayesian detection rate between 98.583% and 99.845%, thus significantly outperforming existing state-of-the-art approaches.

8 Acknowledgments

We want to express our gratitude toward the reviewers for their helpful feedback, valuable comments and suggestions to improve the quality of the paper.

This work was supported by the Office of Naval Research (ONR) under grant N00014-12-1-0165, the Army Research Office (ARO) under grant W911NF-09-1-0553, the Department of Homeland Security (DHS) under grant 2009-ST-061-CI0001, the National Science Foundation (NSF) under grant CNS-1408632, Lastline Inc., and SBA Research.

References

[1] G. Davanzo, E. Medvet, and A. Bartoli, "A Comparative Study of Anomaly Detection Techniques in Web Site Defacement Detection", in *Proceedings of the IFIP 20th World Computer Congress*, Springer, 2008.

[2] Anonymous, *Reference blinded for double-blind review process*, Nov. 2014. [Online]. Available: `http://anonymized`.

[3] Wall Street Journal (WSJ), *Malaysia Airlines Website Hacked by Group Calling Itself 'Cyber Caliphate'*, Jan. 26, 2015. [Online]. Available: `http://goo.gl/RhO2tO`.

[4] British Broadcasting Company (BBC), *Keighley Cougars website hacked to read 'I love you Isis'*, Nov. 2014. [Online]. Available: `http://goo.gl/bzxJ8M`.

[5] R. Preatoni, M. Almeida, K. Fernandez, and other unknown authors, *Zone-H.org - Unrestricted Information*, since January 1998. [Online]. Available: `http://zone-h.org/`.

[6] E. Kovacs, *Softpedia Interview: Alberto Redi, Head of Zone-H*, Jun. 8, 2013. [Online]. Available: `http://goo.gl/cwPBrW`.

[7] Malaysian Computer Emergency Response Team, *MyCERT Incident Statistics*, Jan. 2014. [Online]. Available: `http://goo.gl/0LTRPj`.

[8] CyberSecurity Malaysia, "MyCERT 2nd Quarter 2013 Summary Report", *eSecurity Bulletin*, vol. 34, Aug. 2013.

[9] S. Mansfield-Devine, "Hacktivism: assessing the damage", *Network Security*, vol. 2011, no. 8, 2011.

[10] M. Gorge, "Cyberterrorism: hype or reality?", *Computer Fraud & Security*, vol. 2007, no. 2, 2007.

[11] H. Kircher, "The Practice of War: Production, Reproduction and Communication of Armed Violence", in. Berghahn Books, Mar. 2011, ch. 12. Martyrs, Victims, Friends and Foes: Internet Representations by Palestinian Islamists.

[12] G. Weimann, "Terror on the Internet: The New Arena, the New Challenges", in. US Institute of Peace Press, 2006, ch. 6. Fighting Back: Responses to Terrorism on the Internet, and Their Cost.

[13] Wall Street Journal (WSJ), *Google Access Is Disrupted in Vietnam*, Feb. 23, 2015. [Online]. Available: `http://goo.gl/JlVtfW`.

[14] L. Makani, *100+ Zambian websites hacked & defaced: Spar, Postdotnet, SEC, Home Affairs, Ministry of Finance*, Apr. 2014. [Online]. Available: `http://goo.gl/NvQsJM`.

[15] British Broadcasting Company (BBC), *Angry Birds website hacked after NSA-GCHQ leaks*, Jan. 2014. [Online]. Available: `http://goo.gl/kHDIAj`.

[11] Detection time increases with each observed defacement; it is at best in $O(\log n)$ and at worst in $O(n)$, with n being all observed defacements.

[16] A. Mittal, *NIC of Suriname, Antigua & Barbuda and Saint Lucia Hacked by Pakistani Hackers*, Oct. 2013. [Online]. Available: http://goo.gl/ynGG0y.

[17] J. Leyden, *Islamist hackers attack Danish sites*, Feb. 2006. [Online]. Available: http://goo.gl/jcE7iv.

[18] ——, *Hacktivists attack UN.org*, Aug. 2007. [Online]. Available: http://goo.gl/SfvkUc.

[19] G. Maone, *United Nations vs. SQL Injections,* Aug. 2007. [Online]. Available: http://goo.gl/v8oXih.

[20] S. Reid, *Hip-Hop Sites Hacked By Apparent Hate Group; SOHH, AllHipHop Temporarily Suspend Access*, Jun. 2008. [Online]. Available: http://goo.gl/VtW4i6.

[21] B. Acohido, *State Department webpages defaced*, Oct. 23, 2013. [Online]. Available: http://goo.gl/698XRW.

[22] J. Leyden, *Foxconn website defaced after iPhone assembly plant suicides*, May 2010. [Online]. Available: http://goo.gl/6BtZbX.

[23] ——, *Anti-Israel hackers deface central bank site*, Apr. 2008. [Online]. Available: http://goo.gl/7Ve2xT.

[24] British Broadcasting Company (BBC), *Nottinghamshire Police website hacked by AnonGhost*, Nov. 2014. [Online]. Available: http://goo.gl/Gbldxt.

[25] ——, *Shropshire Fire Service website hacked by AnonGhost*, Nov. 2014. [Online]. Available: http://goo.gl/3dq4Cq.

[26] D. Dagon, M. Antonakakis, P. Vixie, T. Jinmei, and W. Lee, "Increased DNS Forgery Resistance Through 0x20-Bit Encoding: SecURItY viA LeET QueRieS", in *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, ACM, 2008.

[27] G. Vigna and C. Kruegel, "Host-based Intrusion Detection", *Handbook of Information Security. John Wiley and Sons*, 2005.

[28] A. Bartoli, G. Davanzo, and E. Medvet, "The Reaction Time to Web Site Defacements", *Internet Computing, IEEE*, vol. 13, no. 4, 2009.

[29] D. S. Anderson, C. Fleizach, S. Savage, and G. M. Voelker, "Spamscatter: Characterizing Internet Scam Hosting Infrastructure", in *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, ser. SS'07, USENIX Association, 2007.

[30] K. Borgolte, C. Kruegel, and G. Vigna, "Delta: Automatic Identification of Unknown Web-based Infection Campaigns", in *Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, ACM, 2013.

[31] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks", in *Proceedings of the 2nd International Conference on Learning Representations (ICLR)*, CBLS, Apr. 2014.

[32] Q. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G. Corrado, J. Dean, and A. Ng, "Building High-level Features Using Large Scale Unsupervised Learning", in *Proceedings of the 29th International Conference on Machine Learning (ICML)*, IMLS, Jun. 2012.

[33] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks.", in *Advances in Neural Information Processing Systems 25 (NIPS)*, vol. 1, 2012.

[34] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation", *arXiv preprint arXiv:1311.2524*, 2013.

[35] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, vol. 86, no. 11, 1998.

[36] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors", in *Proceedings of the 26th International Conference on Machine Learning (ICML)*, 2009.

[37] Q. V. Le, J. Ngiam, Z. Chen, D. J. hao Chia, P. W. Koh, A. Y. Ng, and D. Chia, "Tiled convolutional neural networks.", in *Advances in Neural Information Processing Systems 23 (NIPS)*, 2010.

[38] H. Lee, R. Grosse, R. Ranganath, and A. Y. Ng, "Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations", in *Proceedings of the 26th Annual International Conference on Machine Learning (ICML)*, ACM, 2009.

[39] P. Sermanet, S. Chintala, and Y. LeCun, "Convolutional neural networks applied to house numbers digit classification", in *Proceedings of the 21st International Conference on Pattern Recognition (ICPR)*, IEEE, 2012.

[40] A. Hyvärinen, J. Hurri, and P. O. Hoyer, *Natural Image Statistics: A Probabilistic Approach to Early Computational Vision.* Springer, 2009, vol. 39.

[41] K. Gregor and Y. LeCun, "Emergence of complex-like cells in a temporal product network with local receptive fields", *arXiv preprint arXiv:1006.0448*, 2010.

[42] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun, "What is the best multi-stage architecture for object recognition?", in *Proceedings of the 12th IEEE International Conference on Computer Vision*, IEEE, 2009.

[43] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors", *arXiv preprint arXiv:1207.0580*, 2012.

[44] Y. Jia, *Caffe: An Open Source Convolutional Architecture for Fast Feature Embedding*, 2013. [Online]. Available: http://goo.gl/Fo9YO8.

[45] S. Axelsson, "The Base-Rate Fallacy and the Difficulty of Intrusion Detection", *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, no. 3, 2000.

[46] G. Davanzo, E. Medvet, and A. Bartoli, "Anomaly Detection Techniques for a Web Defacement Monitoring Service", *Expert Systems with Applications*, vol. 38, no. 10, 2011.

[47] A. Bartoli and E. Medvet, "Automatic Integrity Checks for Remote Web Resources", *Internet Computing, IEEE*, vol. 10, no. 6, 2006.

[48] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. Tygar, "Adversarial Machine Learning", in *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence (AISEC)*, ACM, Oct. 2011.

[49] M. Barreno, B. Nelson, A. D. Joseph, and J. Tygar, "The Security of Machine Learning", *Machine Learning*, vol. 81, no. 2, 2010.

[50] M. Barreno, B. Nelson, R. Sears, A. D. Joseph, and J. D. Tygar, "Can machine learning be secure?", in *Proceedings of the 13th ACM Symposium on Information, Computer and Communications Security (CCS)*, ACM, Oct. 2006.

[51] N. Šrndic and P. Laskov, "Practical Evasion of a Learning-Based Classifier: A Case Study", in *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, IEEE, May 2014.

[52] D. Lowd and C. Meek, "Adversarial Learning", in *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining (KDD)*, ACM, Aug. 2005.

[53] N. Dalvi, P. Domingos, Mausam, S. Sanghai, and D. Verma, "Adversarial Classification", in *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, ACM, 2004.

[54] A. Globerson and S. Roweis, “Nightmare at Test Time: Robust Learning by Feature Deletion”, in *Proceedings of the 23rd International Conference on Machine Learning (ICML)*, ACM, 2006.

[55] H. Xiao, H. Xiao, and C. Eckert, “Adversarial label flips attack on support vector machines”, in *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI)*, Aug. 2012.

[56] D. Wagner and P. Soto, “Mimicry Attacks on Host-based Intrusion Detection Systems”, in *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, ACM, 2002.

[57] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, “Automating Mimicry Attacks Using Static Binary Analysis”, in *Proceedings of the 14th Conference on USENIX Security Symposium*, USENIX Association, 2005.

[58] A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegel, and G. Vigna, “Revolver: An Automated Approach to the Detection of Evasive Web-based Malware”, in *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[59] C. Kolbitsch, E. Kirda, and C. Kruegel, “The Power of Procrastination: Detection and Mitigation of Execution-stalling Malicious Code”, in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, ACM, 2011.

[60] K. Borgolte, C. Kruegel, and G. Vigna, “Relevant Change Detection: Framework for the Precise Extraction of Modified and Novel Web-based Content as a Filtering Technique for Analysis Engines”, in *Proceedings of the Companion Publication of the 23rd International World Wide Web Conference (WWW)*, IW3C2, 2014.

[61] E. Medvet, C. Fillon, and A. Bartoli, “Detection of Web Defacements by Means of Genetic Programming”, in *Proceedings of the 3rd International Symposium on Information Assurance and Security*, IEEE Computer Society, 2007.

[62] G. H. Kim and E. H. Spafford, “The Design and Implementation of Tripwire: A File System Integrity Checker”, in *Proceedings of the 2nd ACM Conference on Computer and Communications Security (CCS)*, ACM, 1994.

[63] A. G. Pennington, J. D. Strunk, J. L. Griffin, C. A. N. Soules, G. R. Goodson, and G. R. Ganger, “Storage-based Intrusion Detection: Watching Storage Activity for Suspicious Behavior”, in *Proceedings of the 12th Conference on USENIX Security Symposium*, USENIX Association, 2003.

[64] E. Medvet, E. Kirda, and C. Kruegel, “Visual-similarity-based Phishing Detection”, in *Proceedings of the 4th International Conference on Security and Privacy in Communication Networks (SecureComm)*, ACM, 2008.

[65] W. Liu, X. Deng, G. Huang, and A. Y. Fu, “An Antiphishing Strategy Based on Visual Similarity Assessment”, *Internet Computing, IEEE*, vol. 10, no. 2, 2006.

[66] A. Nappa, M. Rafique, and J. Caballero, “Driving in the Cloud: An Analysis of Drive-by Download Operations and Abuse Reporting”, English, in *Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. Lecture Notes in Computer Science, K. Rieck, P. Stewin, and J.-P. Seifert, Eds., vol. 7967, Springer Berlin Heidelberg, 2013. [Online]. Available: `http://goo.gl/Z2IJ4D`.

[67] C. Grier, L. Ballard, J. Caballero, N. Chachra, C. J. Dietrich, K. Levchenko, P. Mavrommatis, D. McCoy, A. Nappa, A. Pitsillidis, N. Provos, M. Z. Rafique, M. A. Rajab, C. Rossow, K. Thomas, V. Paxson, S. Savage, and G. M. Voelker, “Manufacturing Compromise: The Emergence of Exploit-as-a-Service”, in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS ’12, ACM, 2012. [Online]. Available: `http://goo.gl/M1DOdZ`.

Appendix

A Reporter Cross-validation Split

In our reporter split experiment (Section 4.4), we split the dataset by reporter to simulate that a new defacer group emerges. Each cross-validation bin contains the same amount of reporters, but because they reported different numbers of defacements, bins do not contain the same amount of samples. We account for the size difference in our experiments by weighting each bin. Table 3 lists the number of samples per bin.

Bin	Defacements	Legitimate Websites
1	1,116,808	308,202
2	992,232	273,823
3	712,270	196,563
4	907,306	250,387
5	696,069	192,092
6	734,208	202,617
7	1,276,764	352,345
8	789,895	217,985
9	979,309	270,257
10	1,053,147	290,634
Total	9,258,008	2,554,905

Table 3: Number of samples per cross-validation bins used for the reporter split. Note that the total number of defacements in the reporter split contains 168 defacements less than available in the whole dataset because otherwise reporters would be distributed unevenly per bin. However, due to the considerable size of the dataset, omitting these defacements has negligible impact.

B Image-based Object Recognition

Much prior work has been carried out in computer vision to classify images and recognize objects in images. Most recently, object recognition underwent a “new spring” with the rise of deep learning. Deep learning gained traction because training them on large datasets became computationally feasible, and they consistently outperformed other algorithms. We discuss our two main inspirations.

Le et al. [32] introduce a feature learning approach that leverages unsupervised learning with a deep networks comprised of stacked sparse autoencoders utilizing pooling and local contrast normalization. The main idea is to learn high-level features from only unlabeled data (10 million pictures from random Youtube videos); high-level features such as if the image contains a cat, or a human face or body part. After training, the network improves relatively to prior state-of-the-art by 70% on the ImageNet dataset.

Krizhevsky et al. [33] employed supervised learning to train a deep convolutional neural network to classify 1.2 million images spanning 1,000 classes from a subset of the ImageNet dataset and they improve considerably on the state-of-the-art with a top-1 error rate of 37.5% (the classifier is correct for 62.5%) and a top-5 error of 17.0% (for 83% images, the correct class is among top 5 classes). To not overfit the dataset and to reduce the network’s training time, they use rectified linear units as the neurons’ output functions.

Recognizing Functions in Binaries with Neural Networks

Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi
University of California, Berkeley
{*ricshin, dawnsong, rezamoazzezi*}*@berkeley.edu*

Abstract

Binary analysis facilitates many important applications like malware detection and automatically fixing vulnerable software. In this paper, we propose to apply artificial neural networks to solve important yet difficult problems in binary analysis. Specifically, we tackle the problem of function identification, a crucial first step in many binary analysis techniques. Although neural networks have undergone a renaissance in the past few years, achieving breakthrough results in multiple application domains such as visual object recognition, language modeling, and speech recognition, no researchers have yet attempted to apply these techniques to problems in binary analysis. Using a dataset from prior work, we show that recurrent neural networks can identify functions in binaries with greater accuracy and efficiency than the state-of-the-art machine-learning-based method. We can train the model an order of magnitude faster and evaluate it on binaries hundreds of times faster. Furthermore, it halves the error rate on six out of eight benchmarks, and performs comparably on the remaining two.

1 Introduction

Binary analysis enables many useful applications in computer security, given the plethora of possible situations in which the original high-level source code is unavailable, has been lost, or is otherwise inconvenient to use. For example, detection of malware, hardening software against common vulnerabilities, and protocol reverse-engineering are most useful when the procedures involved can directly operate on binaries.

The central challenge of binary analysis is perhaps the lack of high-level semantic structure within binaries, as compilers discard it from the source code during the process of compilation. Malware authors often go a step further and obfuscate their output in an attempt to frustrate any possible analysis by researchers.

Functions are a seemingly basic yet fundamental piece of structure in all programs, but most binaries come as an undifferentiated sequence of machine-language instructions without any information about how parts group into functions. Therefore, the many binary analysis techniques which rely on function boundary information must first attempt to recover it through *function identification*. For instance, function identification can assist the addition of control-flow integrity enforcement to binaries, in restricting jumps appropriately. Similarly, decompilers and debuggers need to know the locations of functions to provide useful output to the user [2].

Several previous works have attempted the function identification task, ranging from simple heuristics to approaches using machine learning. The problem might seem simple at first glance, but Bao et al. showed with ByteWeight [2], a recently-proposed machine-learning-based approach, that the simpler techniques used by popular tools like IDA Pro and the CMU Binary Analysis Platform have relatively poor accuracy. By constructing signatures of function starts as weighted prefix trees, ByteWeight greatly improves on the accuracy of function identification results compared to past work. Nevertheless, it leaves much room for improvement, especially in terms of computational efficiency: the authors report that training on their dataset of 2,064 binaries required 587 compute-hours, whereas running the method on the dataset took on the order of several compute-days. Also, while ByteWeight achieves about 98% accuracy on some benchmarks, it performs at just 92-93% on some others.

In this paper, we propose a new approach to function identification leveraging artificial neural networks. First proposed in the 1940s, artificial neural networks arose as a simple approximation of interconnected biological neurons in the central nervous systems of animals, and have remained an active area of of research since then. However, in the past few years, neural networks have experienced a significant surge in popularity (often under the name "deep learning"), largely been driven by new empirical results. The vastly larger amounts of processing power and storage available today enabled researchers to train much larger networks containing many more stages of processing (hence the "deep" appellation) and parameters than before, making full use of the massive labeled datasets available today; these factors have led to repeated breakthroughs in benchmarks of the computer vision and speech recognition communities, among others.

We note some attractive features of neural networks. First, they can learn directly from the original representation with minimal preprocessing (or "feature engineering") needed. As an example, the preprocessing for images might discard information about the precise shading of objects; for binaries, Bao et al. disassembles the code into instructions and removes immediate operands from them. Second, neural networks can learn *end-to-end*, where each of its constituent stages are trained simultaneously in order to best solve the end goal. In contrast, other state-of-the-art approaches to tasks like machine translation or question answering use pipelines of discrete components trained separately at an unrelated task, such as parsers or part-of-speech taggers. Empirical evidence suggests that end-to-end learning enables each stage to directly learn the intermediate representations necessary to solve the task, with less need for preconceived notions (such as syntax trees) about what they should look like.

Given the success that neural networks have shown in other applications, we raise the question of whether they would also prove adept at problems in binary analysis, such as function identification. Our search turned up no other works which attempted using neural networks to solve problems in binary analysis. Nevertheless, our experimental results show that they can successfully solve the function identification task accurately and efficiently. If the experience in other fields can serve as a guide, they may also prove useful for more complicated tasks in program and binary analysis, especially for those which require complicated modeling or analysis difficult to specify by hand. Furthermore, advances with neural networks in other applications might prove directly adaptable and lead to "free" gains in performance; this work certainly relies on general advances within neural networks targeted at entirely different applications.

With our proposed solution, we train a recurrent neural network to take bytes of the binary as input, and predict, for each location, whether a function boundary is present at that location. We found that we did not need to perform any preprocessing, such as disassembly or normalization of immediates, in order to obtain good results. We evaluate our approach using the dataset provided by Bao et al. [2], enabling a direct comparison. We found that recurrent neural networks can learn much more efficiently than ByteWeight, which reported using 587 compute-hours; we can train on the same dataset in 80 compute-hours, while achieving similar or better accuracy. Testing the method on the dataset takes only about 43 minutes of computation, whereas Bao et al. [2] reported needing over 2 weeks.

In the rest of the paper, we first precisely define the problem at hand. We explain the necessary background in neural networks, and describe the particular architecture we chose to use for our method. We give the results of our empirical evaluation, describe some related works in the areas of function identification and neural networks, and then conclude with some discussion.

We make the following contributions in this paper:

- We find that neural networks are a viable approach towards solving some problems in binary analysis.
- In particular, we show that recurrent neural networks can solve the function identification problem more efficiently than the previous state-of-the-art, as shown by empirical evaluation on a dataset consisting of multiple operating systems, architectures, compilers, and compiler options.
- We describe the challenges we faced in correctly applying neural networks to this problem, and how to address them.

2 Problem Definition

We first define notation so that we can precisely define the function identification task that we address in this paper. We then provide a formal definition of function identification.

2.1 Notation

We concern ourselves with the machine code contained within a program binary or library. A typical executable contains many different sections containing various information in addition to the code: for example, dynamically-linked libraries to load, constant strings, and statically-allocated variables, all of which we ignore.

We treat the code C itself as a sequence of bytes $C[0], C[1], \cdots, C[l]$, where $C[i] \in \mathbb{Z}_{256}$ is the i^{th} byte in the sequence. We denote the n functions in the binary as $f_1, \cdots, f_n$. We label the indices of the bytes of code which belong to each function f_i (i.e., the bytes corresponding to instructions which might get executed while running that function) as $f_{i,1}, \cdots, f_{i,l_i}$, where l_i is the total number of bytes in f_i. Without loss of generality, we assume $f_{i,1} < f_{i,2} < \cdots < f_{i,k}$. Each byte may belong to any number of functions, and functions may contain any set of bytes, contiguous or not.

Note that we defined the code and functions as sets of bytes rather than instructions. In the x86 and x86-64 ISAs, a sequence of bytes can have many plausible instruction decodings depending on the offset at which decoding begins; therefore each byte might belong to a handful of possible instructions. Working in terms of bytes allows us to avoid this ambiguity.

2.2 Task definition

Let us assume that we have access the code C of a binary, but no information about the functions $f_1, \cdots, f_n$ within the code. We define the following tasks:

- Function start identification: Given C, find $\{f_{1,1}, \cdots, f_{n,1}\}$. In other words, recover the location of the first byte of each function.
- Function end identification: Given C, find $\{f_{1,l_1}, \cdots, f_{n,l_n}\}$. In other words, find the bytes where each of the n functions in the binary ends. The length of each function is not given.
- Function boundary identification: Given C, find $\{(f_{1,1}, f_{1,l_1}) \cdots, (f_{n,1}, f_{n,l_n})\}$. In other words, discover the location of the first and last byte within each function. This task is more than a simple combination of function start and end identification. If the starts and ends of functions have been identified separately, they need to be paired correctly so that each pair contains the start and end of the same function.
- General function identification: Given C, find $\{(f_{1,1}, f_{1,2}, \cdots, f_{1,l_1}) \cdots, (f_{n,1}, f_{n,2}, \cdots f_{n,l_n})\}$; i.e., determine the number of functions in the file, and all of the bytes which make up each function.

Function boundary identification is a superset of function start and end identification, whereas general function identification is a superset of all other tasks. In this paper, we attempt the first three problems, and leave the fourth to future work.

2.3 Metrics

To evaluate results from the model, we use the precision, recall, and F1 metrics. They have the following definitions:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{F1} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

where TP is the number of true positive predictions, FP is the number of false positive predictions, and FN is the number of false negative predictions. The F1 score is the harmonic mean of precision and recall and allows us to conveniently compare different results using one number.

Since most bytes within a program do not begin or end functions, these metrics can give a better picture of the effectiveness of the model than the simple accuracy metric. For example, predicting that there exists no functions in the code would give greater than 99.9% accuracy, since fewer than 0.1% of the bytes begin or end a function. The accuracy metric does not reveal that these predictions would be mostly useless. In contrast, the 0% recall these predictions would achieve makes it clear.

2.4 Examples

In Figure 1, we show an example of a short C function and its corresponding binary code after compilation at two different optimization levels.

The code in Figure 1b contains very clear markers of function start and end: the function prologue of `push %rbp` and `mov %rsp,%rbp` saves the caller's stack frame, and the function ends with `retq` which occurs nowhere else within the function. In contrast, Figure 1c does not use the stack at all, so the function begins with some accesses to the function arguments passed in `edi` and `esi`; looking for `push %rbp` would fail. Moreover, similar accesses to arguments occur again within the body of the function, making it difficult to solely rely on that as a marker of the function start. Likewise, `retq` occurs twice within the code, and so predicting a function end when we see this instruction would fail.

This example gives an instance of why function identification can pose much difficulty, with simple heuristics unlikely to suffice, contrary to what intuition might suggest.

3 Background

In this section, we describe what neural networks are and how they are trained. In particular, we focus on the various forms of recurrent neural networks, which are the class of model we use for our method.

3.1 Multi-layer perceptrons

A multi-layer perceptron (MLP), also referred to as a feedforward neural network, is a function $L : \mathbb{R}^s \to \mathbb{R}^t$ parameterized in a particular way. As its name implies, a multi-layer perceptron consists of multiple layers L_i,

```
int mul_inv(int a, int b) {
  int b0 = b, t, q;
  int x0 = 0, x1 = 1;
  if (b == 1) return 1;
  while (a > 1) {
    q = a / b;
    t = b, b = a % b, a = t;
    t = x0, x0 = x1 - q * x0, x1 = t;
  }
  if (x1 < 0) x1 += b0;
  return x1;
}
```

(a) A C function which computes the modular multiplicative inverse.

```
00000000004005a1 <mul_inv>:
  4005a1: push   %rbp
  4005a2: mov    %rsp,%rbp
  4005a5: mov    %edi,-0x24(%rbp)
  4005a8: mov    %esi,-0x28(%rbp)
  4005ab: mov    -0x28(%rbp),%eax
...
  400615: jns    40061d <mul_inv+0x7c>
  400617: mov    -0xc(%rbp),%eax
  40061a: add    %eax,-0x8(%rbp)
  40061d: mov    -0x8(%rbp),%eax
  400620: pop    %rbp
  400621: retq
```

(b) Compiled with `gcc -O0`.

```
0000000000400830 <mul_inv>:
  400830: cmp    $0x1,%esi
  400833: mov    %edi,%eax
  400835: je     400878 <mul_inv+0x48>
  400837: cmp    $0x1,%edi
  40083a: jle    400878 <mul_inv+0x48>
  40083c: mov    %esi,%ecx
  40083e: mov    $0x1,%r8d
  400844: xor    %edi,%edi
  400846: jmp    400855 <mul_inv+0x25>
  400848: nopl   0x0(%rax,%rax,1)
  40084f:
...
  400869: jg     400850 <mul_inv+0x20>
  40086b: add    %edi,%esi
  40086d: mov    %edi,%eax
  40086f: test   %edi,%edi
  400871: cmovs  %esi,%eax
  400874: retq
  400875: nopl   (%rax)
  400878: mov    $0x1,%eax
  40087d: retq
  40087e: xchg   %ax,%ax
```

(c) Compiled with `gcc -O3`. Function ends at `40087d`; `40087e` is padding between this function and the next one.

Figure 1: An example function in C (taken from `http://rosettacode.org/wiki/Modular_inverse#C`), and its corresponding machine code (with uninteresting parts omitted for brevity). The function was compiled using GCC 4.9.1 on Linux x86-64. The `-O3` version does not contain a conventional function prologue and epilogue which manipulates the stack or frame pointer.

each of which computes

$$
\begin{aligned}
L_i &: \mathbb{R}^{m_{i-1}} \to \mathbb{R}^{m_i} \\
G &: \mathbb{R} \to \mathbb{R} \\
L_i(x) &= G(W_i x + b_i) \\
W_i &\in \mathbb{R}^{m_i \times m_{i-1}} \\
b_i &\in \mathbb{R}^{m_i}
\end{aligned}
$$

then the entirety (consisting of k layers) is simply these layers composed together:

$$
\begin{aligned}
L(x) &= L_k(L_{k-1}(\cdots(L_1(x)))) \\
m_0 &= s \\
m_k &= t
\end{aligned}
$$

with the dimensions of the output of one layer matching the dimensions of the input of the subsequent layer. The m_i are the dimensionality of the input and the ultimate output of the network, as well as the intermediates produced by each of the layers.

The term “layer” is often used to refer to not the parameters of the functions L_i, but the inputs or outputs of these functions. In turn, each element of the inputs or outputs of the functions are often called “units”, or by analogy, “neurons”.

In this definition, G is referred to as an *activation function* or *nonlinearity*, and computed separately for each element. Without the activation function, L would simply be an affine function which we could write as $Wx + b$, which does not enable the expressivity that we need. Common nonlinearities are the logistic sigmoid function and the hyperbolic tangent function:

$$
\begin{aligned}
\sigma(x) &= \frac{1}{1 + e^{-x}} \\
\tanh(x) &= \frac{e^{2x} - 1}{e^{2x} + 1}
\end{aligned}
$$

which have the ranges of $(0,1)$ and $(-1,1)$, respectively.

Usually, the final layer will have no activation function because we do not wish to bound the output to a limited range, or a softmax function if we want to use the MLP as a multi-class classifier, so that we can interpret the values as a probability distribution. The softmax function is computed as follows:

$$
S(\mathbf{x})_i = \frac{e^{\mathbf{x}_i}}{\sum_{k=1}^{n} e^{\mathbf{x}_k}}
$$

Note that unlike the other activation functions, it does not operate elementwise. Due to the normalization term in the denominator, $S(\mathbf{x})$ sums to 1. A multi-layer perceptron consisting of one layer with a softmax activation function is equivalent to multi-class logistic regression.

3.2 Loss functions

Now that we have defined multi-layer perceptrons as a class of parameterized functions, we need a method to find appropriate parameters so that the neural network does what we want. First, we define a *loss function* in order to quantify how much differently the network behaves from our target. A common loss function is the squared Euclidean distance:

$$d(y, \hat{y}) = \|y - \hat{y}\|_2^2$$

where y is the "true" output and $\hat{y}$ is the one produced by the neural network.

In the multi-class classification case, if y is the correct class and $\pi(x)$ is the probability distribution produced by the neural network, then we can use the negative log probability:

$$d(y, \pi(x)) = -\log \pi(x)_y$$

Usually, we will have a list of correct input-output pairs $(x_1, y_1), \cdots, (x_n, y_n)$ for the purpose of training the network. Then we can seek to minimize the mean of the losses, or $\frac{1}{n} \sum_{i=1}^{n} d(y_i, f(x_i))$. We use this type of loss function throughout this paper.

3.3 Gradient descent and backpropagation

To minimize the loss, and therefore obtain a neural network which performs our desired function, we can consider various standard optimization methods. Specifically, we wish to minimize D defined as such:

$$\begin{aligned} \theta &= (W_1, b_1, \cdots, W_k, b_k) \\ D(\theta) &= \frac{1}{n} \sum_{i=1}^{n} d(y_i, f_\theta(x_i)) \\ \min_\theta \quad & D(\theta) \end{aligned}$$

A typical way to minimize differentiable functions is gradient descent, which works by repeated applications of the following update:

$$\theta' = \theta - \alpha \cdot \frac{\partial D(\theta)}{\partial \theta}$$

where α is generally a small number. Intuitively, the derivative allows us to analytically determine which direction we should move in within each dimension of θ to reduce the value of F. Subtracting a small multiple of the gradient performs this function.

If D is convex, this procedure is guaranteed to converge at the optimal value of θ given appropriate choices of α. Many machine learning models and classifiers involve optimizing a convex function in a similar way. Unfortunately, neural networks are generally non-convex in its parameters, allowing for a richer class of possible functions, but which means that these theoretical guarantees do not hold. Instead, the procedure may lead us to a local optimum or a saddle point where the derivative is zero.

We now need the derivative of D. Estimating the derivative numerically seems a simple and straightforward solution, but it is a highly inefficient one requiring as many evaluations of D as the dimensionality of θ. Instead, we can use a method called *backpropagation* to compute the derivative analytically. We describe the details of backpropagation in Section A.

3.4 Recurrent neural networks

While multi-layer perceptrons can approximate a wide variety of functions, they can only operate on inputs of fixed size and produce an output of fixed size. In principle, given a large input, we could divide it into fixed-size pieces and give them separately to a multi-layer perceptron. However, the output of each piece depends only on that input piece, and we cannot represent any dependencies between parts of the input in one piece and the output for a different piece.

Recurrent neural networks are one paradigm for addressing this conundrum, and map sequences to sequences (recursive neural networks, which have the same initialism, are an alternative developed for computing on trees).

We can formally define them in the following way:

$$\begin{aligned} L &: \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^n \\ G &: \mathbb{R} \to \mathbb{R} \\ L(x, h) &= G(W_{hx}x + W_{hh}h + b) \\ W_{hx} &\in \mathbb{R}^{n \times m} \\ W_{hh} &\in \mathbb{R}^{n \times n} \\ b &\in \mathbb{R}^n \end{aligned}$$

Given an input sequence $(x_1, \cdots, x_T)$ (where $x_i \in R^m$), we compute $(h_1, \cdots, h_T)$ like this:

$$\begin{aligned} h_0 &= 0 \\ h_1 &= L(x_1, h_0) \\ &\vdots \\ h_T &= L(x_T, h_{T-1}) \end{aligned}$$

Note that the operation on each elements uses the same weights. Nevertheless, the use of h enables the network to remember information from past elements to use while processing the current element, and propagate information into the future.

We can use the h_ts as inputs to another recurrent neural network, or apply to them a linear transformation possibly with a softmax activation function (as done in the final layer of a multi-layer perceptron):

$$y_i = S(W_{yh}h_t + b)$$

To define a loss function for a recurrent neural network, we can apply a loss function for a multi-layer perceptron separately to each input-output pair within the sequence and simply sum the losses together:

$$d(y,\hat{y}) = \sum_{i=1}^{T} d(y_t,\hat{y}_t)$$

The input and output sequences of a recurrent neural network need not have the same lengths. For instance, we might allow an arbitrary number of inputs but only one output to summarize the contents of the input in some way. In this case, we can simply adjust the loss function to only compute the loss at the relevant parts of the output sequence. We can also train a recurrent neural network to map an input sequence to an arbitrary number of output symbols, if we run the network to obtain some number of outputs until it produces a special 'stop' output.

As with the multi-layer perceptron, we would like to learn appropriate parameters so that the recurrent neural network parameterized with them computes a desired function, using gradient descent. We can compute the derivative of the RNN with respect to its parameters in the same way as earlier. In particular, we can unroll the RNN so that it becomes a long feedforward neural network which computes on a fixed-length sequence, and compute the gradient for this network using an appropriate loss function with backpropagation. After unrolling, note that the time-dependent layers should share the same weights This procedure is also called *backpropagation through time* [16].

3.5 Limitations of recurrent neural networks

In this section, we point out some limitations of recurrent neural networks which can limit their usefulness.

As specified in this paper, recurrent neural networks cannot compute for an arbitrary number of timesteps before computing the answer. For example, RNNs can easily compute the parity of an arbitrarily long stream of bits [15], as this requires a constant number of operations per input. In contrast, we can reason that a RNN could not multiply numbers of arbitrary size, as multiplication is a $O(n^2)$ operation on the length of the numbers [22].

Also, h has a fixed size which we cannot easily adapt if necessary in order to store more information. For example, previous works have shown success with using RNNs for machine translation, in which the RNN first reads a sentence in the source language and stores its meaning in h before producing the corresponding words in the target language using the information in h. While we can pick a size for h such that it has enough capacity to store information on a typical-length sentence, we can imagine that this scheme would break down for a sentence of sufficient length.

The most-studied limitation revolves around difficulties in training a recurrent neural network, due to what are referred to as the vanishing gradient and exploding gradient problems [16]. Consider that

$$\frac{\partial h_v}{\partial h_t} = \prod_{v \geq i > t} \frac{\partial h_i}{\partial h_{i-1}} = \prod_{v \geq i > t} W_{hh} G'(h_{i-1})$$

The repeated multiplication with W_{hh} ($v-t$ times) can cause $\frac{\partial h_v}{\partial h_t}$ to grow exponentially large ("explodes") or go to 0 ("vanishes") depending on whether the largest eigenvalue of W_{hh} is greater or smaller than 1. Therefore, an input will often have a very large or vanishingly small effect on an output which occurs far in the future, in terms of the gradient computation. For exploding gradients, a simple solution involves rescaling the gradient to a fixed norm if its magnitude is too large. On the other hand, dealing with vanishing gradients can prove more challenging.

3.6 Long Short-Term Memory and Gated Recurrent Units

To avoid the exploding and vanishing gradient problems with recurrent neural networks, previous work has proposed RNN architectures carefully designed to remove the long-range multiplicative characteristics of RNNs which lead to these problems.

Long Short-Term Memory (LSTM), one of these architectures, have enabled impressive empirical results in areas such as speech recognition, machine translation, and image captioning. Within this model, the state which propagates through time has no multiplicative updates at each step; instead, it is stored in a *memory cell* c_t which receives additive updates, combined with a mechanism for erasing irrelevant information from the previous time step. The "input modulation gate" (g) and the "forget gate" (f), respectively, control whether the memory cell receives the additive update or discards (some part of the) previous memory cell contents.

Following the notation in Zaremba et al. [23], we can formally define the LSTM:

$$\begin{aligned}
x_t, h_{t-1}, c_{t-1} &\rightarrow h_t, c_t \\
i &= \sigma(W_{xi}x_t + W_{hi}h_{t-1}) \\
f &= \sigma(W_{xf}x_t + W_{hf}h_{t-1}) \\
o &= \sigma(W_{xo}x_t + W_{ho}h_{t-1}) \\
g &= \tanh(W_{xg}x_t + W_{hg}h_{t-1}) \\
c_t &= f \odot c_{t-1} + i \odot g \\
h_t &= o \odot \tanh(c_t)
\end{aligned}$$

Here, $\odot$ represents element-wise multiplication.

In principle, these gates enable the gradient to propagate across long time scales, since the LSTM can ignore irrelevant inputs through the input modulation gate, remember information only until necessary using the forget gate, and output only relevant information using the output gate. When the forget gate is "open", i.e. close to 1, then the gradient will propagate mostly unchanged. The input and output at each time step only influences the gradient when the corresponding gates are open.

Gated Recurrent Units (GRU) have been proposed more recently as a simpler alternative to LSTMs, while sharing the same goals of avoiding the long-range dependency problems that have plagued RNNs. The main differences lie in that there exists no separate memory state c_t from the hidden state h_t, and the network exposes the entire hidden state at each time step. The forget gate interpolates between the previous hidden state and the new input i, with no separate input modulation gate. Instead, g modulates the amount of influence the previous hidden state has on i.

We define the GRU formally:

$$\begin{aligned} &x_t, h_{t-1} \to h_t \\ &\quad g = \tanh(W_{xg}x_t + W_{hg}h_{t-1}) \\ &\quad i = \tanh(W_{xi}x_t + W_{hi}(g \odot h_{t-1})) \\ &\quad f = \sigma(W_{xf}x_t + W_{hf}h_{t-1}) \\ &\quad h_t = f \odot h_{t-1} + (1-f) \odot i \end{aligned}$$

While the GRU theoretically lacks some of the flexibility provided by the LSTM, it is both simpler to implement and easier to compute, requiring about half as many calculations in each time step compared to the LSTM.

4 Methods

In this section, we describe how we built upon the background in Section 3 to perform the task of function identification.

4.1 Basic architecture

Our simplest architecture uses a recurrent neural network, described in Section 3.4, to process each byte and and output a decision for that byte as to whether it begins a function or not.

Recall that neural networks, as we have defined them, take real-valued vectors $\mathbb{R}^m$ as input, containing m real values. In contrast, a byte is a single 8-bit integer, which can have one of 256 ($= 2^8$) possible values. We cannot input a byte into the neural network directly and need to convert them into a real-valued vector.

Converting the 8-bit integer into a single floating-point number to input into the neural network might seem like a reasonable solution; however, neural networks process their inputs by multiplying them with the weight parameters, which only makes sense when the input values represent intensities (like brightness or loudness).

Instead, we use "one-hot encoding", which converts a byte into a $\mathbb{R}^{256}$ vector (since a byte can have 256 distinct values) where exactly one of the values is 1 and all others are 0. The byte's identity determines the location of the 1 within the vector. For example, a NUL byte (0) would be represented as

$$[1 \quad \underbrace{0 \cdots 0}_{\text{255 elements}}\ ].$$

and a nop in x86 (0x90, or 144) would be

$$[\ \underbrace{0 \cdots 0}_{\text{144 elements}} \ 1 \ \underbrace{0 \cdots 0}_{\text{111 elements}}\ ].$$

Multiplying a matrix A with a one-hot vector x is equivalent to extracting a column from A. In our case, the RNN multiplies a parameter matrix $W_{hx} \in \mathbb{R}^{m\times 256}$ with the one-hot input x, which is equivalent to selecting a column from W_{hx}. Effectively, each byte of input is represented with a h-dimensional vector during computation of the RNN, with the precise representation learned during training of the neural network. Such a mapping is often referred to as an *embedding*. Such embeddings have proved useful in other fields such as natural language processing, with embeddings of words into high-dimensional spaces showing interesting properties.

We could have instead considered encoding the byte as a $\mathbb{R}^8$ vector, with the elements corresponding to the eight bits and having values of 0 or 1. However, this representation imposes the constraint that the embedding of a particular byte is the sum of the embeddings of its constituent bits, even though the bits do not have compositional meaning in typical binary code. We do not further discuss this approach in the paper.

We want our output to serve as a binary classifier at each byte position. We use the softmax function to produce a probability distribution over whether the byte begins (or ends) a function or not. During training, the loss function sums over the error at each position within the sequence. The error at each position is the negative log of the probability that the neural network assigned to the correct answer. We penalize each false positive and false negative equally, without a weighting to discourage one at the expense of the other.

4.2 Optimization with stochastic gradient descent and *rmsprop*

In the beginning, we initialize the weights of the neural network randomly, uniformly drawn from a small range

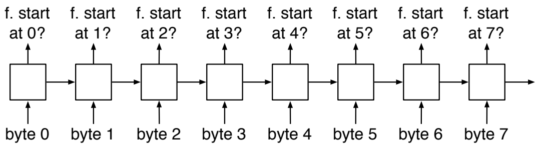


Figure 2: A depiction of the basic architecture of our approach.

near 0. A normal distribution with small variance and mean 0 also sees much use for this purpose. Having the proper initialization can prove crucial to whether we can successfully learn useful parameters for the neural network. We do not initialize the weights to 0, as this leaves the loss function on a saddle point and prevents optimization.

Recall the form of the loss function from Section 3. If we have N different items in the training data, then the loss function for the network requires evaluating the network over all training examples, since it takes the form

$$\frac{1}{N}\sum_{i=1}^{N} d(y_i, \hat{y}_i).$$

Due to how backpropagation works, computing the gradient also requires evaluating the network on all training data. Since we need to compute the gradient a very large number of times during optimization, we would like to avoid performing such an expensive step as a part of it.

Instead of computing the loss over all N items, we can instead compute it over a randomly-selected one at a time. The expectation of the gradient computed in this way equals the gradient averaged over all N examples. Optimization using these gradients is called *stochastic gradient descent*. It is possible to show that given a well-behaved convex function, stochastic gradient descent will find the minimum value. Even in the case of neural networks, where we lack such theoretical guarantees, experience shows that stochastic gradient descent can work quite well; in fact, since computing each gradient takes much less time, results show that stochastic gradient descent allows for much faster convergence in practice.

The most elementary gradient descent methods prescribe changing the parameters in the direction of the gradient each iteration, but optimization of some kinds of functions can benefit from moving in a slightly different direction. Consider a two-dimensional function, which when graphed looks like an elliptical bowl. Then along the axis in which we are closest to the minimum point, the gradient will have the largest magnitude, as the surface of the bowl is steeper in that direction, even though we should move further in the other axis and only a little bit in this one.

In this work, we use a method called *rmsprop* [19]; it involves keeping a running average of the magnitude of each dimension in the gradients seen so far. It then scales each dimension in the gradient, enlarging the dimensions which have a small average and shrinking those which have a large one. This follows the intuition given in the previous paragraph about the elliptical bowl.

We also scale the entire gradient each time by a step size. If the step size is too big, then the optimization might fail as the value of the function does not decrease; if the step size is too small, then optimization will progress slowly or get stuck at a local minimum. Often it makes sense to reduce the learning rate over time, since in the beginning we expect radically-incorrect weights (given their random initialization), whereas after some iterations, the weights should have nearly reached an optimum value. For our experiments, we scaled the learning rate by the inverse square root of the current iteration number (i.e., halved after 4 iterations, quartered after 16 iterations, and so on), which we found to work well.

4.3 Training with mini-batches

In stochastic gradient descent, we compute the gradient of the weights with respect to only one example in each iteration. However, this can cause a large variance in the gradients since each example might significantly differ from one to the next. So instead of computing the gradient over only one example at a time, it can help to average the gradients from a small number of examples, called a *mini-batch*.

While this increases the time needed for each iteration, it does so more modestly than it may initially seem. Evaluating the neural network with a single example involves a large number of matrix-vector multiplications, so we can efficiently and simultaneously evaluate for many examples by replacing these with matrix-matrix multiplications, especially when using highly-optimized linear algebra libraries.

In our application, since each example is a sequence of bytes from a binary, one might vary in length from another. However, to compute with mini-batches efficiently, we need to pack the examples together into a matrix or tensor with padding to extend too-short examples. Then all examples get evaluated for the same number of time steps, so it helps to put examples of similar length together in a mini-batch to avoid wasted computation. Also, we need to take care as to avoid computing the loss over those parts of the mini-batch added as padding.

4.4 Data preparation

For the task of function identification, we can intuitively expect that solving the problem likely does not require

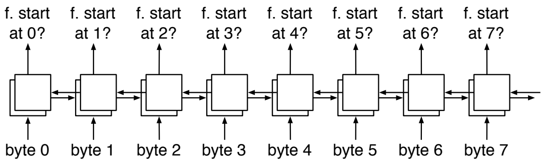


Figure 3: A bi-directional RNN. Note the horizontal arrows pointing in both directions. The forward-propagated and backward-propagated hidden states, represented by the overlapping squares, do not directly interact with each other. However, computing the output uses a concatenation of the two states.

remembering information from hundreds of thousands of bytes in the past or in the future. Code calling other functions can occur far away from the location of that function, and theoretically, we might track such references to help determine where functions occur. In practice, functions typically perform some series of steps at entry and exit, the patterns for which we can learn and should largely suffice for detecting functions.

Therefore, we use fixed-length subsequences taken from binaries instead of entire binaries themselves. Except in rare cases where functions occur near the boundary of the subsequence, there should be enough information to make the determination of the existence of a function or not. Similar to how stochastic gradient descent enables faster convergence by speeding up each update, computing the gradient on truncated sequences takes much less time and enables faster iterations.

We also try reversing the order of bytes in the input before providing it to the neural network, under the intuition that the function prologue, which identifies the beginning of a function and makes it recognizable as such, occurs after the position where we want to predict the beginning of a function. Since the RNN only has access to bytes from before the current position, not after, reversing the order should help the RNN learn.

4.5 Bi-directional RNNs

With the recurrent neural networks discussed in Section 3.4, the output at each time step depends only on the inputs which occur at that time step or before. This model makes sense in some applications where there exists an inherent temporal component to the input; for example, in real-time speech or handwriting recognition. For binary analysis, we have access to the entire binary at once, so there exists no need to confine ourselves in this way.

An extension which allows access to both the past and the future in making a prediction for the present is to combine two recurrent neural networks, one which operates from the beginning of the sequence to the end, and another which operates in the other direction. Figure 3 illustrates the approach.

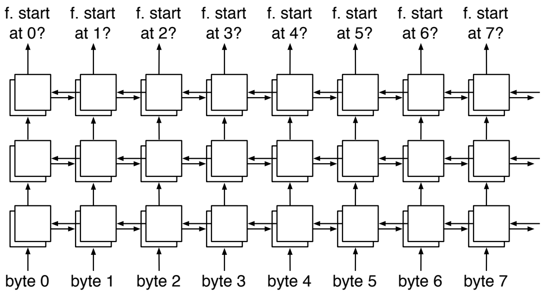


Figure 4: A multi-layer RNN with three bi-directional hidden layers. In the second and third layers, both the forward-propagated and backward-propagated states have access to either state from the previous layer.

In terms of graphical models, we could say that regular (unidirectional) RNNs behave like hidden Markov models, where the hidden state at each time step depends on only the hidden state of the previous time step. Then bidirectional RNNs are analogous to a chain conditional random field, since the hidden state there relates to the hidden states of both the previous and next time steps.

4.6 Multi-layer RNNs

The approaches we have described so far contain only one hidden layer. Depending on the complexity of the pattern we wish to learn, a single hidden layer may prove insufficient due to its limited capacity. If we limit ourselves to one hidden layer, achieving good results may require a very large one, which can significantly increase the amount of processing power required.

In other applications of neural networks like computer vision and speech recognition, using many smaller hidden layers has worked better than using one hidden layer of larger size. During the evaluation, we empirically verify the results of using one versus multiple hidden layers. Figure 4 illustrates an example architecture.

5 Evaluation

In this section, we describe the empirical results we obtained from training a variety of different models on a dataset of binaries. We seek to answer the following questions:

- Can recurrent neural networks successfully solve the problem of function identification in binaries?
- How much computational power do recurrent neural networks require for solving this task?

	ELF x86	ELF x86-64	PE x86	PE x86-64
Number of binaries	1,032	1,032	68	68
Number of bytes	138,547,936	145,544,012	29,093,888	33,351,168
Number of functions	303,238	295,121	93,288	94,548
Average function length	448.84	499.54	292.85	330.03

Table 1: Characteristics of the binary dataset used for evaluation.

	ELF x86			ELF x86-64		
	P	R	F1	P	R	F1
ByteWeight (func. start)	98.41%	97.94%	98.17%	99.14%	98.47%	98.80%
Our models (func. start)	99.56%	99.06%	99.31%	98.80%	97.80%	98.30%
Our models (func. end)	98.69%	97.87%	98.28%	97.45%	95.03%	96.22%
	PE x86			PE x86-64		
	P	R	F1	P	R	F1
ByteWeight (func. start)	93.78%	95.37%	94.57%	97.88%	97.98%	97.93%
Our models (func. start)	99.01%	98.46%	98.74%	99.52%	99.09%	99.31%
Our models (func. end)	99.24%	98.35%	98.79%	99.28%	99.20%	99.24%

Table 2: Function start and end identification: summary of our best results, and comparison with previous work. “P” is precision and “R” is recall. Results of previous work comes from Table 3 of Bao et al. [2]; they did not attempt to identify function ends independently, so we lack those results here.

- How do variations in the model’s design affect the performance?

We ran our experiments on Amazon EC2 using `c4.2xlarge` instances, each of which contains 8 cores of a 2.9 GHz Intel Xeon processor and 15 GB of RAM.

5.1 Dataset

Our dataset comes from Bao et al. [2], consisting of 2200 separate binaries. 2064 of the binaries were for Linux, obtained from the `coreutils`, `binutils`, and `findutils` packages. The remaining 136 for Windows consist of binaries from popular open-source projects. Half of the binaries were for x86, and the other half for x86-64. Half of the Linux binaries were compiled with Intel’s `icc`, while the other half used `gcc`. The binaries for Windows were compiled using Microsoft Visual Studio. Each binary was compiled with one of four different optimization levels. Table 1 summarizes some statistics from the dataset.

Following the procedure in Bao et al. we trained a separate model for each of the four (architecture, OS) configurations. To report comparable results, we also use 10-fold cross-validation as in Bao et al.; we train ten models for each of the four configurations, where each of the ten models uses a different 10% of the binaries as the testing set.

5.2 Implementation

We implemented our models in Python using Theano [4], a linear algebra and automatic differentiation library designed to aid in implementation of machine learning and optimization methods. In Theano, we specify our model as operations on symbolic variables, allowing for construction of a computation graph that describes the operations necessary to compute the result. It can convert this graph into C/C++ code and automatically compute partial derivatives of functions through application of the chain rule.

While Theano can also compile code for use on the GPU, we only used the CPU in our experiments for simpler implementation. Also, while both training and evaluation of RNNs are amenable to parallelization, we also did not use multi-threading for our experiments, and instead ran an independent experiment on each core.

5.3 Summary of results

Tables 2 and 3 summarize our main experimental results. In both tables, we compare to the results as reported by Bao et al. [2], which are marked as “ByteWeight”.

For the function start identification problem, our methods consistently obtain F1 scores in the range of 98-99%. This is in line with the results from Bao et al., except on the PE x86 dataset where we improve by about 4 percentage points in F1 score.

For function boundary identification, we trained two

	ELF x86			ELF x86-64		
	P	R	F1	P	R	F1
ByteWeight	92.78%	92.29%	92.53%	93.22%	92.52%	92.87%
Our models	97.75%	95.34%	96.53%	94.85%	89.91%	92.32%

	PE x86			PE x86-64		
	P	R	F1	P	R	F1
ByteWeight	92.30%	93.91%	93.10%	93.04%	93.13%	93.08%
Our models	97.53%	95.27%	96.39%	98.43%	97.33%	97.88%

Table 3: Function boundary identification: summary of our best results, and comparison with previous work. “P” is precision and “R” is recall.

	ELF x86	ELF x86-64	PE x86	PE x86-64
Our models (func. boundary)	1061.76 s	1017.90 s	236.93 s	264.50 s
ByteWeight (func. start only)	3296.98 s	5718.84 s	10269.19 s	11904.06 s
ByteWeight (func. boundary)	367018.53 s	412223.55 s	54482.30 s	87661.01 s
ByteWeight (func. boundary with RFCR)	457997.09 s	593169.73 s	84602.56 s	97627.44 s

Table 4: Computation time for testing on the data set of 2200 binaries. Numbers for ByteWeight are taken from Bao et al. [2].

models separately for each dataset: one to find function starts, and the other to find function ends. We combine the predictions from each model using a simple heuristic:

- If we predict multiple function ends in sequence after a function start, ignore all but the last.
- If we predict multiple function starts in sequence before a function end, ignore all but the first.
- Otherwise, pair adjacent function starts and ends into a function boundary.

Except on the ELF x86-64 dataset, this allows us to obtain 97-98% in F1 score. In contrast, Bao et al. report 92-93%.

To obtain these results, we used bidirectional models with RNN hidden units (i.e., rather than GRU or LSTM) and one hidden layer of size 16. We trained each model on 100,000 randomly-extracted 1000-byte chunks from the corresponding binaries (or 100 megabytes in total). To clarify, this means that we run two separate recurrent neural networks forward and backward on a 1000-byte sequence from the binary. The forward and backward RNN each computes a $\mathbb{R}^{16}$ hidden representation, which are concatenated together and fed into a linear transformation and the softmax function, producing a probability distribution (a $\mathbb{R}^2$ vector) over whether that byte corresponds to the beginning (or end) of a function or not.

We used *rmsprop* with a step size of 0.1, which was scaled by the inverse square root of the current number of iterations. We used a batch size of 32, which means that in each iteration, we computed the gradients for 32 of the 1000-byte chunks, averaged them together, and applied them to the current weights (per the description in Section 4.3).

These *hyper-parameters* (like the step size, the batch size, and the output size of the RNN), unlike the weights in the neural network, cannot be trained using gradient descent. They have to be selected manually or through an exhaustive search. We selected ours informed by some smaller-scale experiments and our intuition.

5.4 Computation time

Training. In many other applications, neural networks have gained a reputation as requiring a lot of computational power to train. Indeed, it has become standard to train large neural networks using GPUs (graphical processing units), which excel at the large number of linear algebra operations that training a neural network requires. However, the networks we use are relatively small, so training with the CPU seemed to work fine.

In addition, determining the precise number of iterations to train a neural network remains more of an art than a science. Due to noisy gradients in stochastic gradient descent, and the non-convex objective, the accuracy of the neural network does not improve monotonically or predictably as the number of iterations increases. In fact, training for too long can cause the parameters to overfit the training data, and worsen the performance on the test data.

To avoid the issue, we trained our neural networks for the fixed time of two hours each, and report the performance after that. We set the duration of training by

	ELF x86	ELF x86-64	PE x86	PE x86-64
Start	98.95%	98.02%	95.56%	98.37%
End	97.83%	95.51%	94.99%	98.06%
Boundary	95.89%	92.67%	92.45%	95.91%

Table 7: Results with when trained on 10% of the data (F1 scores).

(wall-clock) elapsed time rather than the number of iterations or parameter updates, to avoid biasing in favor of more complicated but powerful architectures which might make more progress per iteration but also take longer to compute each one.

Producing the results in Tables 2 and 3 requires training 40 models, since there are 4 ISA/OS combinations and we used 10-fold cross-validation. Therefore, in total, 80 compute-hours were required. In contrast, Bao et al. [2] report that ByteWeight took 586.44 compute hours to train, over 7× longer.

Furthermore, we needed to train 40 models only for the purposes of matching the 10-fold cross-validation protocol used by Bao et al. We really only need four models to achieve equivalent results, which would take just 8 hours to produce. While abandoning cross-validation for training ByteWeight would presumably also save a significant amount of time, we can expect the factor to be less than 10 since much of the computation (extracting counts of short instruction sequences) occurs before splitting data for cross-validation, further widening the gap between ByteWeight and our method.

Testing. Table 4 summarizes the amount of time needed to run each method on the data set after training completes. Our method is hundreds of times faster than the equivalent complete version ByteWeight which computes function boundaries instead of just function ends.

The disparity mainly arises as our method works without conventional program analysis techniques, such as the static control-flow graph generation used by ByteWeight. We trained the neural network to directly identify both function start and ends, and combine them together using a simple algorithm to recover plausible function boundaries. In addition, the neural network operates directly on bytes rather than instructions, avoiding the need for a disassembly step. In contrast, ByteWeight computes a CFG starting from each identified function start both to identify more functions, and to compute the function boundary. These extra steps require a considerable amount of computation time, and yet our approach gives better results without them.

5.5 Experiments

In this section, we describe some smaller-scale experiments we performed in order to gain insight into how various choices we made in designing our method affects the accuracy of results. We trained each model for two hours, and we did not use cross-validation for these experiments to save on computation time.

Reducing training data. In Table 7, we describe results from training a randomly-selected 10% of the binaries in the dataset and testing on the remainder (normally, the fractions were switched), to simulate common applications of binary analysis where only a small amount of representative training data is available. Despite this, the results dropped by 2-3 points at most.

Unidirectional RNNs. For our main model, we used bidirectional RNNs where the output at each position depends on both previous and future inputs. In Table 5, we compare how bidirectional RNNs fare against the simpler unidirectional ones. As we might expect, the unidirectional RNNs do significantly worse than the bidirectional ones on every benchmark.

In Section 4.4, we speculated that on unidirectional RNNs, reversing the order of the input might provide better results if the bytes which come after, instead of before, a certain location in the binary provide more information about whether that location is the start or end of a function. We found that reversed inputs help with identifying ends of functions, and ordinary inputs with identifying starts. One reason for this may be that with optimization turned on, the compiler will insert no-ops between functions so that function starts occur at an aligned offset; the model can identify these to help find the start or the end.

Variations in model architecture. Table 5 also compares how Gated Recurrent Unit (GRU) and Long Short-term Memory (LSTM) fare against conventional RNNs. As we might expect, GRU and LSTM perform better than RNN in most of the benchmarks. The comparison between GRU and LSTM is more mixed. Since LSTMs take more time to run each iteration, and we trained for a fixed amount of computation time, they may not have converged as much to optimal parameters. Also, while GRU and LSTM are more powerful models than conventional RNNs, this was not enough to beat the bidirectional RNN.

We can also examine what happens when we vary the number of hidden layers or the dimensionality of the hidden layer. Table 6 shows the different results obtained using one hidden layer of size 8, two hidden layers of size 8, or one hidden layer of size 16. The larger models

	Function start identification				Function end identification			
	ELF x86	ELF x86-64	PE x86	PE x86-64	ELF x86	ELF x86-64	PE x86	PE x86-64
RNN	92.36%	86.51%	94.48%	97.07%	54.52%	61.20%	72.32%	77.34%
GRU	95.09%	92.64%	96.46%	98.26%	70.55%	72.21%	83.78%	85.95%
LSTM	94.32%	89.89%	95.72%	97.58%	70.69%	68.21%	79.58%	82.46%
RNN (rev.)	94.74%	76.05%	66.02%	83.47%	91.12%	84.91%	95.52%	95.68%
GRU (rev.)	95.92%	84.93%	78.97%	87.52%	95.33%	89.44%	96.77%	95.86%
LSTM (rev.)	94.18%	94.18%	72.48%	83.43%	94.84%	87.78%	97.09%	95.42%
Bidir. RNN	98.88%	96.06%	98.04%	99.42%	95.93%	92.94%	97.98%	99.25%

Table 5: Comparison of unidirectional RNNs with different hidden unit types and input directionality, on the function start and end identification problems. "(rev.)" indicates that we trained and tested the model with bytes in the binary reversed. All models (including the bidirectional RNN) had one layer and 8 hidden units. All percentages are F1 scores.

	Function start identification				Function end identification			
	ELF x86	ELF x86-64	PE x86	PE x86-64	ELF x86	ELF x86-64	PE x86	PE x86-64
Separate								
$h = 8, l = 1$	98.88%	96.07%	98.04%	99.42%	95.93%	92.94%	97.98%	99.25%
$h = 8, l = 2$	99.03%	97.69%	98.00%	99.43%	97.71%	94.49%	98.30%	99.19%
$h = 16, l = 1$	99.24%	98.13%	98.33%	99.50%	98.09%	95.74%	98.56%	99.24%
Shared								
$h = 8, l = 1$	97.79%	95.28%	97.30%	99.23%	95.86%	91.94%	97.08%	98.90%
$h = 8, l = 2$	98.60%	96.67%	97.96%	99.45%	97.41%	94.92%	97.58%	99.12%
$h = 16, l = 1$	98.29%	97.41%	98.42%	99.47%	97.20%	95.51%	98.32%	99.38%

Table 6: Comparison of bidirectional RNNs on the function start and end identification problems. Separate means two models were trained separately for predicting starts and ends; shared means one model does both. h is the size of the hidden layer and l is the number of layers. All percentages are F1 scores.

perform better, but it turns out that increasing the hidden layer size rather than the number of layers provides a slightly greater benefit.

Task sharing. In our prior experiments, we trained two separate neural networks for performing function start and end identification. However, we could instead train one model to recognize both; at each byte, the model would decide among four possibilities instead of two. This could halve the amount of training time required. Also, what the network needs to learn in order to recognize function starts probably overlaps considerably with learning to recognizing function ends, so a network which simultaneously performs both tasks may also learn faster and produce more accurate results.

Table 6 summarizes our experimental results for testing this hypothesis. Overall, the single model which performs both tasks seems to do slightly worse than having separate neural networks for each task. Perhaps the disadvantage incurred from needing to keep track of more information exceeds the advantages mentioned in the previous paragraph.

6 Discussion

Limitations. As with most other machine learning approaches, ours assumes that the same underlying generative process has created both the training set and the test set. If similar patterns from the training data do not exhibit themselves in the test data, our approach will fail to correctly identify the functions.

As a pathological case, consider what would happen if long sequences of instructions which have no effect were inserted at arbitrary locations in the binary, including in the middle of function prologues. Such insertions would cause the internal structure of the binary to differ from what the model saw in the training data, even though it has no affect on the functionality. We might easily remove these instruction sequences if they were simply NOPs (0x90 in x86), but we can imagine the ability to create arbitrarily complicated ones especially if they are allowed to be long. Results from computability theory, such as Rice's theorem, suggest that it could be very difficult (if not impossible) to filter out such sequences from the binary through static analysis.

	ELF x86						ELF x86-64					
	gcc			icc			gcc			icc		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1
O0	99.89%	99.95%	99.92%	99.85%	99.94%	99.90%	99.72%	99.56%	99.64%	99.77%	99.51%	99.64%
O1	99.37%	98.29%	98.82%	99.62%	98.22%	98.91%	98.87%	96.80%	97.83%	99.35%	96.97%	98.15%
O2	99.20%	98.45%	98.82%	99.57%	99.28%	99.43%	97.18%	96.58%	96.88%	98.93%	97.76%	98.34%
O3	99.28%	98.77%	99.02%	99.50%	99.31%	99.40%	96.83%	96.69%	96.76%	98.99%	97.66%	98.33%

	PE x86			PE x86-64		
	P	R	F1	P	R	F1
Od	98.96%	99.43%	99.19%	99.52%	99.39%	99.45%
O1	98.89%	97.21%	98.04%	99.48%	98.68%	99.08%
O2	99.05%	98.60%	98.82%	99.50%	99.14%	99.32%
Ox	99.16%	98.63%	98.90%	99.61%	99.14%	99.37%

Table 8: Performance of our function start identification model on different subsets of the dataset. Each percentage value represents the precision, recall, or F1 score on the binaries of a particular architecture, compiler, and optimization level combination.

As for RNNs, since they accumulate and transfer information in a sequential manner, the input from these irrelevant instructions could easily overwrite the parts of the hidden state necessary for making correct predictions about the locations of the function boundaries.

In some cases, we can foresee that our approach will require preprocessing of the data in order to obtain good results. For example, binaries which decompress or decrypt themselves at runtime would not contain recognizable code within the binary stored on disk. Given that such obfuscations affect all static binary analysis techniques, previous works have addressed the problem of detecting and reversing such transformations [10, 21].

Segmented results. In Table 8, we delineate how the accuracy results for function start identification with our model (as described in Section 5.3) differed among different subsets of the binaries as further segmented by compiler and optimization level.

As we might expect, the model does best when run on binaries compiled without any optimizations (labeled as O0 and Od in the table), given that those tend to have very clear indications at the beginnings of functions. Nevertheless, the model's performance remains roughly constant on binaries compiled with more optimizations, with the exception of `gcc` on Linux for the x86-64 architecture where the F1 score decreased by about 2.9 percentage points. Given that the training data contains examples with every optimization level and compiler used for testing, we would hope that the model can learn to recognize functions in all such cases. However, it seems that `gcc` can produce relatively challenging examples with more optimizations enabled. Since the x86-64 ABI passes some function arguments in registers, it is possible to avoid any manipulation of the stack and base pointers upon function entry.

Error analysis. We randomly sampled some of the binaries to manually inspect the errors made by the model in them. Specifically, we selected 5 binaries for each combination of compiler, optimization level, architecture, and OS, then examined the errors to identify some common features between them.

Here are some observations we made:

- Given the bidirectionality of the model, it seems to exploit the appearance of frequently-occurring sequences at the ends of the previous function in addition to typical function prologues. One obvious example are `ret` and its variants, used to return from function execution. The compiler also often inserted padding between functions (such as `nop` (`0x90`) and other no-op instructions with longer encodings, or in Windows binaries, `int3` which triggers an interrupt), the end of which the model would use to recognize the beginnings of functions.

- As a consequence of the above, false positives often occurred after `nop`, `ret`, and other instructions which usually appear at the end of a function. In fact, it would also find false positives within immediate values encoded into the code if they contained `0x90` or `0xc3`, the encodings of those instructions.

- False negatives often occurred when instructions that would typically occur in the middle of functions occurred at the beginning of a function, as we might expect. The first byte of the program was often also falsely not recognized as a function start, presumably due to the lack of context previous to it.

- As documented by Bao et al. [2], `icc` will generate functions with multiple entry points. Many of the false negatives occurred at the second entry points to functions, given that the instructions before it are not the ones which usually end functions.

- The behavior of the model was not easily characterized by simple rules on short sequences of instructions; for example, while many false positives occurred after `nop` and `ret`, this did not mean that the model marked all (or even a large fraction) of such positions as function starts. For relatively difficult cases like these, the precise content of the surrounding bytes might have a complicated effect on the answer produced by the model.

Future work. Although we have seen some experimental evidence about the performance of the RNN under various conditions, we lack a clear explanation of the internal mechanics of the model. One potential approach towards an explanation proceeds through an analysis of the eigenvector structure by linearizing the state of the network as it evolves over time and analyzing which eigenvectors of the linearized systems carry the task-relevant information [12]. This analysis can provide an understanding of how the network ignores irrelevant information while selecting, integrating, and communicating relevant information, and allows identification of which eigenvector(s) of the linearized system are responsible for these tasks performed by the network. However, if the neural network's parameters are available to adversaries interested in disrupting the accuracy of the model, they may be able to use such analyses to more effectively add extra instructions which are not orthogonal to the eigenvectors carrying the task-relevant information, thus preventing its transmission and significantly affecting the RNN's performance.

7 Related Work

Function identification. Given that function identification serves as a basis for many applications within binary analysis, it should not surprise that many past papers have discussed the topic. For example, Kruegel et al. [9] identify functions as a prelude to static disassembly, and Theiling [18] for inferring control-flow graphs. However, these do not focus specifically on function identification as a specific problem, so here we point out some other works that do.

Rosenblum et al. [14] first framed the function identification problem as a task for machine learning. They combine a logistic regression classifier that uses "idioms" (short patterns of instructions) with a conditional random field to impose some structure between predictions for related instructions. Karampatziakis [8] tackles the related problem of accurate static disassembly using similar machine-learning tools, and Jacobson et al. [7] extend the prior work by Rosenblum et al. to fingerprinting library wrappers which appear in binaries. Bao et al. [2] also address function identification using supervised learning, but use weighted prefix trees which require much less computation than Rosenblum et al.'s approach to train, but still seems to give results with high accuracy.

Some tools built for binary analysis provide function identification as part of their functionality, usually using relatively simple heuristics or hand-coded signatures: Dyninst [6] and IDA Pro are some examples.

Neural networks. Much research using neural networks have focused on domains with continuous input data, such as vision and speech. In contrast, binary code contains discrete, multinomial values, where there typically exists no obvious ordering relationship between the possible values (unlike intensities of light or sound, for example).

Natural language processing also involves multinomial values (typically sequences of words), and neural networks have been successfully used for some applications there. Bengio et al. [3] first used neural networks to make a *language model*. Language models give a probability distribution over the next word in a sentence given the words so far, and see usage in machine translation and speech recognition. Mikolov et al. [11] moved to using a RNN. More recently, Sutskever et al. [17], Bahdanau et al. [1], and Cho et al. [5] have used RNNs for machine translation, and Vinyals et al. [20] for parsing.

We could not find any previous works which applied neural networks to binary code, but some use them on source code. Zaremba and Sutskever [22] attempt to train recurrent neural networks to evaluate short Python programs. Mou et al. [13] learn a vector representation from ASTs for supervised classification of programs.

8 Conclusion

In this paper, we proposed a new machine-learning-based approach for function identification in binary code based on recurrent neural networks. To our knowledge, there exists no previous works which apply neural networks to any problems in binary analysis. We address this gap by demonstrating how to use recurrent neural networks for function identification, and empirically show drastic reductions in computation time despite achieving comparable or better accuracy on a prior test suite. We hope that this work can serve as an inspiration for further advancements in binary analysis through neural networks.

Acknowledgements

We would like to thank Philipp Moritz for help with the initial implementation and experiments. We acknowledge Kevin Chen, Warren He, and the anonymous reviewers for their helpful feedback. This work was supported in part by FORCES (Foundations Of Resilient CybEr-Physical Systems), which receives support from the National Science Foundation (NSF award numbers CNS-1238959, CNS-1238962, CNS-1239054, CNS-1239166); by the National Science Foundation under award CCF-0424422; and by DARPA under award HR0011-12-2-005.

References

[1] BAHDANAU, D., CHO, K., AND BENGIO, Y. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[2] BAO, T., BURKET, J., WOO, M., TURNER, R., AND BRUMLEY, D. Byteweight: Learning to recognize functions in binary code. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, Aug. 2014), USENIX Association, pp. 845–860.

[3] BENGIO, Y., DUCHARME, R., VINCENT, P., AND JANVIN, C. A neural probabilistic language model. *The Journal of Machine Learning Research 3* (2003), 1137–1155.

[4] BERGSTRA, J., BREULEUX, O., BASTIEN, F., LAMBLIN, P., PASCANU, R., DESJARDINS, G., TURIAN, J., WARDE-FARLEY, D., AND BENGIO, Y. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)* (June 2010). Oral Presentation.

[5] CHO, K., VAN MERRIENBOER, B., GULCEHRE, C., BOUGARES, F., SCHWENK, H., AND BENGIO, Y. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).

[6] HARRIS, L. C., AND MILLER, B. P. Practical analysis of stripped binary code. *ACM SIGARCH Computer Architecture News 33*, 5 (2005), 63–68.

[7] JACOBSON, E. R., ROSENBLUM, N., AND MILLER, B. P. Labeling library functions in stripped binaries. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools* (2011), ACM, pp. 1–8.

[8] KARAMPATZIAKIS, N. Static analysis of binary executables using structural svms. In *Advances in Neural Information Processing Systems* (2010), pp. 1063–1071.

[9] KRUEGEL, C., ROBERTSON, W., VALEUR, F., AND VIGNA, G. Static disassembly of obfuscated binaries. In *USENIX security Symposium* (2004), vol. 13, pp. 18–18.

[10] MARTIGNONI, L., CHRISTODORESCU, M., AND JHA, S. Omniunpack: Fast, generic, and safe unpacking of malware. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual* (2007), IEEE, pp. 431–441.

[11] MIKOLOV, T., KARAFIÁT, M., BURGET, L., CERNOCKỲ, J., AND KHUDANPUR, S. Recurrent neural network based language model. In *INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September 26-30, 2010* (2010), pp. 1045–1048.

[12] MOAZZEZI, R. *Change-based population coding*. PhD thesis, UCL (University College London), 2011.

[13] MOU, L., LI, G., LIU, Y., PENG, H., JIN, Z., XU, Y., AND ZHANG, L. Building program vector representations for deep learning. *arXiv preprint arXiv:1409.3358* (2014).

[14] ROSENBLUM, N. E., ZHU, X., MILLER, B. P., AND HUNT, K. Learning to analyze binary computer code. In *AAAI* (2008), pp. 798–804.

[15] SCHMIDHUBER, J. Long short-term memory: Tutorial on lstm recurrent networks. `http://people.idsia.ch/~juergen/lstm/sld004.htm`, 2003.

[16] SUTSKEVER, I. *Training recurrent neural networks*. PhD thesis, University of Toronto, 2013.

[17] SUTSKEVER, I., VINYALS, O., AND LE, Q. V. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems* (2014), pp. 3104–3112.

[18] THEILING, H. Extracting safe and precise control flow from binaries. In *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on* (2000), IEEE, pp. 23–30.

[19] TIELEMAN, T., AND HINTON, G. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

[20] VINYALS, O., KAISER, L., KOO, T., PETROV, S., SUTSKEVER, I., AND HINTON, G. Grammar as a foreign language. *arXiv preprint arXiv:1412.7449* (2014).

[21] YAN, W., ZHANG, Z., AND ANSARI, N. Revealing packed malware. *Security & Privacy, IEEE 6*, 5 (2008), 65–69.

[22] ZAREMBA, W., AND SUTSKEVER, I. Learning to execute. *arXiv preprint arXiv:1410.4615* (2014).

[23] ZAREMBA, W., SUTSKEVER, I., AND VINYALS, O. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329* (2014).

A Backpropagation

We can view backpropagation as repeated application of the chain rule. We sketch how it works using the following example of a three-layer network:

$$\begin{aligned} h_1 &= f_1(x; \theta_1) \\ h_2 &= f_2(h_1; \theta_2) \\ \hat{y} &= f_3(h_2; \theta_3) \end{aligned}$$

where $\theta_i = (W_i, b_i)$, and we have named all of the intermediate hidden values for convenience of reference. We wish to minimize the error between the predicted $\hat{y}$ and the true value y. For example:

$$L = d(y, \hat{y}) = ||\hat{y} - y||^2$$

Then we can compute the following partial derivatives using the chain rule:

$$\begin{aligned} \frac{\partial L}{\partial \hat{y}} &= 2(\hat{y} - y) & \frac{\partial L}{\partial \theta_3} &= \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \theta_3} \\ \frac{\partial L}{\partial h_2} &= \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h_2} & \frac{\partial L}{\partial \theta_2} &= \frac{\partial L}{\partial h_2} \frac{\partial h_2}{\partial \theta_2} \\ \frac{\partial L}{\partial h_1} &= \frac{\partial L}{\partial h_2} \frac{\partial h_2}{\partial h_1} & \frac{\partial L}{\partial \theta_1} &= \frac{\partial L}{\partial h_1} \frac{\partial h_1}{\partial \theta_1} \end{aligned}$$

Reassembleable Disassembling

Shuai Wang, Pei Wang, and Dinghao Wu
College of Information Sciences and Technology
The Pennsylvania State University
{szw175, pxw172, dwu}@ist.psu.edu

Abstract

Reverse engineering has many important applications in computer security, one of which is retrofitting software for safety and security hardening when source code is not available. By surveying available commercial and academic reverse engineering tools, we surprisingly found that no existing tool is able to disassemble executable binaries into assembly code that can be *correctly assembled back* in a fully automated manner, even for simple programs. Actually in many cases, the resulted disassembled code is far from a state that an assembler accepts, which is hard to fix even by manual effort. This has become a severe obstacle. People have tried to overcome it by patching or duplicating new code sections for retrofitting of executables, which is not only inefficient but also cumbersome and restrictive on what retrofitting techniques can be applied to.

In this paper, we present UROBOROS, a tool that can disassemble executables to the extent that the generated code can be assembled back to working binaries *without manual effort*. By empirically studying 244 binaries, we summarize a set of rules that can make the disassembled code *relocatable*, which is the key to *reassembleable disassembling*. With UROBOROS, the disassembly-reassembly process can be repeated thousands of times. We have implemented a prototype of UROBOROS and tested over the whole set of GNU Coreutils, SPEC2006, and a set of other real-world application and server programs. The experiment results show that our tool is effective with a very modest cost.

1 Introduction

In computer security, many techniques and applications depend on binary reverse engineering, i.e., analyzing and retrofitting software binaries with the source code unavailable. For example, software fault isolation (SFI) [33, 46, 2, 19, 18] rewrites untrusted programs at the instruction level to enforce certain security policies. To ensure program control-flow integrity (CFI, meaning that program execution is dictated to a predetermined control-flow graph) [1, 4, 43, 17, 29, 37] without source code, the original control-flow graph must be recovered from a binary executable and the binary must be retrofitted with the CFI enforcement facility embedded [50, 49]. Symbolic taint analysis [34] on binaries must recover assembly code and data faithfully. The defending techniques against return-oriented programming (ROP) attacks also rely on binary analysis and reconstruction to identify and eliminate ROP gadgets [44, 9, 47, 22, 39].

Despite the fact that many security hardening techniques are highly dependent on reverse engineering, flexible and easy-to-use binary manipulation itself remains an unsolved problem. Current binary decompilation, analysis, and reconstruction techniques still cannot fully fulfill many of the requirements from downstream. To the best of our knowledge, there is no reverse engineering tool that can disassemble an executable into assembly code which can be *reassembled* back in a fully automated manner, especially when the processed objects are commercial-off-the-shelf (COTS) binaries with most symbol and relocation information stripped.

We have investigated many existing tools from both the industry and academia, including IDA Pro [24], Phoenix [42], Dagger [12], MC-Semantics [32], SecondWrite [3], BitBlaze [45], and BAP [8]. Unfortunately, these tools focus more on recovering as much information, such as data and control structures, as possible for analysis purpose mainly, but less on producing assembly code that can be readily assembled back without manual effort. Hence, none of them provide the desired disassembly and reassembly functionality that we consider, even if the processed binary is small and simple.

Due to lack of support from reverse engineering tools, people build high-level security hardening applications based on partial binary retrofitting techniques, including binary rewriting tools such as Alto [35], Vulcan [16], Diablo [13], and binary reuse tools such as BCR [11] and TOP [48]. We consider binary rewriting as a partial retrofitting technique because it can only instrument or patch binaries, thus not suitable for program-wide transformations and reconstructions. As for binary reuse tools, they work by dynamically recording execution traces and combining the traces back to an executable, meaning the new binary is only an incomplete part of the original binary due to the incomplete coverage of dynamic program analysis.

Partial retrofitting has notable drawbacks and limitations:

- Patch-based rewriting could introduce non-negligible runtime overhead. Since the patch usually lives in an area different from the original code of the binary, interactions between the patch and the original code usually require a large amount of control-flow transfers.

- Patch-based rewriting usually relocates instructions at the patch point to somewhere else to make space for the inserted code. As a result, it requires the affected instructions to be relocatable by default.

- Instrumentation-based rewriting expands binary sizes significantly, sometimes generating nearly double-sized products.

- Binary reuse often requires a binary component to be small enough for dynamic analysis to cover; otherwise the correctness cannot be guaranteed.

Having investigated previous research on binary manipulation and reconstruction, we believe that it could be a remarkable improvement if we are able to automatically recover the assembly from binaries and make the assembly code ready for reassembly. When a binary can be reconstructed from assembly code, many high-level and program-wide transformations become feasible, leading to new opportunities for research based on binary retrofitting such as CFI, diversification, and ROP defense.

Our goal is quite different from previous reverse engineering research. Instead of trying to recover high-level data and control structures from program binaries which helps binary code analysis, we aim at a more basic objective, i.e., producing assembly code that can be readily reassembled back without manual effort, which we call the *reassembility* of disassembling. Although the research community has made notable progress on binary reverse engineering, reassembility is still somewhat a blank due to lack of attention. In this sense, our contribution is complementary to existing work.

With that said, we believe that the technical challenge is also a cause for the deficiency in binary reassembly support from existing tools. We have confirmed that the key to reassembility is making the assembly code *relocatable*. Relocation is a linker concept, which is basically for ensuring program elements defined in different source files can correctly refer to each other after linked together. Being relocatable is also a premise for supporting program-wide assembly transformations. In COTS binaries, however, the information necessary for making disassembly results relocatable is mostly unavailable. There has been research trying to address the relocation issue [11, 48, 26], but existing work mostly relies on dynamic analysis which is unlikely to cover the whole program.

In this paper, we present UROBOROS, a disassembler that does reassembleable disassembling. In UROBOROS, we develop a set of methods to precisely recover each part of a binary executable. In particular, we are the first to be capable of not only recovering code, but also data and meta-information from COTS binaries without manual effort. We have implemented a prototype of UROBOROS and tested it on 244 binaries, including the whole set of GNU Coreutils and the C programs in SPEC2006 (including both 32-bit and 64-bit versions). In our experiments, most programs reassembled from UROBOROS's output can pass functionality tests with negligible execution overhead, even after repeated disassembly and reassembly. Our preliminary study shows that UROBOROS can provide support for program-wide transformations on COTS binaries.

In summary, we make the following contributions:

- We initiate a new focus on reverse engineering. Complementary to historical work which mostly focuses on recovering high-level semantic information from binary executables or providing support for binary analysis, our work seeks to deliver *reassembility*, meaning we disassembles binaries in a way that the disassembly results could be directly assembled back into working executables, without manual edits.

- We identify the key challenge is to make the disassembled program *relocatable*, and propose our key technique to recover references among immediate values in the disassembled code, namely "symbolization".

- With reassembility, our research enables direct binary-based transformation without resort to the previously used *patching* method, and can potentially become the foundation of binary-based software retrofitting.

- We implement a prototype of UROBOROS and evaluate its strength on binary reassembly. We applied our technique to 244 binaries, including the whole set of GNU Coreutils and SPEC2006 C binaries. The experiment results show that our tool does correct disassembly and introduces only modest cost.

- Our disassembler produces "normal" assembly code in the sense that binaries reassembled from UROBOROS's output assembly can again be disassembled (and hence the name UROBOROS[1]), or be used to accomplish other reverse engineering tasks. We verify this by repeating the disassemble-reassemble loop for thousands of times on different binaries.

The remainder of the paper is organized as follows. We first discuss the related work and challenges in §2 and §3, respectively. We then present the design and implementation of UROBOROS in §5. The experimental results are presented in §6, followed by some discussions in §7. We conclude the paper in §8.

2 Related Work

This section reviews literature on binary disassembly, binary rewriting, and binary reuse.

2.1 Disassembly

As aforementioned, there is no disassembler known to us that can generate working assembly code from binaries whose symbol and relocation information is stripped. IDA Pro [24] is considered as the best commercial disassembler available on the market. It can decode binaries into assembly and further decompile assembly into C code for program analysis. However, the assembly code produced by IDA Pro cannot be directly used as the input of any assembler. As stated in its manual [21], assembly code produced by IDA Pro is meant for analysis and cannot be directly reassembled or recompiled.

SecondWrite [3] leverages multiple static analysis techniques to lift binaries into LLVM IR. It is reported that the recovered LLVM IR can be converted back into C code given the LLVM's IR-to-C backend. However, it is unclear to us how SecondWrite symbolizes the data sections and recovers the meta-data information of the binaries. The paper does not contain an evaluation on this recompilation functionality. Moreover, the IR-to-C backend has been removed from LLVM release since 3.1, because it is not mature enough to handle non-trivial programs [30].

Dagger [12] is another tool that translates native code into LLVM IR, but the implementation is far from complete. There is a pre-release version available online. We tried to use it to decompile a simple binary (compiled from a C program with only empty main function). The decompiler reported several errors and generated an LLVM IR file which cannot be compiled back into binary due to lack of some symbol definitions.

MC-Semantics [32] is yet another tool for native code to LLVM IR translation. We used MC-Semantics to decompile some quickly written mini programs. Although the code produced by MC-Semantics can be made binaries, the execution results of these binaries are not the same as the originals, which we believe is due to incorrect symbol references. In addition, different from previously reviewed work, MC-Semantics works at the scale of object files rather than executables. Lacking the ability to handle linked binary programs narrows its scope of application.

BAP [8] is a binary analysis platform that comes with a disassembler. It can lift assembly code to a BAP-defined high-level intermediate representation that can be further analyzed statically. Several reverse engineering tools have been built based on BAP, including the C type recovery tool TIE [28] and the C control-flow recovery tool Phoenix [42]. Although BAP provides solid support for binary analysis, the strength of its disassembler is also limited to analysis only.

There could be multiple reasons that existing tools fail on reassembling. One reason is the technical challenges such as separating code and data, symbolizing the data sections, etc. The other reason could be the difference in the design goals. Most existing tools aim to produce more readable code or code that can be analyzed, not for the purpose of translation and reassembly. We emphasize that the ability to reassemble the output from a disassembler can provide an enabling infrastructure, facilitating further research.

2.2 Binary Rewriting

Binary rewriting techniques can be either static or dynamic. Static binary rewriting is widely used in security hardening such as control-flow hijacking mitigation [47], software control-flow integrity enforcement [50, 49], and binary instrumentation [3, 35, 13, 16]. Most static binary rewriting tools make strong assumptions on the input binaries. For example, Vulcan [16], Alto [35, 13], and Diablo [13] require binaries to be compiled from specific compilers or require symbol information not stripped. SecondWrite [3] can patch binaries with new code and data, but the original content in the binary shall remain unmodified.

As aforementioned, typical static binary rewriting has to relocate instructions at the patch point to make room for newly inserted code. In order to make sure that the rearranged instructions can be relocated while still pre-

[1] Uroboros is a symbol depicting a serpent eating its own tail.

serving program semantics, a stub-based idea is adopted to redirect control flow from the original location to the relocated new place at run time [14, 50, 47]. Control transfer instructions, i.e., stubs, are inserted at memory addresses that are pointed to by some code pointers. This strategy broadens the application scope of binary rewriting tools. However, there could be a large amount of stubs inserted, thus incurring notable execution overhead and size expansion on the rewritten binaries.

Dynamic binary rewriting tools, such as Pin [31] and DynamoRIO [7], can trace the execution of a binary and instrument or patch the program on the fly. Dynamic rewriters are able to handle COTS binaries, whereas with the cost of considerable performance penalty. Also, dynamic binary rewriting requires the rewriter itself to be shipped with or embedded into the target binaries.

Dyninst [10, 20] is a tool that features both static and dynamic binary rewriting. It supports performance measurement and computational steering. It can disassemble the stripped binaries and instrument them statically or dynamically, but does not deliver reassembleable disassembling either.

2.3 Binary Reuse

Binary reuse is mostly based on dynamic analysis. One of the representative binary reuse tools is BCR [11]. BCR extracts and reuses functions from binaries with a hybrid approach. BCR first executes binaries in a monitored environment and records execution traces and memory dumps. Binaries are then statically disassembled starting from the entry point. In the disassembly process, the dynamically collected information is used to resolve the destinations of indirect branches. In the end BCR manages to extract a “closure” of code reachable from the entry point which can be reused by other programs. Clearly, the correctness of the reused code cannot be guaranteed if BCR does not cover all feasible execution paths.

In addition to BCR, there are other binary reuse tools that employs similar basic ideas, such as Inspector Gadget [26] and TOP [48]. While these tools have made improvements in different aspects, the fact that they all rely on dynamic analysis leads to the incompleteness issue, more or less. In general, these tools can only do partial binary retrofitting.

3 Challenges

We have briefly discussed the technical challenges for developing a disassembler which can deliver *reassembility*. In this section, we discuss these difficulties in more details. In this research, we assume that the binaries to disassemble are stripped COTS binaries, namely binaries without any relocation information or symbols, except those necessary for dynamic linking. We also assume that the binaries are compiled from unobfuscated C programs, without self-modifying features. The target hardware architectures of the binaries are x86 and x64. The binary executable format is the Executable and Linkable Format (ELF).

3.1 Raw Disassembly

In this paper, raw disassembly is referred to as the process of parsing the binary form of a program to its raw textual representation. The difficulty of raw disassembly can vary a lot in different situations. In the most general case, this problem is undecidable. One of the reasons is that the problem of statically determining the addresses of indirect jumps is undecidable [23]. Furthermore, the existence of advanced program features such as self-modifying code makes the problem harder. Another issue is that current computer architectures do not distinguish code and data, and there is no easy way for a raw disassembler to distinguish them either. This problem is further worsened by the variable-length instruction encoding used by, for example, the x86 instruction set architecture.

However, with years of intensive effort on improving related techniques, the state of the art can already reach a very high success rate when disassembling binaries compiled from practical legitimate C source code by mainstream compilers. A recent paper by Zhang et al. [50] proposed a novel raw disassembly method which combines two existing disassembly algorithms together. We reimplemented this algorithm and applied it to our evaluation set which includes 244 binaries. No errors were reported by the raw disassembler and subsequent evaluation also verified the correctness of this algorithm on our evaluation set. As a result, we do not consider raw disassembly, or binary decoding, as a major challenge to address in this research.

3.2 Reassembly

Successfully decoding the binaries is only the first step to the goal of this research. Ideally, binary reverse engineering tools should be able to support at least the following process:

- The reverse engineering tool disassembles the original binary into assembly code.
- Users can perform static analysis on the disassembled program.
- Users can perform transformations on the disassembled program.

- The transformed program can be assembled back into an usable binary executable, with all transformation effects retained.

Although it may not be obvious, the feasibility of the first three steps does not naturally imply the feasibility of the last step. There have been reverse engineering tools or platforms that can (partially) enable the first three steps [8, 45], but support for reassembly is still a blank.

As mentioned in the introduction, making the assembly code relocatable is the crux of reassembility. Figure 1 is an artificial example comparing relocatable and unrelocatable assembly code. In COTS binaries, information required for making disassembly results relocatable is unavailable. Most program transformations inevitably change binary layouts, but a reverse engineering tool has only very limited control over how the linkers assign memory addresses of the program elements, leading to situations illustrated by Figure 1. Note the memory cell located at address `0xc0` in the original memory, which is possibly a global variable. The raw disassembly process does not recognize the concrete value `0xc0` in the code as a reference. Thus when this unrelocatable assembly code is reassembled, the resulting binary will very likely be defected because the content of the memory cell at `0xc0` in the original binary may not be placed the same address in the new binary. In the relocatable assembly, however, the data originally living at `0xc0` is given a symbolic name, and the concrete address `0xc0` is replaced by a reference to this name. This is why relocatable assembly can be reassembled into a working executable.

As suggested by the example, if a reverse engineering tool seeks to reassemble the transformed assembly code into a working executable, it has to identify program elements whose addresses could possibly change in the new binary, and lift concrete memory addresses referring to them to abstract symbolic references. Obtaining relocatable assembly from a COTS binary is non-trivial because very little auxiliary information in the binary can be utilized to help identify references among concrete values. Essentially, the problem can be generalized as the following: given an immediate value in the assembly code (either in a code section or data section), is it an memory address or a constant? Although this looks like a typical type analysis problem, in the context of binary reassembly it becomes much more challenging. From a static point of view, since most machine assembly languages are untyped, type inference is difficult in the first place. Compared to high-level programming languages, assembly languages lack explicit syntax for denoting procedure boundaries and basic control-flow logic, making static analysis even more difficult. What is worse, many references live in the data sections, some of which are indirectly referred to by the code via numerous reference hops. At present, most proposed program analysis techniques, either static or dynamic, are code oriented, lacking the capability of analyzing the property of a given data chunk. Finally, reassembly has almost zero tolerance for type inference errors, because a single false positive or false negative can place the reassembled binary in a non-functional state.

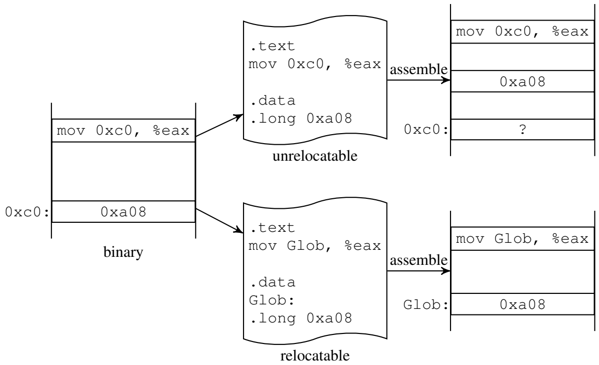


Figure 1: Relocatable and unrelocatable assembly code

Solving the relocatable problem in binary disassembly is the main purpose and contribution of this paper. In the rest of the paper, we call the process of identifying references among immediate values in the raw assembly the process of "symbolization". To distinguish the concept from the traditional meaning of disassembling, we call our work *reassembleable disassembling* that generates relocatable assembly code.

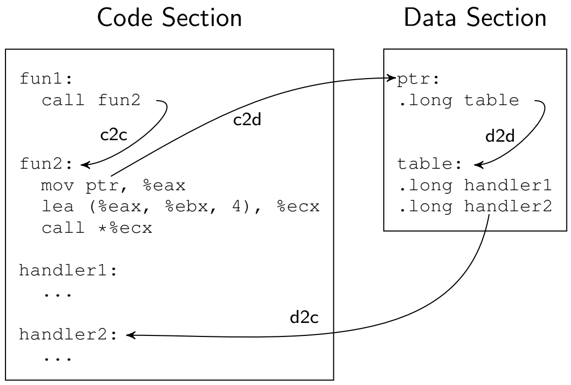
In addition to relocation information, a full-fledged disassembler also needs to recover some meta information to make the reassembly feasible. Meta-data sections in a binary executable provide information to direct some link-time and runtime behavior of the program. They should also be recovered properly in order to ensure the reassembled binaries are semantic-equivalent to the originals.

4 Symbolization

This section describes the symbolization problem in detail and presents our solution.

4.1 Classification

There are four types of symbol references that we need to identify for reassembility. The classification is based on two criteria—where a reference lives and where a reference points. Basically, we divide the binary into two parts, i.e., the code sections and the data sections, whose contents are as suggested by their names. For ELF binaries on Unix-like platforms, typical code sections include



```
fun1:
  call fun2

fun2:
  mov ptr, %eax
  lea (%eax, %ebx, 4), %ecx
  call *%ecx

handler1:
  ...

handler2:
  ...
```

```
ptr:
.long table

table:
.long handler1
.long handler2
```

Figure 2: Different types of symbol references in assembly code

.text and .init etc. Typical data sections include .data, .rodata, .bss, etc. A symbol reference can live in either code sections or data sections, and can point to either code sections or data sections as well, leading to a total of four types. Figure 2 is an example showing all four types of symbol references. We give each of them a short name, i.e., c2c, c2d, d2c, and d2d references.

4.2 Method

When it comes to solving the symbolization problem, we have considered various potential solutions. Due to the reasons listed in §3.2, we conclude that no existing program analysis technique can handle the symbolization problem in our special context. Hence, we decide to turn to another direction. In this work, we identify the immediate values which are actually symbol references by applying several matching rules inferred from our study on a large amount of binaries. Although some of these strategies may not seem exciting at the first sight, they work surprisingly well in our evaluation on 244 binaries compiled from C code.

Since we are solving the symbolization problem in an empirical way, the matching strategies are all based on certain assumptions. Depending on whether an assumption is accepted or not, different rules are applied for symbolization. We now introduce the assumptions and the corresponding symbolization strategies.

At the point of symbolization, we assume that we have already obtained the raw assembly decoded from binaries using the algorithm by Zhang et al. [50], so we can get all immediate values that appear in a binary. There are two kinds of immediate values—constants used as instruction operands and the byte stream living in data sections. Among all these immediate values, some can be excluded from being considered for symbolization at the first place. Unless a program intentionally causes memory access errors, which is rarely the case, an immediate value can be a reference to symbols *only if* this value falls in the address space allocated for the binary. For a binary of reasonable size, the utilization of address space is usually sparse, so there is a wide range of address space which is actually invalid.

Assuming all immediate values are potential symbol references, we can filter out obviously invalid references based on their target addresses. According to our symbol reference classification in §4.1, a reference can only point to code sections or data sections; especially, if a reference points to code sections, the destination must be the starting address of some instruction. Our study on 244 binaries shows that this simple filter is sufficient to identify c2c and c2d symbol references with full correctness.

The really challenging part is data section symbolization, i.e., identifying d2c and d2d references. The first step of data section symbolization is to slice the data sections, which are continuous areas of binary bytes, into individual values of different lengths. Since the raw disassembly process does not assign the data sections any semantics, there is no ready-made guidance on how they should be sliced. Regarding this problem, we introduce the first assumption which is about binary layout:

> (A1) *All symbol references stored in data sections are n-byte aligned, where n is 4 for 32-bit binaries and 8 for 64-bit binaries.*

Since unaligned memory accesses cause considerable performance penalty, compilers tend to keep data aligned by its size. For data alignment, compilers can even sacrifice memory efficiency by inserting padding into data sections. With that said, A1 stays as an assumption because occasionally programmers do want non-aligned data layout. For example, the "packed" attribute supported by GCC allows programmers to override the default alignment settings.

If we accept assumption A1, only *n*-byte long values which are also *n*-byte aligned in data sections are considered for symbolization. Alternatively with A1 rejected, all *n*-byte long memory content in data sections are considered for symbolization. This is implemented as an *n*-byte sliding window which starts from the beginning of a data section and scans through the entire section in a *first-fit* manner. Each time the sliding window moves forward 1 byte and check the value of the covered bytes. If the value fulfills the basic requirements for being a d2d or d2c reference, it will be considered for symbolization and the sliding window advances *n* bytes forward. In case that the value does not meet the requirements, the sliding window moves forward 1 byte only.

In addition to assuming the characteristics of binaries, making assumptions on user requirements for our tool also helps improve its performance. As stated earlier, the goal of symbolization is to make assembly code relocatable so that users can perform program-wide transformations on the assembly and then assemble it back to a working executable. From our experience, most transformations on assembly only touch the instructions without modifying the original data. If we make the following assumption

> (A2) *Users do not need to perform transformation on the original binary data.*

then we can keep the starting addresses of data sections the same as their old addresses when performing reassembly, by providing a directive script to the linker. In this way, we can ignore d2d references during symbolization simply because we do not need them to be relocatable anymore. Thus, with A2 accepted, only the immediate values that fall within code sections (d2c references) are considered for symbolization. Contrarily, without deterministically fixing the starting addresses of data sections in the new binary, the immediate values that fall within either code sections or data sections are considered for symbolization.

We want to avoid symbolizing d2d references because they are used in a very flexible manner. On the other hand, there are more common patterns in d2c references which can be exploited by our symbolization method. We summarize the patterns with the following assumption:

> (A3) *d2c symbol references are only used as function pointers or jump table entries.*

By accepting A3, an *n*-byte value in data sections is lifted to a d2c reference if it is the starting address of some function, or it forms a jump table together with other *n*-byte values adjacent to it. Otherwise with A3 rejected, an *n*-byte data section value is symbolized whenever it is within the address space of code sections.

When A3 is taken, we will need to know whether a code section address is the start of a function. We also need to clarify what a jump table would be in the binary form. Identifying function beginnings in a binary is not a new research topic. Based on machine learning techniques, recent research [6] can reportedly identify function starting addresses with over 98% precision and recall. To avoid reinventing the wheel, we assume we have already known all the function start addresses. Since the binaries used in our research are all compiled from source code, we are able to get the ground truth by controlling the compilation and linking process.

Regarding the identification of jump tables, our algorithm is as follows:

- *Jump table start.* We traverse the data sections from the beginning to the end. If the address of an *n*-byte value is referred to by an instruction as the operand, it is considered as the first entry of a new jump table.
- *Jump table entry.* If an *n*-byte value follows an already identified jump table entry, this value is also considered as an entry as long as it refers to instructions within the same function that previous entries point to.

The three assumptions A1, A2, and A3 are the basics of our symbolization method. With different choices of an assumption being applied or not, we can derive different strategies when processing a binary. §6 has a detailed evaluation on the correctness of reasonable combinations of these assumptions.

5 Design and Implementation

5.1 Overview

The architecture of UROBOROS is shown in Figure 3. UROBOROS consists of two main modules—the disassembly module and the analysis module. The disassembly module decodes instructions with raw disassembling (§5.2) and dumps the data sections. The analysis module symbolizes memory references in both code and data sections (§4) and recovers the meta-information from the dumped content (§5.4). UROBOROS also recovers part of the control-flow structures from direct transfers so that it provides basic support for program-wide transformation (§5.3).

The disassembly module employs an interactive process to validate disassembled code from a linear disassembler. The linear disassembler decodes the code sections and dumps out all data and meta information sections. A validator is then invoked to correct disassembly errors due to "data gaps" embedded inside code sections. The details are presented in §5.2.

After the raw disassembly is over, the dumped code, data, and meta-data are sent to the analysis module. This module identifies symbol references among immediate values in the code and data. As elaborated in §4, we propose three assumptions for reassembleable disassembling. The corresponding strategies are implemented in UROBOROS to guide the symbolization process. UROBOROS can be configured to utilize different combinations of assumptions for symbolization. We give a detailed evaluation on the correctness of different strategies in §6.1.

Given the symbolized instructions, the analysis module also partially recovers the control flows based on direct control-flow transfers. With the relocatable assembly and the basic control-flow structures, users of UROBOROS can easily perform advanced program anal-

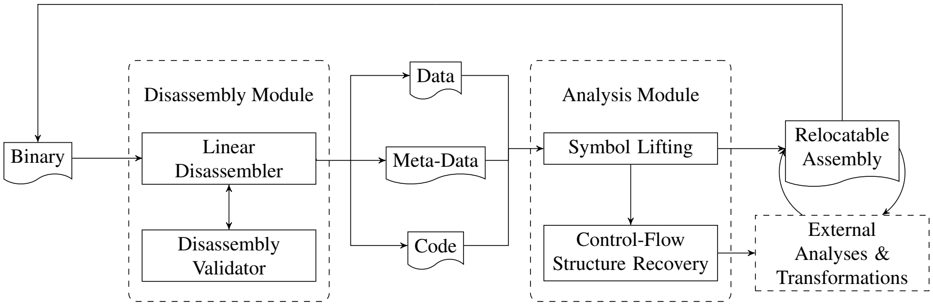


Figure 3: The architecture of UROBOROS

ysis and program-wide transformations before they assemble the code back to binaries.

Finally, we emphasize that the assembly code generated and transformed by UROBOROS can be directly assembled back as a working binary by normal assemblers. In particular, the binary output is indeed a *normal* executable file without any abnormal characteristics such as patched or duplicated sections. Therefore, the reassembled binary can be disassembled again by UROBOROS or be processed by other reverse engineering tools.

We have implemented a prototype of UROBOROS in OCaml and Python, with a total of 13,209 lines of code. Our prototype works for both x86 and x64 ELF binaries.

5.2 Disassembly

In our prototypical implementation, the linear disassembler employed by UROBOROS's disassembly module is `objdump` from GNU Binutils. We implement an interactive disassembly process originally proposed in BinCFI [50].[2] In this process, the disassembler communicates with a validator which corrects disassembly errors due to "data gaps" between adjacent code blocks. The interactive procedure is as follows:

- `objdump` tries to decode the input binary for the first time.
- The validator examines the output and check if there are explicit errors reported by `objdump`. In case there are no errors, the raw disassembly process terminates. Otherwise, the validator assumes the errors are caused by data embedded in code and computes the upper and lower bounds of identified "data gaps".
- With the computed range of identified "gaps", the validator guides `objdump` to decode the binary again, with those "gaps" skipped.
- Repeat this decode-validate process until no error occurs or the running time of the whole process reaches a time limit specified by users.

We leverage three rules proposed in BinCFI to validate the disassembly results and locate the data "gaps", i.e., "invalid opcode", "direct control transfers outside the current module", and "direct control transfer to the middle of an instruction". Since identifying bounds of each data gap can rely on the control-flow information of decoded instructions, the validator occasionally leverages UROBOROS's analysis module to retrieve the control-flow information.

5.3 Support for Program Transformation

UROBOROS provides basic support for program-wide transformations by partially recovering control-flow structures of the decoded instructions. We collect all the control transfer instructions to divide each function into multiple basic blocks. Control-flow graphs are rebuilt on top of these basic blocks. As a prototype, UROBOROS currently only processes direct control transfers. Regarding the intractable indirect transfers, a potential solution is to use value set analysis (VSA) [5] for destination computation. We leave including indirect control transfers in the CFG as future work.

5.4 Meta-Information Recovery

UROBOROS recovers the program-linkage table (PLT) and the export table in ELF binaries. The PLT table supports dynamic linkage by redirecting intra-module transfers on its stubs to external functions. As the base address of the PLT table can change after reassembling, we translate the memory references to PLT stubs to their corresponding external function names, and let the linker to rebuild the PLT table with correct memory references during link time. In particular, this table is dumped out from the input binary and parsed into multiple entries, each containing the memory address of a PLT stub with

[2] The BinCFI tool is available open source. We choose to reimplement the algorithm to make the codebase of UROBOROS more consistent such that it is fully automated and easy to extend. We refer readers to BinCFI [50] for the details of the disassembly process.

its corresponding function name. Next, we scan the program and identify the addresses that match to a table entry. These addresses are then replaced by the corresponding function name.

Symbols need to be "exported" so that other compilation units can refer to them. The exported symbols together with their memory addresses are recorded in the export table. As ELF binaries do not keep a standalone export table, we construct this table by searching for all global objects in the symbol table. The symbol name of each entry and its memory address are then kept in a map. The export table can help identify functions and variables that are only referred to by other compilation units. We iterate each entry of the export table to insert symbols and `.globl` macros to the corresponding addresses.

For typical ELF binaries compiled from C code, `.eh_frame` and `.eh_frame_hdr` sections are used by compilers to store information for some rarely-used compiler-specific features, such as the "cleanup" attribute supported by GCC. For these sections, we dump the content out and directly write them back to the output. These sections are also used to store exception information for C++ programs. Regarding this, we have a related discussion in §7.

5.5 Position Independent Code

Position independent code (PIC) typically employs a particular routine to obtain its memory address at run time. This address is then added by a fixed memory offset to access static data and code. According to our observation, the routine below is utilized by PIC code in 32-bit binaries to achieve relative addressing.

```
804C452: mov (%esp),%ebx
804C456: ret
```

PIC code invokes this routine by a `call` instruction, and register `ebx` is then assigned the value on top of the stack, which equals the return address. Uroboros identifies this instruction pattern, traces the usage of `ebx`, and rewrites the instructions that add `ebx` with memory offsets to a relocatable format.

An example is shown in Figure 4. Once we identify a `call` instruction targeting the above sequence, we calculate the absolute address by adding `0x804c466` with offset `0x2b8e`, which equals `0x804eff4`. By querying the section information from ELF headers, `0x804eff4` equals the starting address of `.got.plt` table, and we rewrite offset `0x2b8e` to the corresponding symbol, which is `_GLOBAL_OFFSET_TABLE` in this case.

Theoretically PIC could use other patterns besides the above sequence to obtain its own memory address; the above instruction sequence is, however, the only PIC pattern we encountered after testing a broad range of real world applications (compiler and platform information is disclosed in § 6).

```
804c460: push %ebx
804c461: call 804c452
804c466: add $0x2b8e,%ebx
804c46c: sub $0x18,%esp

804c460: push %ebx
804c461: call S_0x804C452
804c466: add $_GLOBAL_OFFSET_TABLE_,%ebx
804c46c: sub $0x18,%esp
```

Figure 4: PIC code reuse

As for x64 architectures, RIP-relative [25] memory references allow assembly code to access data and code relative to the current instruction by leveraging the `rip` register and memory offsets, which makes the implementation of PIC more flexible. In the raw disassembly output, instructions utilizing this mode are commented by `objdump` with the absolute addresses they refer to. We identify the comments, symbolize the memory offsets, and insert labels to the corresponding absolute addresses.

5.6 Redundancy Trim

When a binary is dynamically linked to libc, the prologue and epilogue functions of the library are automatically added to the final product. Uroboros attempts to support multiple iterations of the disassemble-reassemble process. Each time the binary is assembled, a new copy of the prologue and epilogue functions are inserted, which unnecessarily expands binary size. Some tentative experiments show that binary size can grow 5 to 6 times larger with respect to the original, if we perform the disassemble-reassemble iteration for 1,000 times.

We cannot identify the prologue and epilogue functions in COTS binaries as the symbol information has been stripped. However, after the first disassemble-reassemble attempt, we get an unstripped binary with sufficient information indicating which functions are added by the linker. If we are to do another disassemble-reassemble round, Uroboros can skip these functions in the disassembly phase.

Another source of redundancy is the padding bytes in data sections. In ELF binaries,there are three data sections (`.data`, `.rodata`, and `.bss`) that have padding bytes at the beginning. As these padding bytes are not used, we remove them from the recovered program before reassembling.

With the code and data redundancy trimmed, binary size expansion is reduced to almost zero, no matter how many times a binary is disassembled and reassembled.

5.7 Main Function Identification

In a compiler-produced object file, the symbol information of the main function is exported so that it can be accessed by the libc prologue functions in the linking process. However, as this symbol information in executable file is stripped in COTS binaries after linking, we need to recover and export it before reassembling.

Through our investigation, we found that the code sequence shown below is typically used to pass the starting address of main to libc prologue function libc_start_main.

```
push $0x80483b4
call 80482f0 <__libc_start_main@plt>
hlt
```

The first argument of libc_start_main, which is 0x80483b4 in this example, is recognized as the starting address of the main function. We insert a label named main and the type macro .globl main in the output at this address.

5.8 Interface to External Transformation

As briefly discussed in §1, existing binary software security applications mainly rely on patch-based or instrumentation-based binary manipulations. We argue that given the assembly program and support for program-wide transformation from UROBOROS, we can bridge external instrumentation and analysis techniques with binary retrofitting application development. The program-wide security instrumentation such as CFI, ROP attack mitigation, randomization and software diversification could be ported on the basis of UROBOROS to legacy binaries, without the inefficiency, cumbersomeness and restriction brought by previous binary manipulation methods.

In order to demonstrate that UROBOROS is an enabling tool that makes analysis and transformations applicable to legacy binaries in general, we implement a diversification transformation based on basic block reordering. After disassembly, we walk through each function and randomly select two basic blocks from its CFG as the reordering targets. Control-flow transfer instructions and labels are inserted in the selected blocks, their predecessors, and successors to guarantee semantic equivalence. We perform this reordering *iteratively*, namely the output of each iteration becomes the input of the next round. We conducted a quick experiment on gzip. The disassembly-transformation-reassembly process was iterated 1,000 times. The effectiveness of the diversification transformation is evaluated by the elimination rate of ROP gadgets measured by the ROP gadget detector ROPGadget [40]. From this preliminary study, we find that it is much easier than binary rewriting to perform binary-based software retrofitting based on UROBOROS. As the ROP defense is not the focus of this research, we omit the detailed results in this paper.

Table 1: Programs used in UROBOROS evaluation

Collection	Size	Content
COREUTILS	103	GNU Core Utilities
REAL	7	bc, ctags, gzip, mongoose, nweb, oftpd, thttpd
SPEC	12	C programs in SPEC2006

6 Evaluation

We evaluate UROBOROS with respect to correctness, cost, and its ability to support program-wide transformation. The correctness verification examines whether UROBOROS's reassembly is semantic preserving. Evaluation on the cost of UROBOROS reveals its reassembly's impact on binary size and execution speed, and also the running time of UROBOROS itself. As presented in §5.8, we study UROBOROS's support for binary-based software retrofitting, by implementing a basic block reordering algorithm to diversify disassembled binaries and eliminate ROP gadgets. As we have emphasized, UROBOROS is an enabling tool for other security hardening techniques. However, as goal-driven software security hardening is out of the scope of this paper, we do not present the detailed experiment results here.

We use three collections of binaries compiled from C code to evaluate UROBOROS. The first set, referred to as COREUTILS, is the entire GNU core utilities including 103 utility programs for file, shell, and text manipulation. The second set, called REAL, consists of 7 real-world programs picked by us, covering multiple categories such as floating-point and network programs. The last set subsumes all the C programs in the SPEC2006 benchmark suit, thus will be denoted by SPEC. Details of each collection are listed in Table 1. In the evaluation we compile all programs for both 32-bit and 64-bit targets. Since there are 122 programs, the number of tested binaries is 244 in total. The compiler is GCC 4.6.3, using the default configuration and optimization level of each program. All experiments are undertaken on Ubuntu 12.04. For each test case, we use the strip tool from GNU Binutils to strip off the symbol information and debug information before testing.

Table 2: Functionality test input for REAL

Program	Test Input
bc	Test cases shipped with the program
gzip	Test cases shipped with the program
ctags	Parse a C source file of 152,270 lines
oftpd	Login and fetch a large file
thttpd	Request some web pages & a large file
mongoose	Request some web pages & a large file
nweb	Request some web pages & a large file

6.1 Correctness

We verify the correctness of UROBOROS's reassembility in two steps. First, we execute binaries assembled from UROBOROS's output with test input shipped with the software. Both COREUTILS and SPEC have test cases shipped with the software by default. As for the REAL programs, most of them do not have test cases, so we develop input by ourselves to verify the major functionality. The input we use for testing the REAL collection is listed in Table 2.

Second, we examine the false positives and false negatives of our symbolization process for all the binaries of the three collections. In our context, a false positive is an immediate value that we mistakenly symbolize, while a false negative is a symbol reference that we fail to identify.

As described in §4, we have different assumptions to guide the symbolization process, so the correctness of different assumption combinations are verified. Since the three assumptions are orthogonal, there are eight different combinations with the choices of the three assumptions. With limited resources, it is difficult to test all 244 programs on all assumption sets. With some tentative experiments, we found that A1 is an assumption which greatly improves the overall performance of our disassembly and reassembly method. Therefore, we reduce the eight candidates to five by always including A1 except in the empty assumption set. In detail, the five assumption sets applied are {} (empty set), {A1}, {A1, A2}, {A1, A3}, and {A1, A2, A3}.

For all tested assumption sets, all reassembled binaries from COREUTILS and REAL pass the functionality tests. Some binaries from SPEC, however, fail to pass the tests, which are listed in Table 3. With the assumption set {A1, A2, A3}, only the 32-bit version of `gobmk` from SPEC (out of 244 cases in total) fails the functionality test. By inspecting this defected binary, we successfully locate the cause of failure. Some 4-byte sequences in the data sections happen to contain the same value as the starting address of a function, but they are not code pointers. UROBOROS incorrectly symbolizes them, leading to false positives. After we correct these errors, `gobmk` successfully passes the test.

For symbol-level correctness verification, we provide the statistics on false positives and false negatives of symbolization. A false positive is an immediate value that should not have been symbolized. A false negative is an immediate value which should be symbolized but failed to be after our symbolization process. We obtain the ground truth by parsing the relocation information provided by the linker.

We have verified all binaries in this step. Due to limited space, we only list the results for non-trivial cases, namely programs with at least one symbolization false positive or false negative with any assumption combination. Table 4 and 5 show the false positive and false negative analysis for 32-bit binaries, and Table 6 reports false positive analysis for 64-bit binaries. There are no false negatives on any of the 64-bit binaries. We emphasize in particular that, *with {A1, A2, A3} applied, among all the 244 binaries, only `gobmk` has a few false positives, and none has false negatives.*

The results of symbol-level verification are highly synchronized with the results from the first stage—binaries reassembled with no false positives or false negatives can pass all test cases. The results show that symbolization errors are found in `gobmk` no matter which assumption set we apply. In particular, we have verified that symbolization errors found in `gobmk` when applying {A1, A2, A3} are all caused by program data colliding with some function starting addresses. These collisions cause a functionality test failure for 32-bit `gobmk`, but the 64-bit version can pass the test due to the incompleteness of test input. In summary, the two stages of verification together imply that all three assumptions proposed for symbolization are reasonable.

Although the symbolization errors occurring in the case of `gobmk` seem conceptually "general", our study shows that the collisions are actually rare in practice, unless the disassembled binary has very large data sections like `gobmk` does. See Appendix A for the symbolization errors in `gobmk`. On the other hand, UROBOROS can successfully disassemble large and complicated binaries like `gcc` and `perlbench`. Overall, the results from two stages of correctness verification suggest that UROBOROS is a promising tool with remarkable practical value.

6.2 Cost

The cost of UROBOROS manifests from three aspects: size expansion of reassembled binaries, execution overhead of reassembled binaries, and the processing time of UROBOROS itself.Due to space restrictions, we only report the evaluation results on 32-bit binaries in this paper.

Table 3: Dynamic test results on reassembled binaries

Assumption Set	Binaries Failing Functionality Tests	
	32-bit	64-bit
{}	h264ref, gcc, gobmk, hmmer	perlbench, gcc, gobmk, hmmer, sjeng, h264ref, lbm, sphinx3
{A1}	h264ref, gcc, gobmk	perlbench, gcc, gobmk
{A1, A2}	h264ref, gcc, gobmk	perlbench, gcc, gobmk
{A1, A3}	gobmk	gcc, gobmk
{A1, A2, A3}	gobmk	

Table 4: Symbolization false positives of 32-bit SPEC, REAL and COREUTILS (Others have zero false positive)

Benchmark	# of Ref.	Assumption Set									
		{}		{A1}		{A1, A2}		{A1, A3}		{A1, A2, A3}	
		FP	FP Rate	FP	FP Rate	FP	FP Rate	FP	FP Rate	FP	FP Rate
perlbench	76538	2	0.026‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
hmmer	13127	12	0.914‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
h264ref	20600	27	1.311‰	1	0.049‰	1	0.049‰	0	0.000‰	0	0.000‰
gcc	262698	49	0.187‰	32	0.122‰	32	0.122‰	0	0.000‰	0	0.000‰
gobmk	65244	1348	20.661‰	985	15.097‰	912	13.978‰	78	1.196‰	5	0.077‰

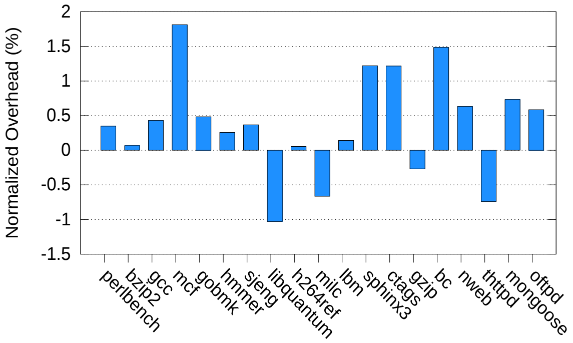


Figure 5: Execution overhead for REAL and SPEC programs relative to the original versions

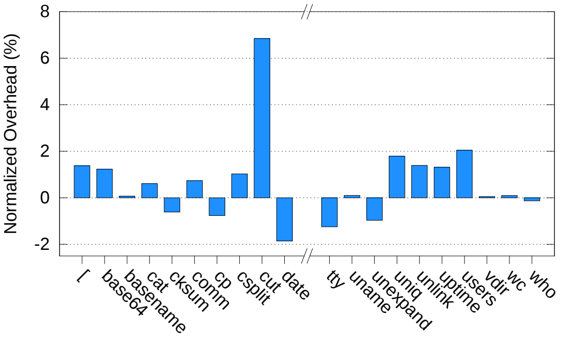


Figure 6: Execution overhead for COREUTILS programs relative to the original versions

6.2.1 Execution Overhead

Some programs in COREUTILS are not suitable for performance benchmarking, including `su`, `nohup`, and `timeout`, etc. After excluding these programs, we have 90 left to inspect in COREUTILS. The experiments are conducted on a machine with Intel Core i7-3770 3.40GHz and 8GB memory running Ubuntu 12.04.

We present the execution slowdown of reassembled binaries in Figure 5 and Figure 6. Since it is hard to present the data of all 90 binaries from COREUTILS, we sort COREUTILS programs by their names in alphabet order and plot the data for the first and last 10 programs in Figure 6. We report that the average slowdown for is 0.44% for COREUTILS, 0.29% for SPEC and 0.52% for REAL. The data suggests that UROBOROS does not have any significant impact on the execution speed of reassembled binaries.

6.2.2 Size Expansion

We use the `stat` program from GNU Coreutils to calculate file size expansion of the reassembled binaries compared to the originals. As the increase is generally negligible, we only report the average data here. The average expansion for COREUTILS is 0.83%, 0.00% for SPEC and -0.02% for REAL. Data shows that UROBOROS has almost zero impact on binary size when delivering reassembility. As aforementioned in §5.6, subsequent disassembly-reassembly iterations have zero expansion.

6.2.3 Processing Time

We measure how long it takes UROBOROS to disassemble binaries. Figure 7 presents the processing time for SPEC and REAL binaries. Figure 8 presents processing time for COREUTILS binaries selected using a same al-

Table 5: Symbolization false negatives of 32-bit SPEC, REAL and COREUTILS (Others have zero false negative)

Benchmark	# of Ref.	Assumption Set									
		{}		{A1}		{A1, A2}		{A1, A3}		{A1, A2, A3}	
		FN	FN Rate	FN	FN Rate	FN	FN Rate	FN	FN Rate	FN	FN Rate
perlbench	76538	2	0.026‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
hmmer	13127	12	0.914‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
h264ref	20600	27	1.311‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
gcc	262698	11	0.042‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
gobmk	65244	86	1.318‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰

Table 6: Symbolization false positives of 64-bit SPEC, REAL and COREUTILS (Others have zero false positive). Also, no false negatives are found for any binary.

Benchmark	# of Ref.	Assumption Set									
		{}		{A1}		{A1, A2}		{A1, A3}		{A1, A2, A3}	
		FP	FP Rate	FP	FP Rate	FP	FP Rate	FP	FP Rate	FP	FP Rate
perlbench	76952	32	0.416‰	10	0.130‰	10	0.130‰	0	0.000‰	0	0.000‰
gcc	259213	506	1.952‰	126	0.486‰	14	0.054‰	112	0.432‰	0	0.000‰
gobmk	65255	2437	37.346‰	1079	16.535‰	7	0.107‰	1073	16.443‰	1	0.015‰
hmmer	13165	11	0.836‰	2	0.152‰	0	0.000‰	2	0.152‰	0	0.000‰
sjeng	8837	22	2.490‰	2	0.226‰	0	0.000‰	2	0.226‰	0	0.000‰
h264ref	20264	15	0.740‰	1	0.049‰	0	0.000‰	1	0.049‰	0	0.000‰
lbm	248	1	4.032‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
sphinx3	8656	3	0.347‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
ctags	12997	2	0.154‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
gzip	3323	11	3.310‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
mongoose	3643	1	0.275‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
df	4202	1	0.238‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
du	4593	1	0.218‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
split	2851	1	0.351‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰
timeout	1935	1	0.517‰	0	0.000‰	0	0.000‰	0	0.000‰	0	0.000‰

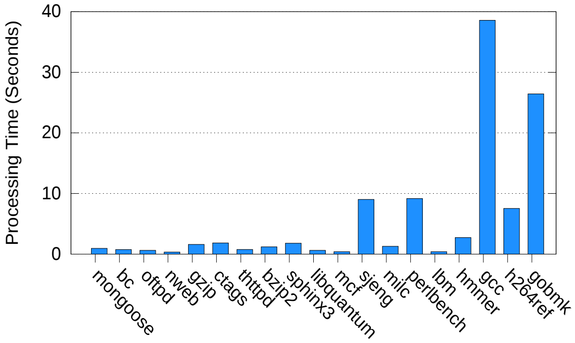


Figure 7: Processing time for SPEC and REAL binaries

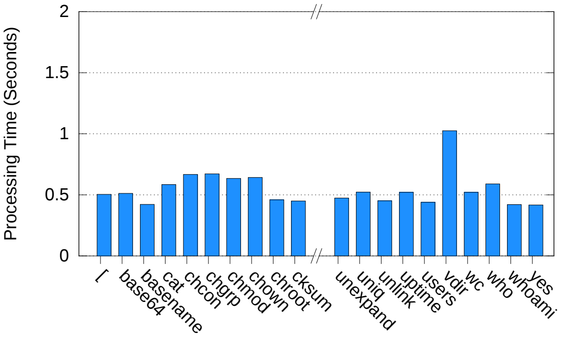


Figure 8: Processing time for COREUTILS binaries

phabet order strategy. As expected, larger binaries take more time to process. On average, UROBOROS spends 8.27 seconds on binaries from SPEC, 0.98 seconds on binaries from REAL, and 0.57 seconds on binaries from COREUTILS. We interpret this as a promising result, and the efficiency of UROBOROS makes it a tool totally practical for production deployment.

7 Discussions and Limitations

Compiler Compatibility. Sometimes binary reverse engineering is compiler dependent, but UROBOROS does not explicitly depend on any compiler-specific features as far as we know. To roughly investigate UROBOROS's compatibility with other compilers, we try to disassemble and reassemble some binaries compiled by Clang, another widely used compiler different from GCC. We only briefly present the results here due to limited space. We repeat the same functionality verification described in §6.1 on the 32-bit binaries in REAL, which are compiled by Clang this time. The applied assumption set in this experiment is the empty assumption set, and all reassembled binaries can pass the functionality tests. We plan to test UROBOROS's compatibility in more depth in the future.

C++ Binary Disassembly. The C++ programming language has more specific features compared with C. Binaries compiled from C++ programs often contain more sections to store meta-information. At this point UROBOROS still cannot fully support C++ disassembly, but we have already worked out a blueprint on how to recover these sections. There are two kinds of meta-information sections specific to C++. We now briefly discuss how to recover them.

- The `.ctors` and `.init_array` sections contain the addresses of constructor functions—functions that need to be executed at start up before the `main` function takes control. These sections can be directly dumped out and symbolized by treating them as data sections.
- The `.eh_frame` and `.gcc_except_table` sections store the information used for stack unwinding and exception handling for C++ programs in the DWARF format [15]. There have been some reverse engineering tools, e.g., Katana [38] and IDA Pro, that can parse the DWARF data. By understanding the semantics of a DWARF entry, we can adjust its content and make the reassembly flawless.

We leave fully supporting C++ binary disassembly as part of our future work.

Availability of function starting addresses. We assume the availability of the function starting addresses in the input binary, as in this research we would like to assess the assumptions and techniques we develop for the symbolization problem. Identifying function starting addresses is an orthogonal research issue which has been the focus of recent work [6, 41]. UROBOROS can leverage existing techniques to make the tool more practical. Nevertheless, this is currently a limitation of UROBOROS and we plan to investigate further in the future.

Data Section Relocation. By accepting the assumption A2 (see §4), we fix the starting address of data sections, which leads to certain limitations related to the relocation of data sections. However, data can still be manipulated as long as the starting addresses stay the same. Besides, `.bss` section can be extended with new elements, as it is at the end of typical ELF binaries and the increase of its size does not need to relocate other sections. In the future, it would be interesting to see whether some more sophisticated heuristics or analysis can be developed to symbolize d2d references.

Extensions. We believe that we have built an enabling technology that could be employed as the basis of many important research and applications, such as software fault isolation (SFI), control-flow integrity (CFI), ROP defense, and in general software retrofitting for binary code, which is extremely important for legacy code systems. Nevertheless, this is a first step in the toolchain development. We plan to build and maintain a sustainable ecosystem, and also link to the existing ecosystems such as LLVM [27] and CIL [36] by lifting assembly to LLVM and CIL IR.

8 Conclusion

We have presented UROBOROS, a tool that can disassemble stripped binaries and produce *reassembleable* assembly code in a fully automated manner. We call this technique *reassembleable disassembling* and have developed a prototype called UROBOROS. Our experiments show that reassembled programs incur negligible execution overhead, and thus UROBOROS can be potentially used as a foundation for binary-based software retrofitting.

9 Acknowledgments

We thank the USENIX Security anonymous reviewers and Stephen McCamant for their valuable feedback. This research was supported in part by the National Science Foundation (NSF) grants CNS-1223710 and CCF-1320605, and the Office of Naval Research (ONR) grant N00014-13-1-0175.

References

[1] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and Communications Security* (2005), ACM, pp. 340–353.

[2] ADL-TABATABAI, A.-R., LANGDALE, G., LUCCO, S., AND WAHBE, R. Efficient and language-independent mobile programs. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation* (1996), ACM, pp. 127–136.

[3] ANAND, K., SMITHSON, M., ELWAZEER, K., KOTHA, A., GRUEN, J., GILES, N., AND BARUA, R. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), ACM, pp. 295–308.

[4] Ansel, J., Marchenko, P., Erlingsson, U., Taylor, E., Chen, B., Schuff, D. L., Sehr, D., Biffle, C. L., and Yee, B. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (2011), pp. 355–366.

[5] Balakrishnan, G. *WYSINWYX: What You See is Not What You eXecute*. PhD thesis, University of Wisconsin-Madison, 2007.

[6] Bao, T., Burket, J., Woo, M., Turner, R., and Brumley, D. ByteWeight: Learning to recognize functions in binary code. In *In Proceedings of the 23rd USENIX Security Symposium* (2014), USENIX Association, pp. 845–860.

[7] Bruening, D. L. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[8] Brumley, D., Jager, I., Avgerinos, T., and Schwartz, E. J. BAP: A binary analysis platform. In *Computer Aided Verification* (2011), Springer, pp. 463–469.

[9] Buchanan, E., Roemer, R., Shacham, H., and Savage, S. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (2008), ACM, pp. 27–38.

[10] Buck, B., and Hollingsworth, J. K. An API for runtime code patching. *Int. J. High Perform. Comput. Appl. 14*, 4 (2000), 317–329.

[11] Caballero, J., Johnson, N. M., McCamant, S., and Song, D. Binary code extraction and interface identification for security applications. In *Proceedings of the Network and Distributed System Security Symposium* (2010), The Internet Society.

[12] Dagger. http://dagger.repzret.org/.

[13] De Sutter, B., De Bus, B., and De Bosschere, K. Link-time binary rewriting techniques for program compaction. *ACM Trans. Program. Lang. Syst. 27*, 5 (2005), 882–945.

[14] Deng, Z., Zhang, X., and Xu, D. BISTRO: Binary component extraction and embedding for software security applications. In *Proceedings of 18th European Symposium on Research in Computer Security* (2013), Springer, pp. 200–218.

[15] DWARF debugging information format, 1993. Proposed Standard, UNIX International Programming Languages Special Interest Group.

[16] Edwards, A., Vo, H., Srivastava, A., and Srivastava, A. Vulcan binary transformation in a distributed environment. Tech. Rep. MSR-TR-2001-50, Microsoft Research, 2001.

[17] Erlingsson, U., Abadi, M., Vrable, M., Budiu, M., and Necula, G. C. XFI: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (2006), USENIX Association, pp. 75–88.

[18] Ford, B., and Cox, R. Vx32: Lightweight user-level sandboxing on the x86. In *Proceedings of USENIX 2008 Annual Technical Conference* (2008), USENIX Association, pp. 293–306.

[19] Graham, S. L., Lucco, S., and Wahbe, R. Adaptable binary programs. In *Proceedings of the USENIX 1995 Technical Conference Proceedings* (1995), USENIX Association, pp. 26–26.

[20] Harris, L. C., and Miller, B. P. Practical analysis of stripped binary code. *SIGARCH Comput. Archit. News 33*, 5 (2005), 63–68.

[21] Hex-Rays Decompiler: Manual. https://www.hex-rays.com/products/decompiler/manual/failures.shtml.

[22] Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., and Davidson, J. W. ILR: Where'd my gadgets go? In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (2012), IEEE, pp. 571–585.

[23] Horspool, R. N., and Marovac, N. An approach to the problem of detranslation of computer programs. *Comput. J. 23*, 3 (1980), 223–229.

[24] The IDA Pro disassembler and debugger. https://www.hex-rays.com/idapro.

[25] Introduction to x64 Assembly. https://software.intel.com/en-us/articles/introduction-to-x64-assembly/.

[26] Kolbitsch, C., Holz, T., Kruegel, C., and Kirda, E. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy* (2010), IEEE, pp. 29–44.

[27] Lattner, C. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, 2005.

[28] Lee, J., Avgerinos, T., and Brumley, D. TIE: Principled reverse engineering of types in binary programs. In *Proceedings of the Network and Distributed System Security Symposium* (2011), The Internet Society.

[29] Li, J., Wang, Z., Jiang, X., Grace, M., and Bahram, S. Defeating return-oriented rootkits with "return-less" kernels. In *Proceedings of the 5th European Conference on Computer Systems* (2010), ACM, pp. 195–208.

[30] LLVM 3.1 release notes. http://llvm.org/releases/3.1/docs/ReleaseNotes.html, 2012.

[31] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (2005), ACM, pp. 190–200.

[32] MC-Semantics. https://github.com/trailofbits/mcsema.

[33] McCamant, S., and Morrisett, G. Evaluating SFI for a CISC architecture. In *Proceedings of the 15th Conference on USENIX Security Symposium* (2006), USENIX Association.

[34] Ming, J., Wu, D., Xiao, G., Wang, J., and Liu, P. TaintPipe: Pipelined symbolic taint analysis. In *Proceedings of the 24th USENIX Security Symposium* (2015), USENIX Association.

[35] Muth, R., Debray, S. K., Watterson, S., and De Bosschere, K. Alto: A link-time optimizer for the Compaq Alpha. *Softw. Pract. Exper. 31*, 1 (2001), 67–101.

[36] Necula, G. C., McPeak, S., Rahul, S. P., and Weimer, W. CIL: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction* (2002), Springer, pp. 213–228.

[37] Niu, B., and Tan, G. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014), ACM, pp. 577–587.

[38] Oakley, J., and Bratus, S. Exploiting the hard-working DWARF: Trojan and exploit techniques with no native executable code. In *Proceedings of the 5th USENIX Conference on Offensive Technologies* (2011), USENIX Association, pp. 11–11.

[39] Pappas, V., Polychronakis, M., and Keromytis, A. D. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (2012), IEEE.

[40] ROPgadget tool. http://shell-storm.org/project/ROPgadget.

[41] Rosenblum, N., Zhu, X., Miller, B., and Hunt, K. Learning to analyze binary computer code. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2* (2008), AAAI, pp. 798–804.

[42] Schwartz, E. J., Lee, J., Woo, M., and Brumley, D. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the 22nd USENIX Security Symposium* (2013), USENIX Association, pp. 353–368.

[43] Sehr, D., Muth, R., Biffle, C., Khimenko, V., Pasko, E., Schimpf, K., Yee, B., and Chen, B. Adapting software fault isolation to contemporary CPU architectures. In *Proceedings of the 19th USENIX Conference on Security* (2010), USENIX Association, pp. 1–11.

[44] Shacham, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and Communications Security* (2007), ACM, pp. 552–561.

[45] Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M. G., Liang, Z., Newsome, J., Poosankam, P., and Saxena, P. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security* (Berlin, Heidelberg, 2008), ICISS '08, Springer-Verlag, pp. 1–25.

[46] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient software-based fault isolation. *SIGOPS Oper. Syst. Rev. 27*, 5 (1993), 203–216.

[47] Wartell, R., Mohan, V., Hamlen, K. W., and Lin, Z. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM conference on Computer and Communications Security* (2012), ACM, pp. 157–168.

[48] Zeng, J., Fu, Y., Miller, K. A., Lin, Z., Zhang, X., and Xu, D. Obfuscation resilient binary code reuse through trace-oriented programming. In *Proceedings of the 2013 ACM Conference on Computer and Communications Security* (2013), ACM, pp. 487–498.

[49] Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., and Zou, W. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (2013), IEEE, pp. 559–573.

[50] Zhang, M., and Sekar, R. Control flow integrity for COTS binaries. In *Proceedings of the 22Nd USENIX Conference on Security* (2013), USENIX Association, pp. 337–352.

A Symbolization Errors in gobmk

Among the 244 binaries from the Coreutils, Real, and Spec collections, we found 5 d2c reference symbolization errors in `gobmk`, one of the largest Spec programs, 4 of which are in the 32-bit version and 1 of which is in the 64-bit version, all false positives.

The software `gobmk` is GNU Go (http://www.gnu.org/software/gnugo/), a program for playing the board game of Go. The program contains a fairly large database of board configuration patterns. In order to speed up pattern matching, it builds Deterministic Finite Automata (DFA) from the pattern database.

The five reference symbolization errors are shown in Table 7 and Figure 9. In Figure 9, the two arrays `state_owl_attackpat` and `state_owl_defendpat` encode two DFAs with 24,701 and 34,044 entries, respectively. Each entry represents a state in the corresponding DFA. Each state has 5 numbers of the C short datatype, the current state and its four neighbors, as Go games are played on 2-dimensional grid boards.

We found two consecutive states in DFA `state_owl_attackpat` of the 64-bit `gobmk` are `{66,{0,0,0,0}},{70,{0,0,0,0}}`. The two C short int numbers 0 and 70 in the middle forms 0x460000 (little-endian), which happens to be the starting address of function `gtp_trymove`. Similar patterns exist in the 32-bit `gobmk`. States in array `state_owl_defendpat` forms value 0x080c0000 from two C short int numbers 0 and 2060 next to each other (little-endian), which collides with the starting address of function `autohelperpat1029`.

Table 7: Source code locations of symbolization errors

Program	Location (file and line no.)
32-bit gobmk	`owl_defendpat.c: 9688`
	`owl_defendpat.c: 9702`
	`owl_defendpat.c: 9703`
	`owl_defendpat.c: 9704`
	`owl_defendpat.c: 9761`
64-bit gobmk	`owl_attackpat.c: 5828`

```
static const state_rt_t
    state_owl_defendpat[34044] = {
  ...
  {0,{2060,2061,2062,2063}}, ...
  {0,{2060,2060,2060,2060}}, ...
  {0,{2060,2060,2060,2060}}, ...
  {0,{2060,2060,2060,2060}}, ...
  {0,{2060,2060,2061,2060}}, ...
};
```

(a) 32-bit `gobmk`

```
static const state_rt_t
    state_owl_attackpat[24701] = {
  ...
  {66,{0,0,0,0}}, {70,{0,0,0,0}}, ...
};
```

(b) 64-bit `gobmk`

Figure 9: Source code of symbolization errors

How the ELF Ruined Christmas

Alessandro Di Federico[1,2], Amat Cama[1], Yan Shoshitaishvili[1], Christopher Kruegel[1], and Giovanni Vigna[1]

[1]University of California, Santa Barbara, CA, USA
{amat,yans,chris,vigna}@cs.ucsb.edu
[2]Politecnico di Milano, Milan, Italy
alessandro.difederico@mail.polimi.it

Abstract

Throughout the last few decades, computer software has experienced an arms race between exploitation techniques leveraging memory corruption and detection/protection mechanisms. Effective mitigation techniques, such as Address Space Layout Randomization, have significantly increased the difficulty of successfully exploiting a vulnerability. A modern exploit is often two-stage: a first information disclosure step to identify the memory layout, and a second step with the actual exploit. However, because of the wide range of conditions under which memory corruption occurs, retrieving memory layout information from the program is not always possible.

In this paper, we present a technique that uses the dynamic loader's ability to *identify* the locations of critical functions directly and call them, without requiring an information leak. We identified several fundamental weak points in the design of ELF standard and dynamic loader implementations that can be exploited to resolve and execute arbitrary library functions. Through these, we are able to bypass specific security mitigation techniques, including partial and full RELRO, which are specifically designed to protect ELF data-structures from being coopted by attackers. We implemented a prototype tool, Leakless, and evaluated it against different dynamic loader implementations, previous attack techniques, and real-life case studies to determine the impact of our findings. Among other implications, Leakless provides attackers with reliable and non-invasive attacks, less likely to trigger intrusion detection systems.

1 Introduction

Since the first widely-exploited buffer overflow used by the 1998 Morris worm [27], the prevention, exploitation, and mitigation of memory corruption vulnerabilities have occupied the time of security researchers and cybercriminals alike. Even though the prevalence of memory corruption vulnerabilities has finally begun to decrease in recent years, classic buffer overflows remain the third most common form of software vulnerability, and four other memory corruption vulnerabilities pad out the top 25 [13].

One reason behind the decreased prevalence of memory corruption vulnerabilities is the heavy investment in research on their prevention and mitigation. Specifically, many mitigation techniques have been adopted in two main areas: system-level hardening (such as CGroups [23], AppArmor [4], Capsicum [41], and GRSecurity [18]) and application-level hardening (such as stack canaries [3], Address Space Layout Randomization (ASLR), and the *No-eXecute* (NX) bit [8]).

In particular, *Address Space Layout Randomization* (ASLR), by placing the dynamic libraries in a random location in memory (unknown to the attacker), lead attackers to perform exploits in two stages. In the first stage, the attacker must use an *information disclosure* vulnerability, in which information about the memory layout of the application (and its libraries) is revealed, to identify the address of code that represents security-critical functionality (such as the `system()` library function). In the second stage, the attacker uses a *control flow redirection* vulnerability to redirect the program's control flow to this functionality.

However, because of the wide range of conditions under which memory corruptions occur, retrieving this information from the program is not always possible. For example, memory corruption vulnerabilities in parsing code (e.g., decoding images and video) often take place without a direct line of communication to an attacker, precluding the possibility of an information disclosure. Without this information, performing an exploit against ASLR-protected binaries using current techniques is often infeasible or unreliable.

As noted in [36], despite the race to harden applications and systems, the security of some little-known aspects of application binary formats and the system components using them, have not received much scrutiny. In particular we focus on the *dynamic loader*, a userspace component of the operating system, responsible for loading binaries, and the libraries they depend upon, into memory. Binaries use the dynamic loader to support the *resolution* of imported symbols. Interestingly, this is the exact behavior that an attacker of a hardened application attempts to reinvent by leaking a library's address and contents.

Our insight is that a technique to eliminate the need for an information disclosure vulnerability could be developed by abusing the functionality of the dynamic loader. Our technique leverages weaknesses in the dynamic loader and in the general design of the ELF format to resolve and execute arbitrary library functions, allowing us to successfully exploit hardened applications without the need for an information disclosure vulnerability. Any library function can be executed with this technique, even if it is not otherwise used by the exploited binary, as long as the library that it resides in is loaded. Since almost every binary depends on the C Library, this means our technique allows us to execute security-critical functions such as `system()` and `execve()`, allowing arbitrary command execution. We will also show application-specific library functions can be re-used to perform sophisticated and stealthy attacks. The presented technique is reliable, architecture-agnostic, and does not require the attacker to know the version, layout, content, or any other unavailable information about the library and library function in question.

We implemented our ideas in a prototype tool, called Leakless[1]. To use Leakless, the attacker must possess the target application, and have the ability to exploit the vulnerability (i.e., hijack control flow). Given this information, Leakless can automatically construct an exploit that, without the requirement of an information disclosure, invokes one or more critical library functions of interest.

To evaluate our technique's impact, we performed a survey of several different distributions of Linux (and FreeBSD) and identified that the vast majority of binaries in the default installation of these distributions are susceptible to the attack carried out by Leakless, if a memory corruption vulnerability is present in the target binary. We also investigated the dynamic loader implementations of various C Libraries, and found that most of them are susceptible to Leakless' techniques. Additionally, we showed that a popular mitigation technique, RELocation Read-Only (RELRO), which protects library function calls from being redirected by an attacker, is completely bypassable by Leakless. Finally, we compared the length of Leakless' ROP chains against ROP compilers implementing similar functionality. Leakless produces significantly shorter ROP chains than existing techniques, which, as we show, allows it to be used along with a wider variety of exploits than similar attacks created by traditional ROP compilers.

In summary, we make the following contributions:

- We develop a new, architecture- and platform-agnostic attack, using functionality inherent in ELF-based system that supports dynamic loading, to enable an attacker to execute arbitrary library functions without an information disclosure vulnerability.
- We detail, and overcome, the challenges of implementing our system for different dynamic loader implementations and in the presence of multiple mitigation techniques (including RELRO).
- Finally, we perform an in-depth evaluation, including a case study of previously complicated exploits that are made more manageable with our technique, an assessment of the security of several different dynamic loader implementations, a survey of the applicability of our technique to different operating system configurations, and a measurement of the improvement in the length of ROP chains produced by Leakless.

2 Related Work: The Memory Corruption Arms Race

The memory corruption arms race (i.e., the process of defenders developing countermeasures against known exploit techniques, and attackers coming up with new exploitation techniques to bypass these countermeasures) has been ongoing for several decades. While the history of this race has been documented elsewhere [37], this section focuses on the sequence of events that has required many modern exploits to be *two-stage*, that is, needing an *information disclosure* step before an attacker can achieve arbitrary code execution.

Early buffer overflow exploits relied on the ability to inject binary code (termed *shellcode*) into a buffer, and overwrite a return address on the stack to point into this buffer. Subsequently, when the program would return from its current function, execution would be redirected to the attacker's shellcode, and the attacker would gain control of the program.

As a result, security researchers introduced another mitigation technique: the *NX* bit. The NX bit has the effect of preventing memory areas not supposed to contain code (typically, the stack) from being executed.

The *NX bit* has pushed attackers to adapt the concept of *code reuse*: using functionality already in the program (such as system calls and security-critical library functions) to accomplish their goals. In return-into-libc exploits [30, 39], an attacker redirects the control flow directly to a sensitive libc function (such as `system()`) with the proper arguments to perform malicious behavior, instead of using injected shellcode.

To combat this technique, a system-level hardening technique named *Address Space Layout Randomization*

[1] The source code is available at: `https://github.com/ucsb-seclab/leakless`

(ASLR) was developed. When ASLR is in place, the attacker does not know the location of libraries, in fact, the program's memory layout (the locations of libraries, the stack, and the heap) is randomized at each execution. Because of this, the attacker does not know *where* in the library to redirect the control flow in order to execute specific functions. Worse, even if the attacker is able to determine this information, he is still unable to identify the location of specific functions inside the library unless he is in possession of a copy of the library. As a result, an attacker usually has to leak the contents of the library itself and parse the code to identify the location of critical functions. To leak these libraries, attackers often reuse small chunks of code (called *gadgets*) in the program's code segment to disclose memory locations. These gadgets are usually combined by writing their addresses onto the stack and consecutively returning to them. Thus, this technique is named *Return Oriented Programming* (ROP) [35].

ROP is a powerful tool for attackers. In fact, it has been shown that a "Turing-complete" set of ROP gadgets can be found in many binaries and can be employed, with the help of a *ROP compiler*, to carry out exploitation tasks [34]. However, because of their generality, ROP compilers tend to produce long ROP chains that, depending on the specific details of a vulnerability, are "too big to be useful" [22]. Later, we will show that Leakless produces relatively short ROP chains, and, depending on present mitigations, requires very few gadgets. Additionally, Leakless is able to function without a Turing-complete gadget set.

In real-world exploits, an attacker usually uses an *information disclosure* attack to leak the address or contents of a library, then uses this information to calculate the correct address of a security-critical library function (such as `system()`), and finally sends a second payload to the vulnerable application that redirects the control flow to call the desired function.

In fact, we observed that that the goal of finding the address of a specific library function is actually already implemented by the *dynamic loader*, an OS component that facilitates the resolution of dynamic symbols (i.e., determining the addresses of library functions). Thus, we realized that we could leverage the dynamic loader to remove the information disclosure step, and craft exploits, which would work without the need of an information disclosure attack. Since our attack does not require an information leak step, we call it *Leakless*.

The concept of using the dynamic loader as part of the exploitation process was briefly explored in the context of return-into-libc attacks [15,21,30]. However, existing techniques are extremely situational [30], platform-dependent, require two stages [21], or are susceptible to current mitigation techniques such as RELRO [30], which we will discuss in future sections. Leakless, on the other hand, is a single-stage, platform-independent, general technique, and is able to function in the presence of such mitigations.

In the next section, we will describe how the dynamic loader works, and afterwards will show how we abuse this functionality to perform our attack.

3 The Dynamic Loader

The dynamic loader is a component of the userspace execution environment that facilitates loading the libraries required by an application at start time and resolving the dynamic symbols (functions or global variables) that are exported by libraries and used by the application. In this section, we will describe how dynamic symbol resolution works on systems based on the ELF binary object specification [33].

ELF is a standard format common to several Unix-like platforms, including GNU/Linux and FreeBSD, and is defined independently from any particular dynamic loader implementation. Since Leakless mostly relies on standard ELF features, it is easily applicable to a wide range of systems.

3.1 The ELF Object

An application comprises a main binary ELF file (the executable) and several dynamic libraries, also in ELF format. Each ELF object is composed of *segments*, and each segment holds one or more *sections*.

Each section has a conventional meaning. For instance, the `.text` section contains the code of the program, the `.data` section contains its writeable data (such as global variables), and the `.rodata` section contains the read-only data (such as constants and strings). The list of sections is stored in the ELF file as an array of `Elf_Shdr` structures.

Note that there are two versions of each ELF structure: one version for 32-bit ELF binaries (e.g., `Elf32_Rel`) and one for 64-bit (e.g., `Elf64_Rel`). We ignore this detail for the sake of simplicity, except in specific cases where it is relevant to our discussion.

3.2 Dynamic Symbols and Relocations

In this section, we will give a summary of the data structures involved in ELF symbol resolution. Figure 1 gives an overview of these data structures and their mutual relationships.

An ELF object can export symbols to and import symbols from other ELF objects. A symbol represents a function or a global variable and is identified by a name.

Each symbol is described by a corresponding `Elf_Sym` structure. This structure, instances of which comprise the `.dynsym` ELF section, contains the following fields relevant to our work:

`st_name`. An offset, relative to the start of the `.dynstr` section, where the string containing the name of the symbol is located.

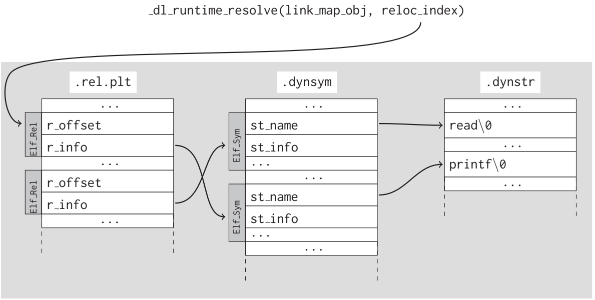


Figure 1: The relationship between data structures involved in symbol resolution (without symbol versioning). Shaded background means read only memory.

st_value. If the symbol is exported, the virtual address of the exported function, NULL otherwise.

These structures are referenced to resolve imported symbols. The resolution of imported symbols is supported by relocations, described by the Elf_Rel structure. Instances of this structure populate the .rel.plt section (for imported functions) and the .rel.dyn section (for imported global variables). In our discussion we are only interested to the former section. The Elf_Rel structure has the following fields:

r_info. The three least significant bytes of this field are used as an unsigned index into the .dynsym section to reference a symbol.

r_offset. The location (as an absolute address) in memory where the address of the resolved symbol should be written to.

When a program imports a certain function, the linker will include a string with the function's name in the .dynstr section, a symbol (Elf_Sym) that refers to it in the .dynsym section, and a relocation (Elf_Rel) pointing to that symbol in the .rel.plt section.

The target of the relocation (the r_offset field of the Elf_Rel struct) will be the address of an entry in a dedicated table: the Global Offset Table (GOT). This table, which is stored in the .got.plt section, is populated by the dynamic loader as it resolves the relocations in the .rel.plt section.

3.3 Lazy Symbol Resolution

Since resolving every imported symbol and applying all relocations at application startup can be a costly operation, symbols are resolved *lazily*. In lazy symbol resolution, the address of a function (which corresponds to an entry in the GOT) is only resolved when necessary (i.e., the first time the imported function is called).

When a program wants to calls an imported function, it instead calls a dedicated stub of code, located in the Procedure Linkage Table (the .plt section). As shown in Listing 1, each imported function has a stub in the PLT that performs an unconditional indirect jump to the associated entry in the GOT.

After symbol resolution, this GOT entry contains the address of the actual function, in the imported library, and execution continues seamlessly into this function. When the function returns, control flow returns to the *caller* of the PLT stub, and the rest of the PLT stub is not executed. However, at program startup, GOT entries are initialized with an address pointing to the *second* instruction of the associated PLT stub. This part of the stub will push onto the stack an identifier of the imported function (in the form of an offset to an Elf_Rel instance in the .rel.plt section) and jump to the PLT0 stub, a piece of code at the beginning of the .plt section. In turn, the PLT0 stub, pushes the value of GOT[1] onto the stack and performs an indirect jump to the address of GOT[2]. These two entries in the GOT have a special meaning and the dynamic loader populates them at application startup:

GOT[1]. A pointer to an internal data structure, of type link_map, which is used internally by the dynamic loader and contains information about the current ELF object needed to carry out symbol resolution.

GOT[2]. A pointer to a function of the dynamic loader, called _dl_runtime_resolve.

In summary, PLT entries basically perform the following function call:

```
_dl_runtime_resolve(link_map_obj, reloc_index)
```

This function uses the link_map_obj parameter to access the information it needs to resolve the desired imported function (identified by the reloc_index argument) and writes the result into the appropriate GOT entry. After _dl_runtime_resolve resolves the imported function, control flow is passed to that function, making the resolution process completely transparent to the caller. The next time the PLT stub for the specified function is invoked execution will be diverted directly to the target function.

Listing 1: Example PLT and GOT.

```
100 PLT0:                    196 ; .plt.got start
100     push *0x200          196 ; Empty entry
106     jmp *0x204           196 0
110 printf@plt:              200 ; link_map object
110     jmp *0x208           200 &link_map_obj
116     push #0              204 ; Resolver function
11B     jmp PLT0             204 &_dl_runtime_resolve
120 read@plt:                208 ; printf entry
120     jmp *0x20C           208 0x116
126     push #1              20C ; read entry
12B     jmp PLT0             20C 0x126
```

The link_map structure contains all the information that the dynamic loader needs about a loaded ELF object. Each link_map instance is an entry in a doubly-linked list containing the information about all loaded ELF objects.

3.4 Symbol Versioning

The ELF standard provides a mechanism to import a symbol with a specific version associated with it. This feature is used to require a function to be imported from a

Table 1: Entries of the .dynamic section. d_tag is the key, while d_value is the value.

d_tag	d_value	d_tag	d_value
DT_SYMTAB	.dynsym	DT_PLTGOT	.got.plt
DT_STRTAB	.dynstr	DT_VERNEED	.gnu.version
DT_JMPREL	.rel.plt	DT_VERSYM	.gnu.version_r

specific version of a library. For instance, it is possible to require the fopen C Standard Library function, as implemented in version 2.2.5 of the GNU C Standard Library, using the version identifier GLIBC_2.2.5. The .gnu.version_r section contains version definitions in the form of Elf_Verdef structures.

The association between a dynamic symbol and the Elf_Verdef structure that it refers to is kept in the .gnu.version section, as an array of Elf_Verneed structures, one for each entry in the dynamic symbol table. These structures have a single field: a 16-bit integer that represents an index into the .gnu.version_r section.

Due to this layout, the index in the r_info field of the Elf_Rel structure is used by the dynamic loader as an index into both the .dynsym and .gnu.version sections. This is important to understand, as Leakless will later leverage this fact.

3.5 The .dynamic section and RELRO

The dynamic loader collects all the information that it needs about the ELF object from the .dynamic section, which is composed of Elf_Dyn structures. An Elf_Dyn is a key-value pair that stores different types of information. The relevant entries of this section, shown in Table 1, hold the absolute addresses of specific sections. One exception is the DT_DEBUG entry, which holds a pointer to an internal data structure of the dynamic loader. This is initialized by the dynamic loader and is used for debugging purposes.

An attacker able to tamper with these values can pose a security risk. For this reason, a protection mechanism known as RELRO (RELocation Read Only) has been introduced in dynamic loaders. RELRO comes in two flavors: partial and full.

Partial RELRO In this mode, some sections, including .dynamic, are marked as read-only after they have been initialized by the dynamic loader.

Full RELRO In addition to partial RELRO, lazy resolution is disabled: all import symbols are resolved at startup time, and the .got.plt section is completely initialized with the final addresses of the target functions and marked read-only. Moreover, since lazy resolution is not enabled, the GOT[1] and GOT[2] entries are not initialized with the values we mentioned in Section 3.3.

As we will see, RELRO poses significant complications that Leakless must (and does) address in order to operate in the presence of these countermeasures.

Note that the previously mentioned link_map structure stores in the l_info field an array of pointers to most of entries in the .dynamic section for internal usage. Since the dynamic loader trusts the content of this field implicitly, Leakless will later be able to misuse this to its own ends.

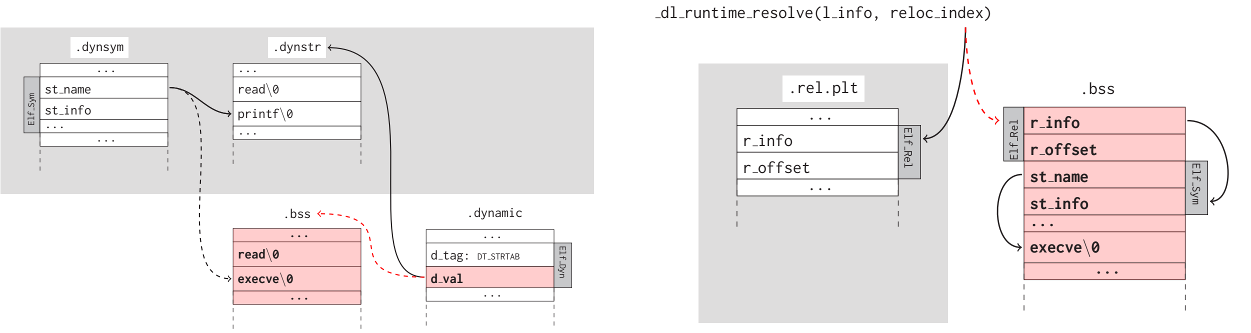
4 The Attack

Leakless enables an attacker to call arbitrary library functions, using only their name, without any information about the memory layout of the vulnerable program's libraries. To achieve this, Leakless abuses the dynamic loader, forcing it to resolve and call the requested function. This is possible for the same reason that memory corruption vulnerabilities are so damaging: the mixing of control data and non-control data in memory. In the case of a stack overflow, the control data in question is a stored return address. For the dynamic loader, the control data is comprised of the various data structures that the dynamic loader uses for symbol resolution. Specifically, the *name* of the function, stored in the .dynstr section, is analogous to a return address: it specifies a specific target to execute when the function is invoked.

The dynamic loader makes the assumption that the parameters it receives and its internal structures are trustworthy because it assumes that they are provided directly by the ELF file or by itself during initialization. However, when an attacker is able to modify this data, the assumption is broken. Some dynamic loaders (FreeBSD) validate the input they receive. However, they still implicitly trust the control structures, which will be readily corrupted by Leakless.

Leakless is designed to be used by an attacker who is attempting to exploit an existing vulnerability. The input to Leakless is comprised of the executable ELF file, a set of ROP gadgets of the binary (we detail what gadgets an attacker needs in Section 5.1), and the name of a library function that the attacker wishes to call (typically, but not necessarily, execve()). Given this information, Leakless outputs a ROP payload that executes the needed library function, bypassing various hardening techniques applied to the binary in question. This ROP chain is generally very short: depending on the mitigations present in the binary, the chain is 3 to 12 write operations. Some examples of the output produced by Leakless are available in the documentation of the Leakless code repository [17].

Leakless does *not* require any information about the addresses or contents of the libraries; we assume that ASLR is enabled for all dynamic libraries and that no knowledge about them is available. However we also assume that the executable is not position-independent, and, thus, is always loaded in a specific location in memory. We discuss this limitation in detail in Section 7.2, and show



(a) Example of the attack presented in Section 4.1. The attacker is able to overwrite the value of the DT_STRTAB dynamic entry, tricking the dynamic loader into thinking that the .dynstr section is in .bss, where he crafted a fake string table. When the dynamic loader will try to resolve the symbol for printf it will use a different base to reach the name of the function and will actually resolve (and call) execve.

(b) Example of the attack presented in Section 4.2. The reloc_index passed to _dl_runtime_resolve overflows the .rel.plt section and ends up in .bss, where the attacker crafted an Elf_Rel structure. The relocation points to an Elf_Sym located right afterwards overflowing the .dynsym section. In turn the symbol will contain an offset relative to .dynstr large enough to reach the memory area after the symbol, which contains the name of the function to invoke.

Figure 2: Illustration of some of the presented attacks. Shaded background means read only memory, white background means writeable memory and bold or red means data crafted by the attacker.

how infrequently Position Independent Executables (PIE) binaries occur in modern OS distributions in Section 6.2.

While in most cases, Leakless works independently of the dynamic loader implementation and version that the target system is running, some of our attacks require minor modifications to accommodate different dynamic loaders.

Note that Leakless's aim, obtaining the address of a library function and call it, is similar to what the dlsym function of libdl does. However, in practice this function is rarely used by applications and, therefore, its address is not generally known to the attacker.

4.1 The Base Case

As explained in Section 3 and illustrated in Figure 1, the dynamic loader starts its work from a Elf_Rel structure in the .rel.plt, then follows the index into the .dynsym section to locate the Elf_Sym structure, and finally uses that to identify the name (a string in the .dynstr section) of the symbol to resolve. The simplest way to call an arbitrary function would be to overwrite the string table entry of an existing symbol with the name of the desired function, and then invoke the dynamic loader, but this is not possible, as the section containing the string table for dynamic symbols, i.e., .dynstr, is not writeable.

However, the dynamic loader obtains the address of the .dynstr section from the DT_STRTAB entry of the .dynamic section, which is at a known location and, by default, writeable. Therefore, as shown in Figure 2a, it is possible to overwrite the d_val field of this dynamic entry with a pointer to a memory area under the control of the attacker (typically the .bss or .data section). This memory area would then include a single string, for example execve. At this point, the attacker needs to choose an existing symbol pointing to the correct offset in the fake string table and invoke the resolution of relocation corresponding to that symbol. This can be done by pushing the offset of this relocation on the stack and then jumping to PLT0.

This approach is simple, but it is only effective against binaries in which the .dynamic section is writeable. More sophisticated attacks must be used against binary compiled with partial or full RELRO.

4.2 Bypassing Partial RELRO

As we explained in Section 3.3, the second parameter of the _dl_runtime_resolve function is the offset of an Elf_Rel entry in the relocation table (.rel.plt section) that corresponds to the requested function. The dynamic loader takes this value and adds it to the base address of the .rel.plt to obtain the absolute address of the target Elf_Rel structure. However most dynamic loader implementations do not check the boundaries of the relocation table. This means that if a value larger than the size of the .rel.plt is passed to _dl_runtime_resolve, the loader will use the Elf_Rel at the specified location, despite being outside the .rel.plt section.

As shown in Figure 2b, Leakless computes an index that will lead _dl_runtime_resolve to look into a memory area under the control of the attacker. It then crafts an Elf_Rel structure that contains, in its r_offset field, the address of the writeable memory location where the address of the function will be written. The r_info field will, in turn, contain an index that causes the dynamic loader to look into the attacker-controlled memory. Leakless stores a crafted Elf_Sym object at this location, which, likewise, holds a st_name field value large enough to point into attacker-controlled memory. Finally, this location is where Leakless stores the name of the desired function to call.

In sum, Leakless crafts the full chain of structures involved in symbol resolution, co-opting the process to invoke the function whose name Leakless has written into attacker-controlled memory. After this, Leakless pushes the computed offset to the fake Elf_Rel structure onto the stack and invokes PLT0.

However, this approach is subject to several constraints. First, the symbol index in Elf_Rel has to be positive, since the r_info field is defined by the ELF standard as an unsigned integer. In practice, this means that the writable memory area (e.g., the .bss section) must be located *after* the .dynsym section. In our evaluation, this has always been the case.

Another constraint arises when the ELF makes use of the symbol versioning system described in Section 3.4. In this case, the Elf_Rel.r_info field is not just used as an index into the dynamic symbol table, but also as an index in the symbol version table (the .gnu.version section). In general, Leakless is able to automatically satisfy these constraints, except for x86-64 small binaries using huge pages [32]. We detail the additional constraints introduced by symbol versioning in Appendix A. When the constraints cannot be satisfied, an alternate approach must be adopted. This involves abusing the dynamic loader by corrupting its internal data structures to alter the dynamic resolution process.

4.3 Corrupting Dynamic Loader Data

We recall that the first parameter to _dl_runtime_resolve is a pointer to a data structure of type link_map. This structure contains information about the ELF executable, and the contents of this structure are implicitly trusted by the dynamic loader. Furthermore, Leakless can obtain the address of this structure from the second entry of the GOT of the vulnerable binary, whose location is deterministically known.

Recall from Section 3.5 that the link_map structure, in the l_info field, contains an array of pointers to the entries of the .dynamic section. These are the pointers that the dynamic loader uses to locate the objects that are used during symbol resolution. As shown in Figure 3, by overwriting part of this data structure, Leakless can make the DT_STRTAB entry of the l_info field point to a specially-crafted dynamic entry which, in turn, points to a fake dynamic string table. Hence, the attacker can reduce the situation back to the base case presented in Section 4.1.

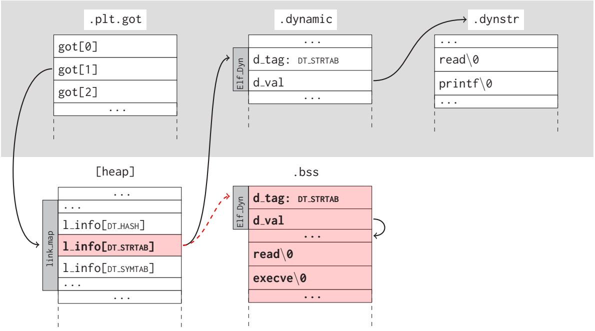


Figure 3: Example of the attack presented in Section 4.3. The attacker dereferences the second entry of the GOT and reaches the link_map structure. In this structure he corrupts the entry of the l_info field holding a pointer to the DT_STRTAB entry in the dynamic table. Its value is set to the address of a fake dynamic entry which, in turn, points to a fake dynamic string table in the .bss section.

This technique has wider applicability than the one presented in the previous section, since there are no specific constraints, and, in particular, it is applicable also against small 64 bit ELF binaries using huge pages. However, while in the previous attacks we were relying exclusively on standard ELF features, in this case (and in the one presented in the next section) we assume the layout of a glibc-specific structure (link_map) to be known. Each dynamic loader implements this structure in its own way, so minor modifications might be required when targeting a different dynamic loader. Note that link_map's layout might change among versions of the same dynamic loader. However, they tend to be quite stable, and, in particular, in glibc no changes relevant to our attack have taken place since 2004.

4.4 The Full RELRO Situation

Leakless is able to bypass full RELRO protection.

When full RELRO is applied, all the relocations are resolved at load-time, no lazy resolving takes place, and the addresses of the link_map structure and of _dl_runtime_resolve in the GOT are never initialized. Thus, it is not directly possible to know their addresses, which is what the general technique to bypass partial RELRO relies upon.

However, it is possible to indirectly recover these two values through the DT_DEBUG entry in the dynamic table. The value of the DT_DEBUG entry is set by the dynamic loader at load-time to point to a data structure of type r_debug. This data structure contains information used by debuggers to identify the base address of the dynamic loader and to intercept certain events related to dynamic loading. In addition, the r_map field of this structure holds a pointer to the head of the linked list of link_map structures.

Leakless corrupts the first entry of the list describing the ELF executable so that the l_info entry for DT_STRTAB points to a fake dynamic string table. This is presented in

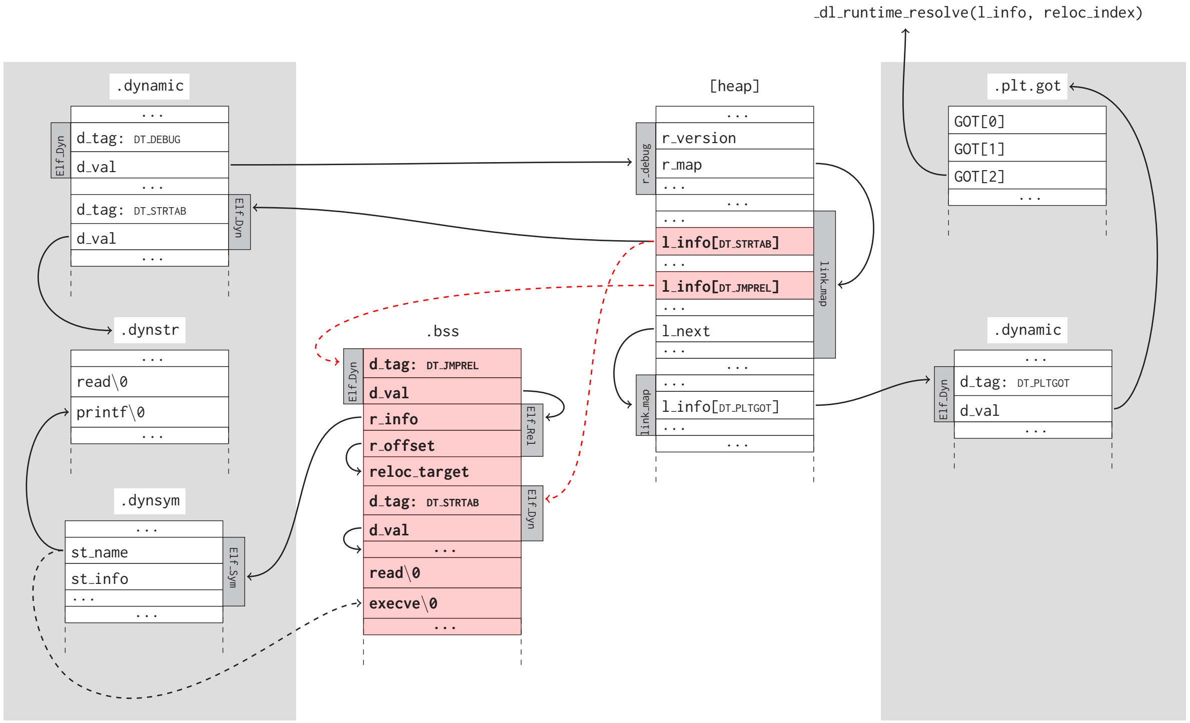


Figure 4: Example of the attack presented in Section 4.4. Shaded background means read only memory, white background means writeable memory and bold or red means data crafted by the attacker. The attacker goes through the DT_DEBUG dynamic entry to reach the r_debug structure, then, dereferencing the r_map field, he gets to the link_map structure of the main executable, and corrupts l_info[DT_STRTAB] as already seen in Section 3.
Since the .got.plt section is read-only due to full RELRO, the attacker also have to forge a relocation. To do so, he corrupts l_info[DT_JMPREL] making it point to a fake dynamic entry in turn pointing to a relocation. This relocation refers to the existing printf symbol, but has an r_offset pointing to a writeable memory area.
Then the attacker also needs to recover the pointer to the _dl_runtime_resolve function, which is not available in the GOT of the main executable due to full RELRO, therefore he dereferences the l_info field of the first link_map structure and gets to the one describing the first shared library, which is not protected by full RELRO. The attacker accesses the l_info[DT_PLTGOT] field and gets to the corresponding dynamic entry (the .dynamic on the right), and then to the .plt.got section (always on the right), at the second entry of which he can find the address of _dl_runtime_resolve.

Figure 4.

After this, Leakless must invoke _dl_runtime_resolve, passing the link_map structure that it just corrupted as the first argument and an offset into the new .dynsym as the second parameter. However, as previously mentioned, _dl_runtime_resolve is not available in the GOT due to full RELRO. Therefore, Leakless must look for its address in the GOT of *another* ELF object, namely, a library loaded by the application that is not protected by full RELRO. In most cases, only ELF executables are compiled with full RELRO, and libraries are not. This is due to the fact that RELRO is designed to harden, at the cost of performance, specific applications that are deemed “risky”. Applying full RELRO to a shared library would impact the performance of all applications making use of this library, and thus, libraries are generally left unprotected. Since the order of libraries in the linked list is deterministic, Leakless can dereference the l_next entry in link_map to reach the entry describing a library that is not protected by full RELRO, dereference the entry in l_info corresponding to the DT_PLTGOT dynamic entry, dereference its value (i.e., the base address of that library’s GOT), and read the address of _dl_runtime_resolve from this GOT.

Leakless must then overcome a final issue: _dl_runtime_resolve will not only call the target function, but will also try to write its address to the appropriate GOT entry. If this happens, the program will crash, as the GOT is read-only when full RELRO is applied. We can circumvent this issue by faking the DT_JMPREL dynamic entry in the link_map structure that points to the .rel.dyn section. Leakless points it to an attacker-controlled memory area and writes an

Elf_Rel structure, with a target (r_offset field) pointing to a writeable memory area, referring to the symbol we are targeting. Therefore, when the library is resolved, the address will be written to a writeable location, the program will not crash, and the requested function will be executed.

5 Implementation

Leakless analyzes a provided binary to identify which of its techniques is applicable, crafts the necessary data structures, and generates a ROP chain that implements the chosen technique. The discovery of the initial vulnerability itself, and the automatic extraction of usable gadgets from a binary are orthogonal to the scope of our work, and have been well-studied in the literature and implemented in the real world [6, 16, 19, 20, 34, 38]. We designed Leakless to be compatible with a number of gadget finding techniques, and have implemented a manual backend (where gadgets are provided by the user) and a backend that utilizes ROPC [22], an automated ROP compiler prototype built on the approach proposed by Q [34].

We also developed a small test suite, composed of a small C program with a stack-based buffer overflow compiled, alternatively, with no protections, partial RELRO, and full RELRO. The test suite runs on GNU/Linux with the x86, x86-64 and ARM architectures and with FreeBSD x86-64.

5.1 Required Gadgets

Leakless comprises four different techniques that are used depending on the hardening techniques applied to the binary. These different techniques require different gadgets to be provided to Leakless. A summary of the types of gadgets is presented in Table 2. The write_memory gadget is mainly used to craft data structures at known memory locations, while the deref_write gadget to traverse and corrupt data structures (in particular link_map). The deref_save and copy_to_stack gadgets are used only in the full RELRO case. The aim of the former is to save at a known location the address of link_map and _dl_runtime_resolve, while the latter is used to copy link_map and the relocation index on the stack before calling _dl_runtime_resolve, since using PLT0 is not a viable solution.

For the interested reader, we provide in-depth examples of executions of Leakless in the presence of two different sets of mitigation techniques in the documentation of the Leakless code repository [17].

6 Evaluation

We evaluated Leakless in four ways. First, we determined the applicability of our technique against different dynamic loader implementations. We then analyzed the binaries distributed by several popular GNU/Linux and BSD distributions (specifically, Ubuntu, Debian, Fedora, and FreeBSD) to determine the percentage of binaries that would be susceptible to our attack. Then we applied Leakless in two real-world exploits against a vulnerable version of Wireshark and in a more sophisticated attack against Pidgin. Finally we used a Turing-complete ROP compiler to implement the approach used in Leakless and two other previously used techniques, and compared the size of the resulting chains.

6.1 Dynamic Loaders

To show Leakless' generality, especially across different ELF-based platforms, we surveyed several implementations of dynamic loaders. In particular, we found that the dynamic loader part of the *GNU C Standard Library* (also known as glibc and widely used in GNU/Linux distributions), several other Linux implementations such as *dietlibc*, *uClibc* and *newlib* (widespread in embedded systems) and the *OpenBSD* and *NetBSD* implementations are vulnerable to Leakless. Another embedded library, *musl*, instead, is not susceptible to our approach since it does not support lazy loading. *Bionic*, the C Standard Library used in Android, is also not vulnerable since it only supports PIE binaries. The most interesting case, out of all the loaders we analyzed, is *FreeBSD*'s implementation. In fact, it is the only one which performs boundary checks on arguments passed to _dl_runtime_resolve. All other loaders implicitly trust input arguments argument. Furthermore, *all* analyzed loaders implicitly trust the control structures that Leakless corrupts in the course of most of its attacks.

In summary, out of all of the loaders we analyzed, only two are immune to Leakless by design: musl, which does not support lazy symbol resolution, and bionic, which only supports PIE executables. Additionally, because the FreeBSD dynamic loader performs bounds checking, the technique explained in Section 4.2 is not applicable. However, the other techniques still work.

6.2 Operating System Survey

To understand Leakless' impact on real-world systems, we performed a survey of all binaries installed in default installations of several different Linux and BSD distributions. Specifically, we checked all binaries in /sbin, /bin, /usr/sbin, and /usr/bin on these systems and classified the binaries by the applicability of the techniques used by Leakless. The distributions that we considered were Ubuntu 14.10, Debian Wheezy, Fedora 20, and FreeBSD 10. We used both x86 and x86-64 versions of these systems. On Ubuntu and Debian, we additionally installed the LAMP (Linux, Apache, MySQL, PHP) stack as an attempt to simulate a typical server deployment configuration.

The five categories that we based our ratings on are as

Table 2: Gadgets required for the various approaches. The "Signature" column represents the name of the gadget and the parameters it accepts, while "Implementation" presents the behavior of the gadget in C-like pseudo code. The last four columns indicate whether a certain gadget is required for the corresponding approach presented in Section 4. Under RELRO, "N" indicates RELRO is disabled, "P" means partial RELRO is used, "H" stands for the partial RELRO and small 64 bit binaries using huge pages, and "F" denotes that full RELRO is enabled.

		RELRO			
Signature	Implementation	N	P	H	F
write_memory $(destination, value)$	$*(destination) = value$	✓	✓	✓	✓
deref_write$(pointer, offset, value)$	$*(*(pointer) + offset) = value$			✓	✓
deref_save $(destination, pointer, offset)$	$*(destination) = *(*(pointer) + offset)$				✓
copy_to_stack $(offset, source)$	$*(stack\_pointer + offset) = *(source)$				✓

follows:

Unprotected. This category includes binaries that have no RELRO or PIE. For these binaries, Leakless can apply its base case technique, explained in Section 4.1.

Partial RELRO. Binaries that have partial RELRO, but lack PIE, fall into this category. In this case, Leakless would apply the technique described in Section 4.2.

Partial RELRO (huge pages). Binaries in this category have partial RELRO, use huge pages, and are very small, therefore, they require Leakless to use the technique described in Section 4.3. They are included in this category.

Full RELRO. To attack binaries that use full RELRO, which comprise this category, Leakless must apply the technique presented in Section 4.4.

Not susceptible. Finally, we consider binaries that use PIE to be insusceptible to Leakless (further discussion on this in Section 7.2).

The results of the survey, normalized to the total number of binaries in an installation, are presented in Figure 5. We determined that, on Ubuntu, 84% of the binaries were susceptible to at least one of our techniques and 16% were protected with PIE. On Debian, Leakless can be used on 86% of the binaries. Fedora has 76% of susceptible binaries. Interestingly, FreeBSD ships no binaries with RELRO or PIE, and is thus 100% susceptible to Leakless.

Additionally, we performed a survey on the shared libraries of the systems we considered. We found that, on average, only 11% of the libraries had full RELRO protection. This has some interesting implications for Leakless: for a given binary, the likelihood of finding a loaded library without full RELRO is extremely high and, even if a vulnerable binary employs RELRO, Leakless can still apply its full RELRO attack to bypass this. This has the effect of making RELRO basically useless as a mitigation technique, unless it is applied system-wide.

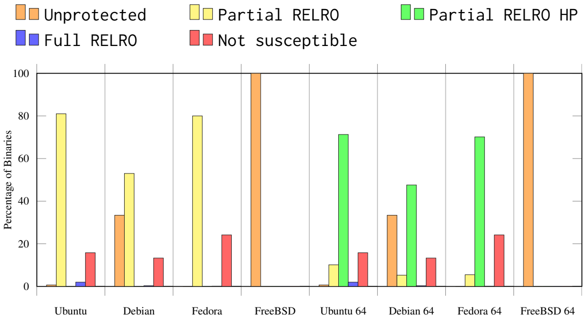


Figure 5: Classification of the binaries in default installations of target distributions. Binaries marked as `Unprotected`, `Partial RELRO`, `Partial RELRO HP` and `Full RELRO` require, respectively, to the attacks detailed in Sections 4.1, 4.2, 4.3 and 4.4, while for `Not susceptible` binaries, the Leakless approach is not applicable

6.3 Case Study: Wireshark

We carried out a case study in applying Leakless to a vulnerability in a program that does not present a direct line of communication to the attacker. In other words, the exploit must be done in one-shot, with no knowledge of the layout of the address space or the contents of libraries.

We picked a recent (April 2014) vulnerability [7], which is a stack-based buffer overflow in *Wireshark*'s MPEG protocol parser in versions 1.8.0 through 1.8.13 and 1.10.0 through 1.10.6. We carried out our experiments against a Wireshark 1.8.2 binary compiled with partial RELRO and one compiled with full RELRO. Both were compiled for x86-64 on Debian Wheezy and used the GNU C Library, without other protections such as PIE and stack canaries.

We used the manual Leakless backend to identify the required gadgets to construct the four necessary primitives (described in Section 5.1): *write_memory*, *deref_write*, *deref_save* and *copy_to_stack*. In the case of Wireshark, it was trivial to find gadgets to satisfy all of these primitives.

Leakless was able to construct a one-shot exploit using the attacks presented in Section 4.2 and Section 4.4. In both cases, the exploit leverages the dynamic loader in order to call the execve function from glibc to launch an executable of our choice.

6.4 Case Study: Pidgin

We also applied Leakless to Pidgin, a popular multi-protocol instant-messaging client, to build a more sophisticated exploit. Specifically, we wanted to perform a malicious operation without calling any anomalous system call (e.g. execve("/bin/sh")) which could trigger intrusion detection systems. We used Pidgin 2.10.7, building it from the official sources with RELRO enabled and targeting the x86 architecture.

To this end, we crafted an exploit designed to masquerade itself in legitimate functionality present in the application logic: tunneling connections through a proxy. The idea of the attack is that an IM service provider exploits a vulnerability such as CVE-2013-6487 [14] to gain code execution, and, using Pidgin's global proxy settings, redirects all IM traffic through a third-party server to enable chat interception.

Once we identified the necessary gadgets to use Leakless with full RELRO protection, it was easy to invoke functions contained in libpurple.so (where the core of the application logic resides) to perform the equivalent of the C code shown in Listing 2.

Listing 2: The Pidgin attack.

```
void *p, *a;
p = purple_proxy_get_setup(0);
purple_proxy_info_set_host(p, "legit.com");
purple_proxy_info_set_port(p, 8080);
purple_proxy_info_set_type(p, PURPLE_PROXY_HTTP);

a = purple_accounts_find("usr@xmpp", "prpl-xmpp");
purple_account_disconnect(a);
purple_account_connect(a);
```

Interestingly, some of this library-provided functionality is not imported into the Pidgin executable itself, and would be very challenging to accomplish in a single-stage payload, without Leakless.

6.5 ROP chain size comparison

To prove the effectiveness of the Leakless approach, we compared it with two existing techniques that allow an attacker to call arbitrary library functions. The first consists in scanning a library backwards, starting from an address in the .plt.got section, until the ELF header is found, and then scan forward to find a fingerprint of the function the attacker wants to invoke. This approach is feasible, but not very reliable, since different versions (or implementations) of a library might not be uniquely identified with a single fingerprint. The second technique is more reliable, since it implements the full symbol resolution process, as it is carried out by the dynamic loader.

Table 3: Size of the ROP chains generated by ROPC for each technique presented in Section 6.5, and by Leakless' manual backend (*). The second column represents the size in bytes for the setup and the first call, while the third column shows the additional cost (in bytes) for each subsequent call. Finally, the fourth column indicates the percentage of vulnerabilities used in Metasploit that would be feasible to exploit with a ROP chain of the *First call* size.

Technique	First call	Subsequent	Feasibility
ROPC - scan library	3468 bytes	+340 bytes	16.38%
ROPC - symbol resolution	7964 bytes	+580 bytes	8.67%
Leakless partial RELRO	648 bytes	+84 bytes	73.78%
Leakless full RELRO	2876 bytes	+84 bytes	17.44%
Leakless* partial RELRO	292 bytes	+48 bytes	95.24%
Leakless* full RELRO	448 bytes	+48 bytes	88.9%

We implemented these two approaches using a Turing-complete ROP compiler for x86, based on Q [34], called ROPC [22]. We compare these approaches against that of Leakless' ROPC backend, in partial RELRO and full RELRO modes. For completeness, we also include the Leakless' manual backend, with gadgets specified by the user.

In fact, the size of a ROP payload is critical, vulnerabilities often involve an implicit limit on the size of the payload that can be injected into a program. To measure the impact of Leakless' ROP chain size, we collected the size limits imposed on payloads of all the vulnerability descriptions included in the *Metasploit Framework* [31], a turn-key solution for launching exploits against known vulnerabilities in various software. We found that 946 of the 1,303 vulnerability specifications included a maximum payload size, with an average specified maximum payload size of 1332 bytes. To demonstrate the increase in the feasibility of automatically generating complex exploits, we include, for each evaluated technique, the percentage of Metasploit vulnerabilities for which the technique can automatically produce short enough ROP chains.

The results, in terms of length of the ROP chain generated for ROPC's test binaries and feasibility against the vulnerabilities used in Metasploit, are shown in Table 3. Leakless outperforms existing techniques, not only in the absolute size of the ROP chain to perform the initial call, but also in the cost of performing each additional call, which is useful in a sophisticated attack such as the one demonstrated in Section 6.4.

7 Discussion

In this section, we will discuss several aspects relating to Leakless: why the capabilities that it provides to attackers are valuable, when it is most applicable, what its limitations are, and what can be done to mitigate against them.

7.1 Leakless Applications

Leakless represents a powerful tool in the arsenal of exploit developers, aiding them in three main areas: functionality reuse, one-shot exploitation, and ROP chains shortening.

One-shot exploitation. While almost any exploit can be simplified by Leakless, we have designed it with the goal of enabling exploits that, without it, require an information disclosure vulnerability, but for which an information disclosure is not feasible or desirable. A large class of programs that fall under this category are file format parsers.

Code that parses file formats is extremely complex and, due to the complex, untrusted input that is involved, this code is prone to memory corruption vulnerabilities. There are many examples of this: the image parsing library `libpng` had 27 CVE entries over the last decade [10], and `libtiff` had 53 [11]. Parsers of complex formats suffer even more: the multimedia library `ffmpeg` has accumulated 170 CVE entries over the last five years alone [9]. This class of libraries is not limited to multimedia. Wireshark, a network packet analyzer, has 285 CVE entries, most of which are vulnerabilities in network protocol analysis plugins [12].

These libraries, and others like them, are often used *offline*. The user might first download a media or PCAP file, and *then* parse it with the library. At the point where the vulnerability triggers, an attacker cannot count on having a direct connection to the victim to receive an information disclosure and send additional payloads. Furthermore, most of these formats are *passive*, meaning that (unlike, say, PDF), they cannot include scripts that the attacker can use to simulate a two-step exploitation. As a result, even though these libraries might be vulnerable, exploits for them are either extremely complex, unreliable, or completely infeasible. By avoiding the information disclosure step, Leakless makes these exploits simpler, reliable, and feasible.

Functionality reuse. Leakless empowers attackers to call arbitrary functions from libraries loaded by the vulnerable application. In fact, the vulnerable application does not have to actually *import* this function; it just needs to link against the library (i.e., call any other function in the library). This is brings several benefits.

To begin with, the C Standard Library, which is linked against by most applications, includes functions that wrap almost every system call (e.g., `read()`, `execve()`, and so on). This means that Leakless can be used to perform any system call, in a short ROP chain, even without a system call gadget.

Moreover, as demonstrated in Section 6.4, Leakless enables easy reuse of existing functionality present in the application logic. This is important for two reasons.

First, this helps an attacker perform stealthy attacks by making it easier to masquerade an exploit as something the application might normally do. This can be crucial when a standard exploitation path is made infeasible by the presence of protection mechanisms such as seccomp [2], AppArmor [1], or SELinux [25].

Second, depending on the goals of the attacker, reusing program functionality may be better than simply executing arbitrary commands. Aside from the attack discussed in our Pidgin case study, an attacker can, for example, silently enable insecure cipher-suites, or versions of SSL, in the Firefox web browser with a single function call to `SSL_CipherPrefSetDefault` [24].

Shorter ROP chains. As demonstrated in Section 6.5, Leakless produces shorter ROP chains than existing techniques. In fact, in many cases, Leakless is able to produce ROP chains less than one kilobyte that lead to the execution of arbitrary functions. As many vulnerabilities have a limit as to the maximum size of the input that they will accept, this is an important result. For example, the vulnerability that we exploited in our Pidgin case study allowed a maximum ROP chain of one kilobyte. Whereas normal ROP compilation techniques would be unable to create automatic payloads for this vulnerability, Leakless was able to call arbitrary functions via an automatically-produced ROP chain that remained within the length limit.

7.2 Limitations

Leakless' biggest limitation is the inability to handle Position Independent Executables (PIEs) without a prior information disclosure. This is a general problem to any technique that uses ROP, as the absolute addresses of gadgets must be provided in the ROP chain. Additionally, without the base address of the binary, Leakless would be unable to locate the dynamic loader structures that it needs to corrupt.

When presented with a PIE executable, Leakless requires the attacker to provide the application's base address, which is presumably acquired via an information disclosure vulnerability (or, for example, by applying the technique presented in BROP [5]). While this breaks Leakless' ability to operate without an information disclosure, Leakless is likely still the most convenient way to achieve exploitation, as no library locations or library contents have to be leaked. Additionally, depending on the situation, the disclosure of just the address of the binary might be more feasible than the disclosure of the *contents* of an entire library. Unlike other techniques, which may need the latter, Leakless only requires the former.

In practice, PIEs are uncommon due to the associated cost in terms of performance. Specifically, measurements have shown that PIE overhead on x86 processors averages at 10%, while the overhead on x86-64 processors, thanks to instruction-pointer-relative addressing, averages at 3.6% [28].

Because of the overhead associated with PIE, most distributions ship with PIE enabled only for those applications deemed "risky". For example, according to their documentation, Ubuntu ships only 27 of their officially supported packages (i.e., packages in the "main" repository) with PIE enabled, out of over 27,000 packages [40]. As shown in Section 6.1, PIE executables comprise a minority of the executables on all of the systems that we surveyed.

7.3 Countermeasures

There are several measures that can be taken against Leakless, but they all have drawbacks. In this sections we analyze the most relevant ones.

Position Independent Executables. A quick countermeasure is to make every executable on the system position independent. While this would block Leakless's automatic operation (as discussed in Section 7.2), it would still allow the application of the Leakless technique when any information disclosure does occur. For that reason, and the performance overhead associated with PIE, we consider the other countermeasures described in this section to be better solutions to the problem.

Disabling lazy loading. When the `LD_BIND_NOW` environment variable is set, the dynamic loader will completely disable *lazy loading*. That is, all imports, for the program binary and any library it depends on, are resolved upon program startup. As a side-effect of this, the address of `_dl_runtime_resolve` does not get loaded into the GOT of any library, and Leakless cannot function. This is equivalent to enable full RELRO on the whole system, and consequently, it incurs in the same, non-negligible, performance overhead.

Disabling DT_DEBUG. Finally, Leakless also uses the `DT_DEBUG` dynamic entry, used by debuggers for intercepting loading-related events, to bypass full RELRO. Currently, this field is always initialized, opening the doors for Leakless' full RELRO bypass. To close this hole, the dynamic loader could be modified to only initialize this value when a debugger is present or in the presence of an explicitly-set environment variable.

Better protection of loader control structures. Leakless heavily relies on the fact that dynamic loader control structures are easily accessible in memory, and their locations are well-known. It would be beneficial for these structures to be better protected, or hidden in memory, instead of being loaded at a known location. For example, as shown in [29], these structures, along with any sections that provide control data for symbol resolution, could be marked as read-only after initialization. Such a development would eliminate Leakless' ability to corrupt these structures and would prevent the attack from redirecting the control flow to sensitive functions.

Additionally, modifying the loading procedure to use a table of `link_map` structure, and letting `_dl_runtime_resolve` take an index in this table, instead of a direct pointer, will break Leakless' bypass of full RELRO. However, this change would also break compatibility with any binaries compiled before the change is implemented.

Isolation of the dynamic loader. Isolating the dynamic loader from the address space of the target program could be an effective countermeasure. For instance, on Nokia's *Symbian OS*, which has a micro-kernel, the dynamic loader is implemented in a separate process as a *system server* which interfaces with the kernel [26]. This guarantees that the control structures of the dynamic loader cannot be corrupted by the program, and, therefore, this makes Leakless virtually ineffective. However, such a countermeasure would have a considerable impact on the overall performance of applications due to the overhead of IPC (Inter-Process Communication).

In general, the mitigations either represent a runtime performance overhead (PIE or loader isolation), a load-time performance overhead (non-lazy loading and system-wide RELRO), or a modification of the loading process (`DT_DEBUG` disabling or loader control structure hiding). In the long run, we believe that a redesign of the dynamic loader, with security in mind, would be extremely beneficial to the community. In the short term, there are options available to protect against Leakless, but they all come with a performance cost.

8 Conclusion

In this paper, we presented Leakless, a new technique that leverages functionality provided by the dynamic loader to enable attackers to use arbitrary, security-critical library functions in their exploits, without having to know where in the application's memory these functions are located. This capability allows exploits that, previously, required an information disclosure step to function.

Since Leakless leverages features mandated in the ELF binary format specification, the attacks it implements are applicable across architectures, operating systems, and dynamic loader implementations. Additionally, we showed how our technique can be used to bypass hardening schemes such as RELRO, which are designed to protect important control structures used in the dynamic resolution process. Finally, we proposed several countermeasures against Leakless, discussing the advantages and disadvantages of each one.

References

[1] AppArmor. http://wiki.apparmor.net/.

[2] A. Arcangeli. seccomp. https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt.

[3] A. Baratloo, N. Singh, and T. K. Tsai. Transparent Run-Time Defense Against Stack-Smashing Attacks. In *USENIX Annual Technical Conference, General Track*, pages 251–262, 2000.

[4] M. Bauer. Paranoid penguin: an introduction to Novell AppArmor. *Linux Journal*, 2006(148):13, 2006.

[5] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh. Hacking blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.

[6] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 380–394. IEEE, 2012.

[7] Common Vulnerabilities and Exposures. CVE-2014-2299. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-2299.

[8] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, volume 2, pages 119–129. IEEE, 2000.

[9] CVEDetails.com. ffmpeg: CVE security vulnerabilities. http://www.cvedetails.com/product/6315/Ffmpeg-Ffmpeg.html.

[10] CVEDetails.com. Libpng: Security Vulnerabilities. http://www.cvedetails.com/vendor/7294/Libpng.html.

[11] CVEDetails.com. Libtiff: CVE security vulnerabilities. http://www.cvedetails.com/product/3881/Libtiff-Libtiff.html.

[12] CVEDetails.com. Wireshark: CVE security vulnerabilities. http://www.cvedetails.com/product/8292/Wireshark-Wireshark.html.

[13] CWE. CWE/SANS Top 25 Most Dangerous Software Errors. http://cwe.mitre.org/top25/.

[14] N. V. Database. NVD - Detail - CVE-2013-6487. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-6487.

[15] A. Di Federico, A. Cama, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Leakless source code repository. https://github.com/ucsb-seclab/leakless.

[16] S. Dudek. The Art Of ELF: Analysis and Exploitations. http://bit.ly/1a8MeEw.

[17] T. Dullien, T. Kornau, and R.-P. Weinmann. A Framework for Automated Architecture-Independent Gadget Search. In *WOOT*, 2010.

[18] M. Fox, J. Giordano, L. Stotler, and A. Thomas. Selinux and grsecurity: A case study comparing linux security kernel enhancements. 2009.

[19] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *USENIX Security*, pages 49–64, 2013.

[20] C. Heitman and I. Arce. BARFgadgets. https://github.com/programa-stic/barf-project/tree/master/barf/tools/gadgets.

[21] inaz2. ROP Illmatic: Exploring Universal ROP on glibc x86-64. http://ja.avtokyo.org/avtokyo2014/speakers#inaz2.

[22] P. Kot. A Turing complete ROP compiler. https://github.com/pakt/ropc.

[23] P. Menage. Cgroups. *Available on-line at: http://www. mjmwired. net/kernel/Documentation/c-groups. txt*, 2008.

[24] Mozilla. SSL_CipherPrefSetDefault. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/SSL_functions/sslfnc.html#__SSL_CipherPrefSetDefault_.

[25] National Security Agency. Security-Enhanced Linux. http://selinuxproject.org/.

[26] Nokia. Symbian OS Internals - The Loader. http://developer.nokia.com/community/wiki/Symbian_OS_Internals/10._The_Loader#The_loader_server.

[27] H. Orman. The Morris worm: a fifteen-year perspective. *IEEE Security & Privacy*, 1(5):35–43, 2003.

[28] M. Payer. Too much PIE is bad for performance. 2012. https://nebelwelt.net/publications/12TRpie/gccPIE-TR120614.pdf.

[29] M. Payer, T. Hartmann, and T. R. Gross. Safe Loading - A Foundation for Secure Execution of Untrusted Programs. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 18–32, Washington, DC, USA, 2012. IEEE Computer Society.

[30] Phrack. Phrack - Volume 0xB, Issue 0x3a. `http://phrack.org/issues/58/4.html`.

[31] Rapid7, Inc. The Metasploit Framework. `http://www.metasploit.com/`.

[32] RedHat, Inc. Huge Pages and Transparent Huge Pages. `https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Performance_Tuning_Guide/s-memory-transhuge.html`.

[33] Santa Cruz Operation. System V Application Binary Interface, 2013. `http://www.sco.com/developers/gabi/latest/contents.html`.

[34] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit Hardening Made Easy. In *USENIX Security Symposium*, 2011.

[35] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[36] R. Shapiro, S. Bratus, and S. W. Smith. "Weird Machines" in ELF: A Spotlight on the Underappreciated Metadata. In *Proceedings of the 7th USENIX Conference on Offensive Technologies*, WOOT'13, pages 11–11, Berkeley, CA, USA, 2013. USENIX Association.

[37] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 48–62. IEEE, 2013.

[38] The Avalanche Project. Avalange - a dynamic defect detection tool. `https://code.google.com/p/avalanche/`.

[39] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning. On the Expressiveness of Return-into-libc Attacks. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, pages 121–141, Berlin, Heidelberg, 2011. Springer-Verlag.

[40] Ubuntu. Ubuntu Wiki - Security/Features. `https://wiki.ubuntu.com/Security/Features#Built_as_PIE`.

[41] R. N. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: Practical Capabilities for UNIX. In *USENIX Security Symposium*, pages 29–46, 2010.

A Symbol versioning challenges

In Section 3.4 we introduced the concept of symbol versioning, and in Section 4.2 we mentioned that its usage introduces additional constraints in the value that `Elf_Rel.r_info` can assume. In this Appendix we illustrate these constraints, and how Leakless can automatically verify and satisfy them.

A.1 Constraints due to symbol versioning

In presence of symbol versioning, the `Elf_Rel.r_info` field is used both as an index into the dynamic symbol table and as an index in the symbol version table (the `.gnu.version` section), which is composed by `Elf_Verneed` values. An `Elf_Verneed` value of zero or one has a special meaning, and stops the processing of the symbol version, which is a desirable situation for the attacker.

To understand the constraints posed by this, we introduce some definitions and naming conventions. *idx* is the index in `Elf_Rel.r_info` that Leakless has computed, baseof(x) is the function returning the base address of section x, sizeof(y) is the function returning the size in bytes of structure y, and $*$ is the pointer dereference operator. We define the following variables:

$$\begin{aligned} sym &= \text{baseof}(\texttt{.dynsym}) + idx \cdot \text{sizeof}(\texttt{Elf\_Sym}) \\ ver &= \text{baseof}(\texttt{.gnu.version}) + \\ &\quad + idx \cdot \text{sizeof}(\texttt{Elf\_Verneed}) \\ verdef &= \text{baseof}(\texttt{.gnu.version\_r}) + \\ &\quad + *(ver) \cdot \text{sizeof}(\texttt{Elf\_Verdef}) \end{aligned}$$

To be able to carry on the attack, the following conditions must hold:

1. *sym* points to a memory area controlled by the attacker, and
2. one of the following holds:
   (a) *ver* points to a memory area containing a zero or a one, or
   (b) *ver* points to a memory area controlled by the attacker, which will write a zero value there, or
   (c) *verdef* points to a memory area controlled by the attacker, which will place there an appropriately crafted `Elf_Verdef` structure.

All the other options result in an access to an unmapped memory area or the failure of the symbol resolution process, both of which result in program termination.

Leakless is able to satisfy these constraints automatically in most cases. The typical successful situation results in an *idx* value that points to a version index with value zero or one in the `.text` section (which usually comes after `.gnu.version`) and to a symbol in the `.data` or `.bss` section. A notable exception, where this is impossible to achieve, is in the case of small x86-64 ELF binaries compiled with the support of huge pages [32]. Using huge pages means that memory pages are aligned to boundaries of 2 MiB and, therefore, the segment containing the read-only sections (in particular, `.gnu.version` and `.text`) is quite far from the writeable segment (containing `.bss` and `.data`). This makes it hard to find a good value for *idx*.

A.2 The huge page issue

The effect of huge pages can be seen in the following examples:

```
$ readelf --wide -l elf-without-huge-pages

Program Headers:
  Type VirtAddr   MemSiz   Flg Align
  ...
  LOAD 0x00400000 0x006468 R E 0x1000
  LOAD 0x00407480 0x0005d0 RW  0x1000
  ...

$ readelf --wide -l elf-with-huge-pages

Program Headers:
  Type VirtAddr   MemSiz   Flg Align
  ...
  LOAD 0x00400000 0x00610c R E 0x200000
  LOAD 0x00606e10 0x0005d0 RW  0x200000
  ...
```

While in the first case the distance between the beginning of the executable and the writeable segments is in the order of the kilobytes, with huge pages is more than 2 MiB, and a valid value for *idx* cannot be found.

There are two ways to resolve the problems posed to Leakless by small 64-bit binaries.

The first option is to find a zero value for `Elf_Verneed` in the read-only segment (usually `.text`). Given *ro_start*, *ro_end* and *ro_size*, as the start and end virtual addresses and the size of the read-only segment respectively, and *rw_start*, *rw_end* and *rw_size* as the respective values for the writeable segment, the following must hold:

$$\begin{aligned} ro\_start \leq\ ver\ &< ro\_end \\ rw\_start \leq sym &< rw\_end \end{aligned}$$

Here, the most difficult case to satisfy is if `.dynsym` or `.gnu.version` start at *ro_start*. If we assume that *both* hold true, we can write the following:

$$\begin{aligned} idx \cdot \text{sizeof}(\texttt{Elf\_Verneed}) &< ro\_end - ro\_start \\ idx \cdot \text{sizeof}(\texttt{Elf\_Sym}) &\geq rw\_start - ro\_start \end{aligned}$$

Or, alternatively:

$$\begin{aligned} idx \cdot \text{sizeof}(\texttt{Elf\_Verneed}) &< ro\_size \\ idx \cdot \text{sizeof}(\texttt{Elf\_Sym}) &\geq 2 \text{ MiB} \end{aligned}$$

Knowing that `Elf_Verneed` and `Elf_Sym` have, respectively, a size of 2 and 24 bytes for 64 bit ELFs, we can compute the minimum value of `ro_size` to make this system of inequalities satisfiable. The result is 170.7 KiB. If the `.rodata` section is smaller than this size, an alternative method must be used.

The second option is to position `Elf_Verneed` in the writeable segment. In this case, the attack requirements can be described by the following system of inequalities:

$$\begin{aligned} rw\_start \leq\ ver\ &< rw\_end \\ rw\_start \leq sym &< rw\_end \end{aligned}$$

If we, once again, consider the most stringent constraints and apply the previously mentioned assumptions, we get the following:

$$\begin{aligned} idx \cdot \text{sizeof}(\texttt{Elf\_Verneed}) &\geq rw\_start - ro\_start \\ idx \cdot \text{sizeof}(\texttt{Elf\_Sym}) &< rw\_start - ro\_start + \\ &\quad + rw\_size \end{aligned}$$

Or, alternatively:

$$\begin{aligned} idx \cdot \text{sizeof}(\texttt{Elf\_Verneed}) &\geq 2 \text{ MiB} \\ idx \cdot \text{sizeof}(\texttt{Elf\_Sym}) &< 2 \text{ MiB} + rw\_size \end{aligned}$$

We can now put a lower bound on the size of the writeable segment (*rw_size*) to make the system satisfiable: 22 MiB. However, this is unreasonably large, and leads us to the conclusion that this approach is not viable with small 64 bit ELF binaries that use huge pages.

Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale

Kai Chen[‡,†], Peng Wang[†], Yeonjoon Lee[†], XiaoFeng Wang[†], Nan Zhang[†], Heqing Huang[§], Wei Zou[‡] and Peng Liu[§]
{chenkai, zouwei}@iie.ac.cn, {pw7, yl52, xw7, nz3}@indiana.edu, hhuang@cse.psu.edu, pliu@ist.psu.edu
[†]*Indiana University, Bloomington*
[‡]*State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences*
[§]*College of IST, Penn State University*

Abstract

An app market's vetting process is expected to be scalable and effective. However, today's vetting mechanisms are slow and less capable of catching new threats. In our research, we found that a more powerful solution can be found by exploiting the way Android malware is constructed and disseminated, which is typically through repackaging legitimate apps with similar malicious components. As a result, such attack payloads often stand out from those of the same repackaging origin and also show up in the apps not supposed to relate to each other.

Based upon this observation, we developed a new technique, called *MassVet*, for vetting apps at a massive scale, without knowing what malware looks like and how it behaves. Unlike existing detection mechanisms, which often utilize heavyweight program analysis techniques, our approach simply compares a submitted app with all those already on a market, focusing on the difference between those sharing a similar UI structure (indicating a possible repackaging relation), and the commonality among those seemingly unrelated. Once public libraries and other legitimate code reuse are removed, such diff/common program components become highly suspicious. In our research, we built this "DiffCom" analysis on top of an efficient similarity comparison algorithm, which maps the salient features of an app's UI structure or a method's control-flow graph to a value for a fast comparison. We implemented MassVet over a stream processing engine and evaluated it nearly 1.2 million apps from 33 app markets around the world, the scale of Google Play. Our study shows that the technique can vet an app within 10 seconds at a low false detection rate. Also, it outperformed all 54 scanners in VirusTotal (NOD32, Symantec, McAfee, etc.) in terms of detection coverage, capturing over a hundred thousand malicious apps, including over 20 likely zero-day malware and those installed millions of times. A close look at these apps brings to light intriguing new observations: e.g., Google's detection strategy and malware authors' countermoves that cause the mysterious disappearance and reappearance of some Google Play apps.

1 Introduction

The phenomenal growth of Android devices brings in a vibrant application ecosystem. Millions of applications (*app* for short) have been installed by Android users around the world from various *app markets*. Prominent examples include Google Play, Amazon Appstore, Samsung Galaxy Apps, and tens of smaller third-party markets. With this prosperity, the ecosystem is tainted by the rampancy of Android malware, which masquerades as a useful program, often through repackaging a legitimate app, to wreak havoc, e.g., intercepting one's messages, stealing personal data, sending premium SMS messages, etc. Countering this menace primarily relies on the effort from the app markets, since they are at a unique position to stop the spread of malware in the first place. Accomplishing this mission, however, is by no means trivial, as highlighted by a recent report [8] that 99% of mobile malware runs on Android devices.

Challenges in app vetting. More specifically, the protection today's app market puts in place is a *vetting process*, which screens uploaded apps by analyzing their code and operations for suspicious activities. Particularly, Google Play operates *Bouncer* [24], a security service that statically scans an app for known malicious code and then executes it within a simulated environment on Google's cloud to detect hidden malicious behavior. The problem here is that the static approach does not work on new threats (i.e., *zero-day* malware), while the dynamic one can be circumvented by an app capable of fingerprinting the testing environment, as discovered by a prior study [30]. Also the dynamic analysis can be heavyweight, which makes it hard to explore all execution paths of an app.

New designs of vetting techniques have recently been proposed by the research community [57, 28] for capturing new apps associated with known suspicious behavior, such as dynamic loading of binary code from a remote untrusted website [57], operations related to component hijacking [28], Intent injection [12], etc. All these approaches involve a heavyweight information-flow analysis and require a set of heuristics that characterize the known threats. They often need a dynamic analysis in addition to the static inspection performed on app code [57] and further human interventions to annotate the code or even participate in the analysis [14]. Moreover, emulators that most dynamic analysis tools employ can be detected and evaded by malware [23]. Also importantly, none of them has been put to a market-scale test to understand their effectiveness, nor has their performance been clearly measured.

Catching unknown malice. Actually, a vast majority of Android malware are repackaged apps [56], whose authors typically attach the same attack payload to different legitimate apps. In this way, not only do they hide their malicious program logic behind the useful functionalities of these apps, but they can also automate the repackaging process to quickly produce and distribute a large number of Trojans[1]. On the other hand, this practice makes such malware stand out from other repackaged apps, which typically incorporate nothing but advertising libraries [2]. Also as a result of the approach, similar code (typically in terms of Java methods) shows up in unrelated apps that are not supposed to share anything except popular libraries.

These observations present a new opportunity to catch malicious repackaged apps, the mainstay of Android malware, without using any heuristics to model their behavior. What we can do is to simply compare the code of related apps (an app and its repackaged versions, or those repackaged from the same app) to check their *different* part, and unrelated apps (those of different origins, signed by different parties) to inspect their *common* part to identify suspicious code segments (at the method level). These segments, once found to be inexplicable (e.g., not common libraries), are almost certain to be malicious, as discovered in our study (Section 4.2). This *DiffCom* analysis is well suited for finding previously unknown malicious behavior and also can be done efficiently, without resorting to any heavyweight information-flow technique.

Mass vetting at scale. Based on this simple idea, we developed a novel, highly-scalable vetting mechanism for detecting repackaged Android malware on one market or cross markets. We call the approach *mass vetting* or simply *MassVet*, as it does not use malware signatures and any models of expected malicious operations, and instead, solely relies on the features of existing apps on a market to vet new ones uploaded there. More specifically, to inspect a new app, MassVet runs a highly efficient DiffCom analysis on it against the whole market. Any existing app related to the new one (i.e., sharing the same repackaging origin) is quickly identified from the structural similarity of their user interfaces (aka., *views*), which are known to be largely preserved during repackaging (Section 2). Then, a *differential analysis* happens to those sharing the similar view structure (indicating a repackaging relation between them) when a match has been found. Also, an *intersection analysis* is performed to compare the new app against those with different view structures and signed by different certificates. The code components of interest discovered in this way, either the common (or similar) methods (through the intersection analysis) or different ones (by the differential analysis), are further inspected to remove common code reuses (libraries, sample code, etc.) and collect evidence for their security risks (dependence on other code, resource-access API calls, etc.), before a red flag is raised.

Supporting this mass vetting mechanism are a suite of techniques for high-performance view/code comparisons. Particularly, innovations are made to achieve a scalable analysis of different apps' user interfaces (Section 3.2). The idea is to project a set of salient features of an app's view graph (i.e., the interconnections between its user interfaces), such as types of widgets and events, to a single dimension, using a unique index to represent the app's location within the dimension and the similarity of its interface structure to those of others. In our research, we calculated this index as a geometric center of a view graph, called *v-core*. The v-cores of all the apps on the market are sorted to enable a binary search during the vetting of a new app, which makes this step highly scalable. The high-level idea here was applied to application clone detection [7], a technique that has been utilized in our research (mapping the features of a Java method to an index, called *m-core* in our research) for finding common methods across different apps (Section 3.3). It is important to note that for the view-graph comparison, new tricks need to be played to handle the structural changes caused by repackaging, e.g., when advertisement interfaces are added (Section 3.2).

Our findings. We implemented MassVet on a cloud platform, nearly 1.2 million real-world apps collected from 33 app markets around the world. Our experimental study demonstrates that MassVet vetted apps within ten seconds, with a low false positive rate. Most importantly, from the 1.2 million apps, our approach discovered 127,429 malware: among them at least 20 are likely zero-day and 34,026 were missed by the majority of the malware scanners run by *VirusTotal*, a website that syn-

[1] Those Trojans are typically signed by different keys to avoid blocking of a specific signer.

dicates 54 different antivirus products [43]. Our study further shows that MassVet achieved a better detection coverage than *any* individual scanner within VirusTotal, such as Kaspersky, Symantec, McAfee, etc. Other highlights of our findings include the discovery of malicious apps in leading app markets (30,552 from Google Play), and Google's strategies to remove malware and malware authors' countermoves, which cause mysterious disappearance and reappearance of apps on the Play Store.

Contributions. The contributions of the paper are summarized as follows:

- *New techniques*. We developed a novel mass vetting approach that detects new threats using nothing but the code of the apps already on a market. An innovative differential-intersection analysis (i.e., DiffCom) is designed to exploit the unique features of repackaging malware, catching the malicious apps even when their behavior has not been profiled *a priori*. This analysis is made scalable by its simple, static nature and the feature projection techniques that enable a cloud-based, fast search for view/code differences and similarities. Note that when the v-core and m-core datasets (only 100 GB for 1.2 million apps) are shared among multiple markets, MassVet can help one market to detect malicious submissions using the apps hosted by all these markets.
- *New discoveries*. We implemented MassVet and evaluated it using nearly 1.2 million apps, a scale unparalleled in any prior study on Android malware detection, up to our knowledge, and on a par with that of Google Play, the largest app market in the world with 1.3 million apps [39]. Our system captured tens of thousands of malware, including those slipping under the radar of most or all existing scanners, achieved a higher detection coverage than all popular malware scanners within VirusTotal and vetted new apps within ten seconds. Some malware have over millions of installs. 5,000 malware were installed over 10,000 times each, impacting hundreds of millions of mobile devices. A measurement study further sheds light on such important issues as how effective Google Play is in screening submissions, how malware authors hide and distribute their attack payloads, etc.

2 Background

Android App markets. Publishing an app on a market needs to go through an approval process. A submission will be inspected for purposes such as quality control, censorship, and also security protection. Since 2012, Google Play has been under the protection of Bouncer. This mechanism apparently contributes to the reduction of malware on the Play store, about 0.1% of all apps there as discovered by F-Secure [15]. On the other hand, this security vetting mechanism was successfully circumvented by an app that fingerprints its simulator and strategically adjusts its behavior [33]. Compared with the Android official market, how third-party markets review submitted apps is less clear. The picture painted by F-Secure, however, is quite dark: notable markets like Mumayi, AnZhi, Baidu, etc. were all found riddled with malware infiltrations [16].

Attempts to enhance the current secure vetting mechanisms mainly resort to conventional malware detection techniques. Most of these approaches, such as VetDroid [52], rely on tracking information flows within an app and the malicious behavior modeling for detecting malware. In the case that what the malware will do is less clear to the market, these approaches no longer help. Further, analyzing information flows requires semantically interpreting each instruction and carefully-designed techniques to avoid false positives, which are often heavyweight. This casts doubt on the feasibility of applying these techniques to a large-scale app vetting.

Repackaging. App repackaging is a process that modifies an app developed by another party and already released on markets to add in some new functionalities before redistributing the new app to the Android users. According to Trend Micro (July 15, 2014), nearly 80% of the top 50 free apps on Google Play have repackaged versions [49]. Even the Play store itself is reported to host 1.2% repackaged apps [58]. This ratio becomes 5% to 13% for third-party markets, according to a prior study [55]. These bogus apps are built for two purposes: either for getting advertisement revenues or for distributing malware [7]. For example, one can wrap AngryBird with ad libraries, including his own adverting ID to benefit from its advertising revenue. Malware authors also found that leveraging those popular legitimate apps is the most effective and convenient avenue to distribute their attack payloads: repackaging saves them a lot of effort to build the useful functionalities of a Trojan and the process can also be automated using the tools like smali/baksmali [36]; more importantly, they can free-ride the popularity of these apps to quickly infect a large number of victims. Indeed, research shows that the vast majority of Android malware is repackaged apps, about 86% according to a study [56]. A prominent feature shared by all these repackaged apps, malicious or not, is that they tend to keep the original user interfaces intact, so as to impersonate popular legitimate apps.

Scope and assumptions. MassVet is designed to detect repackaged Android malware. We do not consider the situation that the malware author makes his malicious payload an inseparable part of the repackaged app, which needs much more effort to understand the legitimate app than he does today. Also, MassVet can handle typical code obfuscation used in most Android apps (Section 3). However, we assume that the code has not been

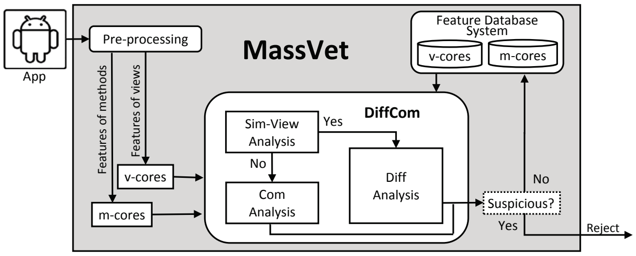


Figure 1: The Architecture of MassVet.

obfuscated to the extent that even disassembly cannot go through. When this happens, not only our approach but also most of other static analyses will fail. Finally, we assume that the app market under protection accommodates a large number of highly-diverse apps, so that for the malicious repackaged app uploaded, on the market there will be either another app sharing its repackaging origin or the one incorporating the same attack payload. To make this more likely to happen, different markets can share the feature datasets of their apps (i.e., v-cores and m-cores) with each other. Note that such datasets are very compact, only 100 GB for 1.2 million apps.

3 MassVet: Design and Implementation

3.1 Overview

Design and architecture. To detect unknown malware at a large scale, we come up with a design illustrated in Figure 1. It includes three key components: a preprocessing module, a feature database system and a DiffCom module. The preprocessing module automatically analyzes a submitted app, which includes extracting the features of its view structure and methods, and then summarizing them into the app's v-cores and m-cores respectively. The DiffCom component then works on these features, searching for them within the app market's v-core and m-core databases. Matches found there are used to identify suspicious different or common methods, which are further screened to remove false positives.

How it works. Here we use an example to walk through the work flow of the system. MassVet first processes all the apps on a market to create a v-core database for view structures and an m-core database for Java methods (Section 3.4). Both databases are sorted to support a binary search and are used for vetting new apps submitted to the market. Consider a repackaged AngryBird. Once uploaded to the market, it is first automatically disassembled at the preprocessing stage into a `smali` representation, from which its interface structures and methods are identified. Their features (for views, user interfaces, types of widgets and events, and for methods, control flow graphs and bytecode) are mapped to a set of v-cores (Section 3.2) and m-cores (Section 3.3) through calculating the geometric centers of the view graphs and control-flow graphs respectively. The app's v-cores are first used to query the database through a binary search. Once a match is found, which happens when there exists another app with a similar AngryBird user interface structure, the repackaged app is compared with the app already on the market at the method level to identify their difference. Such different methods (*diff* for short) are then automatically analyzed to ensure that they are not ads libraries and indeed suspicious, and if so, are reported to the market (Section 3.2). When the search on the v-core database comes back with nothing[2], MassVet continues to look for the AngryBird's m-cores in the method database. If a similar method has been found, our approach tries to confirm that indeed the app including the method is unrelated to the submitted AngryBird and it is not a legitimate code reuse (Section 3.3). In this case, MassVet reports that a suspicious app is found. All these steps are fully automated, without human intervention.

3.2 Fast User-Interface Analysis

As discussed before, the way MassVet vets an app depends on whether it is *related* to any other app already on the market. Such a relation is established in our research through a quick inspection of apps' user interfaces (UI) to identify those with similar view structures. When such apps are not "officially" connected, e.g., produced by the same party, the chance is that they are of the same repackaging origin, and therefore their diffs become interesting for malicious code detection. This interface-based relation identification is an alternative to code-based identification: a malicious repackaged app can be obfuscated and junk code can be easily added to make it look very different from the original version in terms of the similarity between their code (e.g., percentage of similar methods shared between them). On the other hand, a significant change to the user interface needs more effort and most importantly affects user experience, making it more difficult for the adversary to free ride the popularity of the original app. Therefore, most repackaged apps preserve their original UI structures, as found by the prior research [50]. In our research, we further discovered that many repackaged apps incorporate a large amount of new code, even more than that in their original versions, but still keep those apps' UI structures largely intact.

The idea of using view structures to detect repackaged apps has been preliminarily explored in prior research [50], which utilizes subgraph isomorphism algorithms to measure the similarity between two apps. However, the approach is less effective for the apps with relatively simple user-interface structures, and most impor-

[2] The market can also choose to perform both differential and interaction analyses for all new apps (Section 3.3).

tantly, agonizingly slow: it took 11 seconds to compare a pair of apps [50], which would need 165 days to analyze one submission against all 1.3 million apps on the Google Play store.

Following we elaborate our new solution designed for an accurate and high performance app-view analysis.

Feature extraction. An app's user interface consists of a set of views. Each view contains one or more visual widgets such as Button, ListView, TextView, etc. These UI components respond to users' input events (e.g., tapping and swiping) with the operations specified by the app developer. Such responses may cause visible changes to the current view or transitions to other views. This interconnection structure, together with the layouts and functionalities of individual views, was found to be sufficiently unique for characterizing each app [50].

In our research, we model such a UI structure as a *view graph*, which is a directed weighted graph including all views within an app and the navigation relations (that is, the transition from one view to another) among them. On such a graph, each node is a view, with the number of its *active* widgets (those with proper event-response operations) as its *weight*, and the arcs connecting the nodes describe the navigation (triggered by the input events) relations among them. According to the types of the events (e.g., `onClick`, `onFocusChange`, `onTouch`, etc.), edges can be differentiated from each other.

Such a view graph can effectively describe an app with a reasonably complicated UI structure. However, it becomes less effective for the small apps with only a couple of views and a rather straightforward connection structure. To address this issue, we enrich the view graph with additional features, including other UIs and the types of widgets that show up in a view. Specifically, in addition to view, which is displayed through invocation of an Android *Activity*, the UIs such as `AlertDialog` are also treated as nodes for the graph. Custom dialogs can be handled by analyzing class inheritance. Further, each type of widgets is given a unique value, with a sole purpose of differentiating it from other types. In this way, we can calculate a UI node's weight by adding together the values associated with the widgets it carries to make a small view graph more distinctive from others. An example is illustrated in Figure 2.

Note that we avoid using text labels on UI elements or other attributes like size or color. All the features selected here, including UIs, types of widgets and events that cause transitions among UIs, are less manipulable: in the absence of serious effort, any significant change to them (e.g., adding junk widgets, modifying the widget types, altering the transitions among views) will perceivably affect user experience, making it more difficult for the adversary to use them to impersonate popular apps.

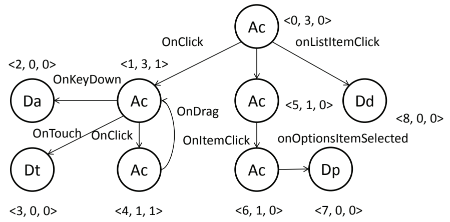


Figure 2: A View-graph example.
Ac: Activity; Da: AlertDialog; Dt: TimePickerDialog
Dp: ProgressDialog; Dd: DatePickerDialog

To construct the view graph for a submitted app, the preprocessing module automatically analyzes its code to recover all UI-related interprocess communication (IPC), the channel through which an Android app invokes user interfaces. Such IPC calls include `startActivity` and `startActivityForResult`. For each call, our approach locates it within a UI and further identifies the UI it triggers. Specifically, the program location of the IPC is examined to determine whether it is inside a UI-related class v. Its parameter is parsed to find out the class it calls (v'). In this case, nodes are created on the view graph for both classes (UIs) and an edge is added to link v to v'. Also, the type of the edge is determined by the event handler at which the IPC is located: for example, when the call is found inside the `onClick` function for a button, we know that this widget is used to cause a view transition. All such widgets are identified from each class for determining the weight of its node.

Design for scale. Once a view graph is recovered from an app, we want to quickly compare it across a market (or markets) to identify those related to the app. This operation needs to be of high-performance, capable of processing over one million apps within seconds. To this end, we applied a recently proposed similarity comparison algorithm, called *Centroids* [7], to the view-graph analysis. Centroid maps the features of a program's control-flow graph (CFG) into a value, which is calculated as the geometric center of the program. This value has a *monotonicity* property: that is, whenever a minor change happens to the CFG, the value changes accordingly at a small scale, reflecting the level of the difference made to the program. This property localizes the global comparison to a small number of "neighbors" to achieve high scalability without losing accuracy. The approach was used for the method comparison in our research (Section 3.3). However, it cannot be directly adopted for analyzing the UI structure, as the view graph is quite different from the CFG. Also, an app's graph is often fragmented due to the unusual ways to trigger some of its modules: e.g., most advertisement views are invoked through callbacks using the APIs of their library; as a result, their graph becomes separated from that of the main program. Here we describe how we address these issues.

Given a set of subgraphs for an app UI, $G_{i=1\cdots n}$, our preprocess module analyzes them one by one to calculate their individual geometric centers, i.e., v-cores. For a subgraph G_i, the first thing that needs to be done is to convert the features of each of its nodes (i.e., view) into a three-dimensional vector $\vec{c} = \{\alpha, \beta, \gamma\}$. Here α is a *sequence number* assigned to each node in G_i, which is done through an ordered deep-first traversal of G_i: starting from its main view, we select nodes to visit in the order of the sizes of their subtrees, and use their individual weights to break a tie; each node traversed in this way gets the number based upon its order of being visited. If two subtrees have the same size, we select the one according to their node types. In this way, we ensure that the assignment of sequence numbers is unique, which only depends on the structure of the directed weighted graph. The second element, β, in the vector is the out degree of the node: that is, the number of UIs the node can lead to. Finally, γ is the number of "transition loops" the current node is involved: i.e., the structure that from the node, there exists a navigation path that by visiting each node on the path only once, the user is able to navigate back to the current view. Figure 2 presents an example that show how such a vector is constructed.

After every node k on G_i has been given a vector $\vec{c}_k$, we can calculate its geometric center, i.e., v-core vc_i, as follows:

$$vc_i = \frac{\sum_{e(p,q) \in G_i} (w_p \vec{c}_p + w_q \vec{c}_q)}{\sum_{e(p,q) \in G_i} (w_p + w_q)}$$

where $e(p,q)$ denotes an edge in G_i from node p to q and w_p is the weight of node p. With the monotonicity of v-cores, we can sort them for a large number of apps to support a binary search. In this way, the subgraph G_i can be quickly compared with millions of graphs to identify similar ones. Specifically, given another graph G_t with a v-core vc_t, we consider that it matches G_i if $|vc_i - vc_t| \leq \tau$, where τ is a threshold. Further, given two apps sharing a subset of their view-graphs $G_{i(l=1\cdots m)}$, we consider that these two apps are similar in their UI structure when the following happens to at least one app: $\sum_l |G_{i(l)}| / \sum_i |G_i| \geq \theta$: that is, most of the app's view structures also appear in the other app (with θ being a threshold). This ensures that even when the adversary adds many new views to an app (e.g., through fake advertisements), the relation between the repackaged app and the original one can still be identified.

In our research, such thresholds were determined from a training process using 50,000 randomly selected apps (Section 3.3). We set different thresholds and measured the corresponding false positive/negative rates. For false positives, we randomly sampled 50 app pairs detected by our approach under each threshold and manually checked their relations. For false negatives, we utilized 100 app pairs known to have repackaging relations as the ground truth to find out the number of pairs our approach identified with different thresholds. The study shows that when $\tau = 0$ and $\theta = 0.8$, we got both a low false positive rate (4%) and a low false negative rate (6%). Among these 50,000 apps, we found that 26,317 app pairs had repackaging relations, involving 3,742 apps in total.

Effectiveness of the view-graph analysis. Compared with existing code-based approaches [7], the view-graph analysis turns out to be more effective at detecting apps of the same repackaging origin. Specifically, we randomly selected 10,000 app pairs (involving 17,964 apps) from those repackaged from the same programs, as discovered from 1.2 million apps we collected (Section 4.1). Many of these repackaging pairs involve the apps whose code significantly differ from each other. Particularly, in 14% of these pairs, *two apps were found to have less than 50% of their individual code in common*. This could be caused by a large library (often malicious) added to an app during repackaging or junk code inserted for the purpose of obfuscation. Since these apps look so different based upon their code, their repackaging relations cannot be easily determined by program analysis. However, they were all caught by our approach, simply because the apps' view-graphs were almost identical.

3.3 DiffCom Analysis at Scale

For an app going through the mass vetting process, the view-graph analysis first determines whether it is related to any app already on the market. If so, these two apps will be further compared to identify their diffs for a malware analysis. Otherwise, the app is checked against the whole market at the method level, in an attempt to find the program component it shares with other apps. The diffs and common component are further inspected to remove common code reuse (libraries, sample code, etc.) and collect evidence for their security risks. This "difference-commonality" analysis is performed by the DiffCom module. We also present the brick and mortar for efficient code-similarity analyzer and discuss the evasion of DiffCom.

The brick and mortar. To vet apps at the market scale, DiffCom needs a highly efficient code-similarity analyzer. In our research, we chose Centroids [7] as this building block. As discussed before, this approach projects the CFG of a program to its geometric center, in a way similar to the view-graph analysis. More specifically, the algorithm takes a basic program block as a node, which includes a sequence of consecutive statements with only a single input and output. The weight of the block is the number of the statements it contains. For each node on the CFG, a sequence number is assigned, together with the counts of the branches it connects and the number of loops it is involved in. These parameters are used to calculate the geometric center of the program.

To prepare for mass vetting, our approach first goes through all the apps on a market and breaks them into methods. After removing common libraries, the preprocessing module analyzes their code, calculates the geometric centers for individual methods (i.e., the m-cores) and then sorts them before storing the results in the database. During the vetting process, if a submitted app is found to share the view graph with another app, their diffs are quickly identified by comparing the m-cores of their individual methods. When the app needs to go through the intersection step, its methods are used for a binary search on the m-core database, which quickly discovers those also included in existing apps. Here we elaborate how these operations are performed. Their overhead is measured in Section 4.2.

Analyzing diffs. Whenever an app is found to relate to another one from their common view graph, we want to inspect the difference part of their code to identify suspicious activities. The rationale is that repackaged apps are the mainstay of Android malware, and the malicious payloads are often injected automatically (using tools like smali/baksmali) without any significant changes to the code of the original app, which can therefore be located by looking at the diffs between the apps of the same repackaging origin. Such diffs are quickly identified by comparing these two apps' m-cores: given two ordered sequences of m-cores L and L', the diff between the apps at the method level is found by merging these two lists according to the orders of their elements and then removing those matching their counterparts on the other list; this can be done within $\min(|L|,|L'|)$ steps.

However, similarity of apps' UIs does not always indicate a repackaging relation between them. The problem happens to the apps produced by the same party, individual developers or an organization. In this case, it is understandable that the same libraries and UIs could be reused among their different products. It is even possible that one app is actually an updated version of the other. Also, among different developers, open UI SDKs such as Appcelerator [3] and templates like Envatomarket [13] are popular, which could cause the view structures of unrelated apps to look similar. Further, even when the apps are indeed repackaged, the difference between them could be just advertisement (ad) libraries instead of malicious payloads. A challenge here is how to identify these situations and avoid bringing in false alarms.

To address these issues, MassVet first cleans up a submitted app's methods, removing ad and other libraries, before vetting the app against the market. Specifically, we maintain a white list of legitimate ad libraries based on [6], which includes popular mobile ad platforms such as MobWin, Admob, etc. To identify less known ones, we analyzed a training set of 50,000 apps randomly sampled from three app markets, with half of them from Google Play. From these apps, our analysis discovered 34,886 methods shared by at least 27,057 apps signed by different parties. For each of these methods, we further scanned its hosting apps using VirusTotal. If none of them were found to be malicious, we placed the method on the white list. In a similar way, popular view graphs among these apps were identified and the libraries associated with these views are white-listed to avoid detecting false repackaging relations during the view-graph analysis. Also, other common libraries such as Admob were also removed during this process, which we elaborate later. Given the significant size of the training set (50,000 randomly selected apps), most if not all legitimate libraries are almost certain to be identified. This is particularly true for those associated with advertising, as they need certain popularity to remain profitable. On the other hand, it is possible that the approach may let some zero-day malware fall through the cracks. In our research, we further randomly selected 50 ad-related methods on the list and searched for them on the Web, and confirmed that all of them were indeed legitimate. With this false-negative risk, still our approach achieved a high detection coverage, higher than any scanner integrated in VirusTotal (Section 4.2).

When it comes to the apps produced by the same party, the code they share is less popular and therefore may not be identified by the approach. The simplest solution here is to look at similar apps' signatures: those signed by the same party are not considered to be suspicious because they do have a good reason to be related. This simple treatment works most of time, since legitimate app vendors typically sign their products using the same certificate. However, there are situations when two legitimate apps are signed by different certificates but actually come from the same source. When this happens, the diffs of the apps will be reported and investigated as suspicious code. To avoid the false alarm, we took a close look at the legitimate diffs, which are characterized by their intensive connections with other part of the app. They are invoked by other methods and in the meantime call the common part of the code between the apps. On the other hand, the malicious payload packaged to a legitimate app tends to stand alone, and can only be triggered from a few (typically just one) program locations and rarely call the components within the original program.

In our research, we leveraged this observation to differentiate suspicious diffs from those likely to be legitimate. For each diff detected, the DiffCom analyzer looks for the calls it makes toward the rest of the program and inspects the `smali` code of the app to identify the references to the methods within the diff. These methods will go through a further analysis only when such interactions are very limited, typically just an inward invocation, without any outbound call. Note that current malware authors do not make their code more connected to the legitimate app they repackage, mainly because more effort is needed to understand the code of the app and carefully construct the attack. A further study is needed to understand the additional cost required to build more sophisticated malware to evade our detection.

For the diff found in this way, DiffCom takes further measures to determine its risk. A simple approach used in our implementation is to check the presence of API calls (either Android framework APIs or those associated with popular libraries) related to the operations considered to be dangerous. Examples include `getSimSerialNumber`, `sendTextMessage` and `getLastKnownLocation`. The findings here indicate that the diff code indeed has the means to cause damage to the mobile user's information assets, though how exactly this can happen is not specified. This is different from existing behavior-based detection [27], which looks for much more specific operation sequences such as "reading from the contact list and then sending it to the Internet". Such a treatment helps suppress false alarms and still preserves the generality of our design, which aims at detecting unknown malicious activities.

Analyzing intersections. When no apparent connection has been found between an app and those already on the market, the vetting process needs to go through an intersection analysis. This also happens when DiffCom is configured to perform the analysis on the app that has not been found to be malicious at the differential step. Identification of common methods a newly submitted app carries is rather straightforward: each method of the app is mapped to its m-core, which is used to search against the m-core database. As discussed before, this can be done through a binary search. Once a match is found, DiffCom further inspects it, removing legitimate connections between the apps, and reports the finding to the market.

Again, the main challenge here is to determine whether two apps are indeed unrelated. A simple signature check removes most of such connections but not all. The "stand-alone" test, which checks whether a set of methods intensively interact with the rest of an app, does not work for the intersection test. The problem here is that the common methods between two repackaged apps may not be the complete picture of a malicious payload, making them different from the diff identified in the differential-analysis step: different malware authors often use some common toolkits in their attack payloads, which show up in the intersection between their apps; these modules still include heavy interactions with other components of the malware that are not found inside the intersection. As a result, this feature, which works well on diffs, cannot help to capture suspicious common code among apps.

An alternative solution here is to look at how the seemingly unrelated apps are actually connected. As discussed before, what causes the problem is the developers or organizations that reuse code internally (e.g., a proprietary SDK) but sign the apps using different certificates. Once such a relation is also identified, we will be more confident about whether two apps sharing code are independent from each other. In this case, the common code becomes suspicious after all public libraries (e.g., those on the list used in the prior research [6]) and code templates have been removed. Here we describe a simple technique for detecting such a hidden relation.

From our training dataset, we found that most code reused legitimately in this situation involves user interfaces: the developers tend to leverage existing view designs to quickly build up new apps. With this practice, even though two apps may not appear similar enough in terms of their complete UI structures (therefore they are considered to be "unrelated" by the view-graph analysis), a close look at the subgraphs of their views may reveal that they actually share a significant portion of their views and even subgraphs. Specifically, from the 50,000 apps in our training set, after removing public libraries, we found 30,286 sharing at least 30% of their views with other apps, 16,500 sharing 50% and 8,683 containing no less than 80% common views. By randomly sampling these apps (10 each time) and analyzing them manually, we confirmed that when the portion goes above 50%, almost all the apps and their counterparts are either from the same developers or organizations, or having the same repackaging origins. Also, once the shared views become 80% or more, almost always the apps are involved in repackaging. Based upon this observation, we run an additional correlation check on a pair of apps with common code: DiffCom compares their individual subgraphs again and if a significant portion (50%) is found to be similar, they are considered related and therefore their intersection will not be reported to the market.

After the correlation check, all the apps going through the intersection analysis are very likely to be unrelated. Therefore, legitimate code shared between them, if any, is almost always public libraries or templates. As described before, we removed such common code through white-listing popular libraries and further complemented the list with those discovered from the training set: methods in at least 2,363 apps were considered legitimate public resources if all these apps were cleared by VirusTotal. Such code was further sampled and manually analyzed in our study to ensure that it indeed did not involve any suspicious activities. With all such libraries removed, the shared code, particularly the method with dangerous APIs (e.g., `getSimSerialNumber`), is reported as possible malicious payload.

Evading MassVet. To evade MassVet, the adversary could try to obfuscate app code, which can be tolerated to some degree by the similarity comparison algorithm we use [7]. For example, common obfuscation techniques, such as variable/method renaming, do not affect centroids. Also the commonality analysis can only be defeated when the adversary is willing to significantly alter the attack payload (e.g., a malicious SDK) each time when he uses it for repackaging. This is not supported by existing automatic tools like ADAM [53] and DroidChameleon [32], which always convert the same code to the same obfuscated form. Further, a deep obfuscation will arouse suspicion when the app is compared with its repackaging origin that has not been obfuscated. The adversary may also attempt to obfuscate the app's view graphs. This, however, is harder than obfuscating code, as key elements in the graph, like event functions `OnClick`, `OnDrag`, etc., are hardcoded within the Android framework and cannot be modified. Also adding junk views can be more difficult than it appears to be: the adversary cannot simply throw in views disconnected from existing sub-graphs, as they will not affect how MassVet determines whether two view-graphs match (Section 3.2); otherwise, he may connect such views to existing sub-graphs (potentially allowing them to be visited from the existing UI), which requires understanding a legitimate app's UI structures to avoid affecting user experience.

We further analyzed the effectiveness of existing obfuscation techniques against our view-graph approach over 100 randomly selected Google-Play apps. Popular obfuscation tools such as DexGuard [37] and ProGuard [38] only work on Java bytecode, not the Dalvik bytecode of these commercial apps. In our research, we utilized ADAM [53] and DroidChameleon [32], which are designed for Dalvik bytecode, and are highly effective according to prior studies [53, 32]. Supposedly they can also work on view-related code within those apps. However after running them on the apps, we found that their v-cores, compared with those before the obfuscation, did not change at all. This demonstrates that such obfuscation is not effective on our view-graph approach.

On the other hand, we acknowledge that a new obfuscation tool could be built to defeat MassVet, particularly its view-graph search and the Com step. The cost for doing this, however, is less clear and needs further effort to understand (Section 5).

3.4 System Building

In our research, we implemented a prototype of MassVet using C and Python, nearly 1.2 million apps collected from 33 markets, including over 400,000 from Google Play (Section 4.1). Before these apps can be used to vet new submissions, they need to be inspected to detect malicious code already there. Analyzing apps of this scale and utilizing them for a real-time vetting require carefully designed techniques and a proper infrastructure support, which we elaborate in this section.

System bootstrapping and malware detection. To bootstrap our system, a market first needs to go through all its existing apps using our techniques in an efficient way. The APKs of these apps are decompiled into `smali` (using the tool baksmali [36][3]) to extract their view graphs and individual methods, which are further converted into v-cores and m-cores respectively. We use NetworkX [29] to handle graphs and find loops. Then these features (i.e., cores) are sorted and indexed before stored into their individual databases. In our implementation, such data are preserved using the Sqlite database system, which is characterized by its small size and efficiency. For all these apps, 1.5 GB was generated for v-cores and 97 GB for m-cores.

The next step is to scan all these 1.2 million apps for malicious content. A straightforward approach is to inspect them one by one through the binary search. This will take tens of millions of steps for comparisons and analysis. Our implementation includes an efficient alternative. Specifically, on the v-core database, our system goes through the whole sequence from one end (the smallest element) to the other end, evaluating the elements along the way to form *equivalent groups*: all those with identical v-cores are assigned to the same group[4].

All the subgraphs within the same group match each other. However, assembling them together to determine the similarity between two apps turns out to be tricky. This is because the UI of each app is typically broken into around 20 subgraphs distributed across the whole ordered v-core sequence. As such, any attempt to make the comparison between two apps requires to go through all equivalent groups. The fastest way to do that is to maintain a table for the number of view subgraphs shared between any two apps. However, this simple approach requires a huge table, half of 1.2 million by 1.2 million in the worst case, which cannot be completely loaded into the memory. In our implementation, a trade-off is made to save space by only inspecting 20,000 apps against the rest 1.2 million each time, which requires going through the equivalent groups for 60 times and uses about 100 GB memory at each round.

The inspection on m-cores is much simpler and does not need to compare one app against all others. This is because all we care about here are just the common methods that already show up within individual equiva-

[3] Very few apps (0.01%) cannot be decompiled in our dataset due to the limitation of the tool.

[4] In our implementation, we set the threshold τ to zero, which can certainly be adjusted to tolerate some minor variations among similar methods.

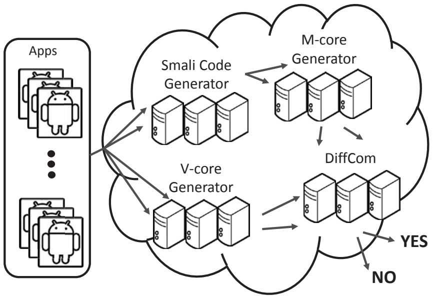


Figure 3: Cloud framework for MassVet.

lent groups. Those methods are then further analyzed to detect suspicious ones.

Cloud support. To support a high-performance vetting of apps, MassVet is designed to work on the cloud, running on top of a stream processing framework (Figure 3). Specifically, our implementation was built on *Storm* [40], an open-source stream-processing engine that also powers leading web services such as WebMD, Alibaba, Yelp, etc. Storm supports a large-scale analysis of a data stream by a set of worker units that connect to each other, forming a topology. In our implementation, the work flow of the whole vetting process is converted into such a topology: a submitted app is first disassembled to extract view graphs and methods, which are checked against the white list to remove legitimate libraries and templates; then, the app's v-cores and m-cores are calculated, and a binary search on the v-core database is performed; depending on the results of the search, the differential analysis is first run, which can be followed by the intersection analysis. Each operation here is delegated to a worker unit on the topology and all the data associated with the app are in a single stream. The Storm engine is designed to support concurrently processing multiple streams, which enables a market to efficiently vet a large number of submissions.

4 Evaluation and Measurement

4.1 Setting the Stage

App collection. We collected 1.2 million apps from 33 Android markets, including over 400,000 from Google Play, 596,437 from 28 app stores in China, 61,866 from European stores and 27,047 from other US stores as elaborated in Table 5 in Appendix. We removed duplicated apps according to their MD5. All the apps we downloaded from Google Play have complete meta data (upload date, size, number of downloads, developer, etc.), while all those from third-party markets do not.

The apps from Google Play were selected from 42 categories, including Entertainment, Tools, Social, Communication, etc. From each category, we first went for its top 500 popular ones (in terms of number of installs) and then randomly picked up 1000 to 30,000 across the whole category. For each third-party market, we just randomly collected a set of apps (Table 5) (190 to 108,736, depending on market sizes). Our collection includes high-profiled apps such as Facebook, Skype, Yelp, Pinterest, WeChat, etc. and those less popular ones. Their sizes range from less than 1 MB to more than 100 MB.

Validation. For the suspicious apps reported by our prototype, we validated them through VirusTotal and manual evaluations. Virustotal is the most powerful public malware detection system, which is a collection of 54 anti-malware products, including the most high-profile commercial scanners. It also provides the scanning service on mobile apps [44]. VirusTotal has two modes, complete scanning (which we call *new scan*) and using cached results (called *cached scan*). The latter is fast, capable of going through 200 apps every minute, but only covers those that have been scanned before. For the programs it has never seen or uploaded only recently, the outcome is just "unknown". The former determines whether an app is malicious by running on it all 54 scanners integrated within VirusTotal. The result is more up-to-date but the operation is much slower, taking 5 minutes for each app.

To validate tens of thousands suspicious cases detected from the 1.2 million apps (Section 4.2), we first performed the cached scan to confirm that most of our findings were indeed malicious. The apps reported to be "unknown" were further randomly sampled for a new scan. For all the apps that VirusTotal did not find malicious, we further picked up a few samples for a manual analysis. Particularly, for all suspicious apps identified by the intersection analysis, we clustered them according to their shared code. Within each cluster, whenever we observed that most members were confirmed malware, we concluded that highly likely the remaining apps there are also suspicious, even if they were not flagged by VirusTotal. The common code of these apps were further inspected for suspicious activities such as information leaks. A similar approach was employed to understand the diff code extracted during the differential analysis. We manually identified the activities performed by the code and labeled it as suspicious when they could lead to damages to the user's information assets.

4.2 Effectiveness and Performance

Malware found and coverage. From our collection, MassVet reported 127,429 suspicious apps (10.93%). 10,202 of them were caught by "Diff" and the rest were discovered by "Com". These suspicious apps are from different markets: 30,552 from Google Play and 96,877 from the third-party markets, as illustrated in Table 5. We first validated these findings by uploading them to VirusTotal for a cached scan (i.e., quickly checking the apps against the checksums cached by VirusTotal),

AV Name	# of Detection	% Percentage
Ours (MassVet)	197	70.11
ESET-NOD32	171	60.85
VIPRE	136	48.40
NANO-Antivirus	120	42.70
AVware	87	30.96
Avira	79	28.11
Fortinet	71	25.27
AntiVir	60	21.35
Ikarus	60	21.35
TrendMicro-HouseCall	59	21.00
F-Prot	47	16.73
Sophos	46	16.37
McAfee	45	16.01

Table 1: The coverages of other leading AV scanners.

which came back with 91,648 confirmed cases (72%), 17,061 presumably false positives (13.38%, that is, the apps whose checksums were in the cache but not found to be malicious when they were scanned) and 13,492 unknown (10.59%, that is, the apps whose checksums were not in VirusTotal's cache). We further randomly selected 2,486 samples from the unknown set and 1,045 from the "false-positive" set, and submitted to VirusTotal again for a new scan (i.e., running all the scanners, with the most up-to-date malware signatures, on the apps). It turned out that 2,340 (94.12%) of unknown cases and 349 (33.40%) of "false positives" are actually malicious apps, according to the new analysis. This gives us a false detection rate (FDR: false positives vs. all detected) of 9.46% and a false positive rate (FPR: false positives vs. all apps analyzed) of 1%, solely based upon VirusTotal's scan results. Note that the Com step found more malware than Diff, as Diff relies on the presence of two apps of same repackaging origins in the dataset, while Com only looks for common attack payloads shared among apps. It turns out that many malicious-apps utilize same malicious SDKs, which make them easier to catch.

We further randomly sampled 40 false positives reported by the new scan for a manual validation and found that 20 of them actually are highly suspicious. Specifically, three of them load and execute suspicious code dynamically; one takes pictures stealthily; one performs sensitive operation to modify the booting sequence of other apps; seven of them get sensitive user information such as SIM card SN number and telephone number/ID; several aggressive adware turn out to add phishing plugins and app installation bars without the user's consent. The presence of these activities makes us believe that very likely they are actually zero-day malware. We have reported all of them to four Antivirus software vendors such as Norton and F-Secure for a further analysis. If all these cases are confirmed, then the FDR of MassVet could further be reduced to 4.73%.

To understand the coverage of our approach, we randomly sampled 2,700 apps from Google Play and scanned them using MassVet and the 54 scanners within VirusTotal. All together, VirusTotal detected 281 apps and among them our approach got 197 apps. The coverage of MassVet, with regard to the collective result of all 54 scanners, is 70.1%, better than what could be achieved by any individual scanner integrated within VirusTotal, including such top-of-the-line antivirus systems as NOD32 (60.8%), Trend (21.0%), Symantec (5.3%) and McAfee (16%). Most importantly, MassVet caught at least 11% malware those scanners missed. The details of the study are presented in Table 1 (top 12).

# Apps	Pre-Processing analysis	v-core database search	differential	m-core database search (Intersection)	sum
10	5.84	0.15	0.33	1.80	8.12
50	5.85	0.15	0.34	1.99	8.33
100	5.85	0.14	0.35	2.23	8.57
200	5.88	0.16	0.35	3.13	9.52
500	5.88	0.16	0.35	3.56	9.95

Table 2: Performance: "Apps" here refers to the number of concurrently submitted apps.

Vetting delay. We measured the performance of our technique, on a server with 260 GB memory, 40 cores at 2.8 GHz and 28 TB hard drives. Running on top of the Storm stream processor, our prototype was tested against 1 to 500 concurrently submitted apps. The average delay we observed is 9 seconds, from the submission of the app to the completion of the whole process on it. This vetting operation was performed against all 1.2 million apps.

Table 2 further shows the breakdown of the vetting time at different vetting stages, including preprocessing (v-core and m-core generation), search across the v-core database, the differential analysis, search over the m-core database and the intersection analysis. Overall, we show that MassVet is indeed capable of scaling to the level of real-world markets to provide a real-time vetting service.

4.3 Measurement and Findings

Over the 127,429 malicious apps detected in our study, we performed a measurement study that brings to light a few interesting observations important for understanding the seriousness of the malware threat to the Android ecosystem, as elaborated below.

Landscape. The malware we found are distributed across the world: over 35,473 from North America, 4,852 from Europe and 87,104 from Asia. In terms of the portion of malicious code within all apps, Chinese app markets take the lead with 12.90%, which is followed by US, with 8.28%. This observation points to a possible lack of regulations and proper security protection in many Chinese markets, compared with those in other countries. Even among the apps downloaded from Google Play, over 7.61% are malicious, which is different from a prior report of only 0.1% malware discovered there [15]. Note that most of the malware here has been confirmed by VirusTotal. This indicates that indeed the portion of the apps with suspicious activities on leading app stores could be higher than previously thought. De-

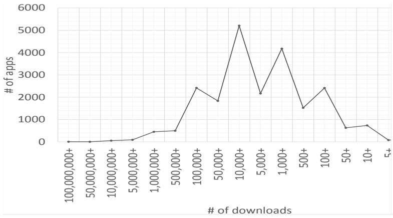


Figure 4: The distribution of downloads for malicious or suspicious apps in Google Play.

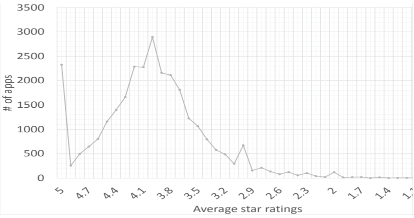


Figure 5: The distribution of rating for malicious or suspicious apps in Google Play.

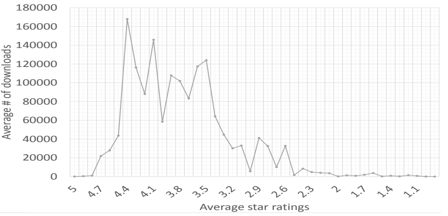


Figure 6: The distribution of average number of downloads for malicious or suspicious apps in Google Play.

tailed numbers of malicious apps are shown in Appendix (Table 5).

We observed that most scanners react slowly to the emergence of new malware. For all 91,648 malicious apps confirmed by VirusTotal, only 4.1% were alarmed by at least 25 out of 54 scanners it hosts. The results are present in Figure 7. This finding also demonstrates the capability of MassVet to capture new malicious content missed by most commercial scanners.

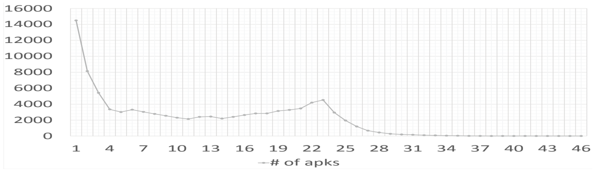


Figure 7: Number of malware detected by VirusTotal.

The impacts of those malicious apps are significant. Over 5,000 such apps have already been installed over 10,000 times each (Figure 4). Also, there are a few extremely popular ones, with the install count reaching 1 million or even more. Also, the Google-Play ratings of the suspicious APKs are high (most of them ranging from 3.6 to 4.6, Figure 5), with each being downloaded for many times (100,000 to 250,000) on average (Figure 6). This suggests that hundreds of millions of mobile devices might have already been infected.

Existing defense and disappeared apps. Apparently, Google Play indeed makes effort to mitigate the malware threat. However, our measurement study also shows the challenge of this mission. As Figure 8 illustrates, most malware we discovered were uploaded in the past 14 months. Also the more recently an app shows up, the more likely it is problematic. This indicates that Google Play continuously inspects the apps it hosts to remove the suspicious ones. For the apps that have already been there for a while, the chance is that they are quite legitimate, with only 4.5% found to be malicious. On the other hand, the newly released apps are much less trustworthy, with 10.69% of them being suspicious. Also, these malicious apps have a pretty long shelf time, as Google needs up to 14 months to remove most of them. Among the malware we discovered, 3 apps uploaded in Dec. 2010 are still there in Google Play.

Interestingly, 40 days after uploading 3,711 apps (those we asked VirusTotal to run *new scan* upon, as mentioned earlier) to VirusTotal, we found that 250 of them disappeared from Google Play. 90 days later, another 129 apps disappeared. Among the 379 disappeared apps, 54 apps (14%) were detected by VirusTotal. Apparently, Google does not run VirusTotal for its vetting but pays close attention to the new malware it finds.

We further identified 2,265 developers of the 3,711 suspicious apps, using the apps' meta data, and monitored all their apps in the follow-up 15 weeks (November 2014 to February 2015). Within this period, we observed that additional 204 apps under these developers disappeared, all of which were detected by MassVet, *due to the suspicious methods they shared with the malware we caught before that period*. The interesting part is that we did not scan these apps within VirusTotal, which indicates that it is likely that Google Play also looked into their malicious components and utilized them to check all other apps under the same developers. However, apparently, Google did not do this across the whole marketplace, because we found that other apps carrying those methods were still there on Google Play. If these apps were missed due to the cost for scanning all the apps on the Play Store, MassVet might actually be useful here: our prototype is able to compare a method across all 1.2 million apps within 0.1 second.

Another interesting finding is that we saw that some of these developers uploaded the same or similar malicious apps again after they were removed. Actually, among the 2,125 reappeared apps, 604 confirmed malware (28.4%) showed up in the Play Store *unchanged*, with the same MD5 and same names. Further, those developers also published 829 apps with the same malicious code (as that of the malware) but under different names. The fact that the apps with known malicious payloads still got slipped in suggests that Google might not pay adequate attention to even known malware.

Repackaging malware and malicious payload. Among the small set of repackaging malware captured by the differential analysis, most are from third-party stores (92.35%). Interestingly, rarely did we observe that malware authors repackaged Google Play apps

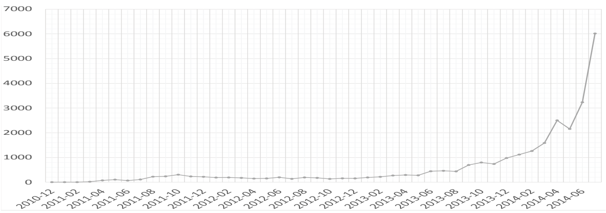


Figure 8: Number of malicious apps overtime.

and distributed them to the third-party stores in China. Instead, malware repackaging appears to be quite localized, mostly between the app stores in the same region or even on the same store. A possible explanation could be the effort that malware authors need to make on the original app so that it works for a new audience, which is certainly higher than simply repackaging the popular one in the local markets.

Figure 9 illustrates the distribution of common code across malware, as discovered from the intersection analysis. A relatively small set of methods have been reused by a large number of malicious apps. The leading one has been utilized by 9,438 Google-Play malware and by 144 suspicious apps in the third-party markets. This method turns out to be part of the library ("com/startapp") extensively used by malware. Over 98% of the apps integrating this library were flagged as malicious by VirusTotal and the rest were also found to be suspicious through our manual validation. This method sends users' fine-grained location information to a suspicious website. Similarly, all other popular methods are apparently also part of malware-building toolkits. Examples include "guohead", "purchasesdk" and "SDKUtils". The malware integrating such libraries are signed by thousands of different parties. An observation is that the use of these malicious SDKs is pretty regional: in Chinese markets, "purchasesdk" is popular, while "startapp" is widely used in the US markets. We also noticed that a number of libraries have been obfuscated. A close look at these attack SDKs shows that they are used for getting sensitive information like phone numbers, downloading files and loading code dynamically.

Signatures and identities. For each confirmed malicious app, we took a look at its "signature", that is, the public key on its X.509 certificate for verifying the integrity of the app. Some signatures have been utilized by more than 1,000 malware each: apparently, some malware authors have produced a large number of malicious apps and successfully disseminated them across different markets (Table 3). Further, when we checked the meta data for the malware discovered on Google Play, we found that a few signatures have been associated with many *identities* (e.g., the `creator` field in the metadata). Particularly, one signature has been linked to 604 identities, which indicates that the adversary might have created many accounts to distribute his app (Table 4).

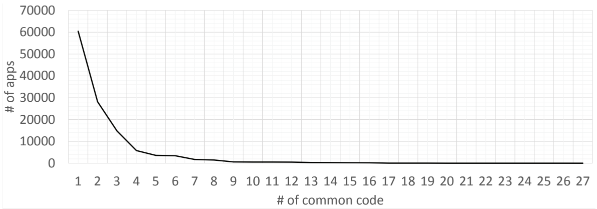


Figure 9: The distribution of common code across malware.

Signature	# of malicious apps
c673c8a5f021a5bdc5c036ee30541dde	1644
a2993eaecf1e3c2bcad4769cb79f1556	1258
3be7d6ee0dca7e8d76ec68cf0ccd3a4a	615
f8956f66b67be5490ba6ac24b5c26997	559
86c2331f1d3bb4af2e88f485ca5a4b3d	469

Table 3: Top 5 signatures used in apps.

Case studies. Among the suspicious apps MassVet reported are a set of APKs not even VirusTotal found to be malicious. We analyzed 40 samples randomly chosen from this set and concluded that 20 of them were indeed problematic through manual analysis, likely to be zero-day malware. We have reported them to 4 malware companies (F-Secure, Norton, Kaspersky, Trend Micro) for further validations. The behaviors of these apps include installing apps without user's consent, collecting user's private data (e.g., take screen shots of other apps) even though such information does not serve apps' stated functionalities, loading and executing native binary for command and control.

These apps use various techniques to evade detection. For example, some hide the suspicious functionality for weeks before starting to run it. "Durak card game" is such an game, which has been downloaded over 5,000,000 times. It was on Google Play before BBC reported it on February 4th 2015 [25]. So far, only two scanners hosted by VirusTotal can detect it. This malware disguises as warning messages when the user unlock her Android smartphone. It waits for several weeks before performing malicious activities. Its advertisements also do not show up until at least one reboot. Although Google removes "Durak card game", other apps with similar functionalities are still on the Play Store now. We also found that some malicious apps conceal their program logic inside native binaries. Some even encrypt the binaries and dynamically decrypt them for execution. Further some utilize Java reflection and other obfuscation techniques to cover their malicious code.

Signature	# of different identities
02d98ddfbcd202b13c49330182129e05	604
a2993eaecf1e3c2bcad4769cb79f1556	447
82fd3091310ce901a889676eb4531f1e	321
9187c187a43b469fa1f995833080e7c3	294
c0520c6e71446f9ebdf8047705b7bda9	145

Table 4: Top 5 signatures used by different identities.

5 Discussion

As discussed before, MassVet aims at repackaging malware, the mainstay of potentially harmful mobile apps: this is because malware authors typically cannot afford to spend a lot of time and money to build a popular app just for spreading malware, only to be forced to do this all over again once it gets caught. Our technique exploits a weakness of their business model, which relies on repackaging popular apps with a similar attack payload to keep the cost for malware distribution low. With the fundamentality of the issue and the effectiveness of the technique on such malware, our current implementation, however, is still limited, particularly when it comes to the defense against evasion.

Specifically, though simply adding the junk views connected to an existing app's view graph can affect user experience and therefore may not work well (Section 3.3), a more effective alternative is to obfuscate the links between views (calls like `StartActivity`). However, this treatment renders an app's UI structure less clear to our analyzer, which is highly suspicious, as the vast majority of apps' view graphs can be directly extracted. What we could do is to perform a dynamic analysis on such an app, using the tools like *Monkey* to explore the connections between different views. Note that the overall performance impact here can still be limited, simply because most apps submitted to an app store are legitimate and their UI structures can be statically determined.

Further, to evade the commonality analysis, the adversary could obfuscate the malicious methods. As discussed earlier (Section 3.3), this attempt itself is nontrivial, as the m-cores of those methods can only be moved significantly away from their original values through substantial changes to their CFGs each time when a legitimate app is repackaged. This can be done by adding a large amount of junk code on the CFGs. Our current implementation does not detect such an attack, since it is still no there in real-world malware code. On the other hand, further studies are certainly needed to better understand and mitigate such a threat.

Critical to the success of our DiffCom analysis is removal of legitimate libraries. As an example, we could utilize a crawler to periodically gather shared libraries and code templates from the web to update our whitelists. Further, a set of high-profile legitimate apps can be analyzed to identify the shared code missed by the crawler. What can also be leveraged here is a few unique resources in the possession of the app market. For example, it knows the account from which the apps are uploaded, even though they are signed by different certificates. It is likely that legitimate organizations are only maintaining one account and even when they do have multiple ones, they will not conceal the relations among them. Using such information, the market can find out whether two apps are actually related to identify the internal libraries they share. In general, given the fact that MassVet uses a large number of existing apps (most of which are legitimate) to vet a small set of submissions, it is at the right position to identify and remove most if not all legitimate shared code.

6 Related Work

Malicious app detection. App vetting largely relies on the techniques for detecting Android malware. Most existing approaches identify malicious apps either based upon how they look like (i.e., content-based signature) [20, 27, 21, 45, 51, 57, 19, 54, 17, 22, 4] or how they act (behavior-based signature) [11, 31, 48, 47, 42, 18, 34]. Those approaches typically rely on heavyweight static or dynamic analysis techniques, and cannot detect the unknown malware whose behavior has not been modeled a priori. MassVet is designed to address these issues by leveraging unique properties of repackaging malware. Most related to our work is PiggyApp [54], which utilizes the features (permissions, APIs, etc.) identified from a major component shared between two apps to find other apps also including this component, then clusters the rest part of these apps' code, called *piggybacked payloads*, and samples from individual clusters to manually determine whether the payloads there are indeed malicious. In contrast, MassVet *automatically* detects malware through inspecting the code diff among apps with a similar UI structure and the common methods shared between those unrelated. When it comes to the scale of our study, ANDRUBIS [26, 46] dynamically examined the operations of over 1 million apps in four years. Different from ANDRUBIS, which is an off-line analyzer for recovering detailed behavior of individual malicious apps, MassVet is meant to be a fast online scanner for identifying malware without knowing its behavior. It went through 1.2 million of apps within a short period of time.

Repackaging and code reuse detection. Related to our work is repackaging and code reuse detection [55, 21, 1, 9, 10, 41, 35, 5]. Most relevant to MassVet is the Centroids similarity comparison [7], which is also proposed for detecting code reuse. Although it is a building block for our technique, the approach itself does *not* detect malicious apps. Significant effort was made in our research to build view-graph and code analysis on top of it to achieve an accurate malware scan. Also, to defeat code obfuscation, a recent proposal leverages the similarity between repackaged apps' UIs to detect their relations [50]. However, it is too slow, requiring 11 seconds to process a pair of apps. In our research, we come up with a more effective UI comparison technique, through mapping the features of view graphs to their geometric centers, as Centroids does. This significantly improves the performance of the UI-based approach, enabling it to help vet a large number of apps in real time.

7 Conclusion

We present MassVet, an innovative malware detection technique that compares a submitted app with all other apps on a market, focusing on its diffs with those having a similar UI structure and intersections with others. Our implementation was used to analyze nearly 1.2 million apps, a scale on par with that of Google Play, and discovered 127,429 malicious apps, with 20 likely to be zero-day. The approach also achieves a higher coverage than leading anti-malware products in the market.

Acknowledgement

We thank our shepherd Adam Doupé and anonymous reviewers for their valuable comments. We also thank Dr. Sencun Zhu and Dr. Fangfang Zhang for sharing their ViewDroid code, which allows us to understand how their system works, and VirusTotal for the help in validating over 100,000 apps discovered in our study. IU authors were supported in part by the NSF 1117106, 1223477 and 1223495. Kai Chen was supported in part by NSFC 61100226, 61170281 and strategic priority research program of CAS (XDA06030600). Peng Liu was supported by NSF CCF-1320605 and ARO W911NF-09-1-0525 (MURI).

References

[1] ANDROGUARD. Reverse engineering, malware and goodware analysis of android applications ... and more. http://code.google.com/p/androguard/, 2013.

[2] APPBRAIN. Ad networks - android library statistics — appbrain.com. http://www.appbrain.com/stats/libraries/ad. (Visited on 11/11/2014).

[3] APPCELERATOR. 6 steps to great mobile apps. http://www.appcelerator.com/. 2014.

[4] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI* (2014), ACM, p. 29.

[5] BAYER, U., COMPARETTI, P. M., HLAUSCHEK, C., KRUEGEL, C., AND KIRDA, E. Scalable, behavior-based malware clustering. In *NDSS* (2009), vol. 9, Citeseer, pp. 8–11.

[6] CHEN, K. A list of shared libraries and ad libraries used in android apps. http://sites.psu.edu/kaichen/ 2014/02/20/a-list-of-shared-libraries-and-ad-libraries-used-in-android-apps.

[7] CHEN, K., LIU, P., AND ZHANG, Y. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *ICSE* (2014).

[8] CISCO. "cisco 2014 annual security report,". http://www.cisco.com/web/offer/gist_ty2_asset/ Cisco_2014_ASR.pdf, 2014.

[9] CRUSSELL, J., GIBLER, C., AND CHEN, H. Attack of the clones: Detecting cloned applications on android markets. *ESORICS* (2012), 37–54.

[10] CRUSSELL, J., GIBLER, C., AND CHEN, H. Scalable semantics-based detection of similar android applications. In *ESORICS* (2013).

[11] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI* (2010), vol. 10, pp. 1–6.

[12] ENCK, W., OCTEAU, D., MCDANIEL, P., AND CHAUDHURI, S. A study of android application security. In *USENIX security symposium* (2011), vol. 2, p. 2.

[13] ENVATOMARKET. Android notification templates library. http://codecanyon.net/item/android-notification-templates-library/5292884. 2014.

[14] ERNST, M. D., JUST, R., MILLSTEIN, S., DIETL, W. M., PERNSTEINER, S., ROESNER, F., KOSCHER, K., BARROS, P., BHORASKAR, R., HAN, S., ET AL. Collaborative verification of information flow for a high-assurance app store.

[15] F-SECURE. F-secure : Internet security for all devices. http://f-secure.com, 2014.

[16] F-SECURE. Threat report h2 2013. Tech. rep., f-secure, http://www.f-secure.com/documents/996508/1030743/ Threat_Report_H2_2013.pdf, 2014.

[17] FENG, Y., ANAND, S., DILLIG, I., AND AIKEN, A. Apposcopy: Semantics-based detection of android malware through static analysis. In *SIGSOFT FSE* (2014).

[18] GILBERT, P., CHUN, B.-G., COX, L. P., AND JUNG, J. Vision: automated security validation of mobile apps at app markets. In *Proceedings of the second international workshop on Mobile cloud computing and services* (2011), ACM, pp. 21–26.

[19] GRACE, M., ZHOU, Y., ZHANG, Q., ZOU, S., AND JIANG, X. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services* (2012), ACM, pp. 281–294.

[20] GRIFFIN, K., SCHNEIDER, S., HU, X., AND CHIUEH, T.-C. Automatic generation of string signatures for malware detection. In *Recent Advances in Intrusion Detection* (2009), Springer, pp. 101–120.

[21] HANNA, S., HUANG, L., WU, E., LI, S., CHEN, C., AND SONG, D. Juxtapp: A scalable system for detecting code reuse among android applications. In *DIMVA* (2012).

[22] HUANG, H., CHEN, K., REN, C., LIU, P., ZHU, S., AND WU, D. Towards discovering and understanding unexpected hazards in tailoring antivirus software for android. In *AsiaCCS* (2015), ACM, pp. 7–18.

[23] JING, Y., ZHAO, Z., AHN, G.-J., AND HU, H. Morpheus: automatically generating heuristics to detect android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference* (2014), ACM, pp. 216–225.

[24] KASSNER, M. Google play: Android's bouncer can be pwned. http://www.techrepublic.com/blog/it-security/-google-play-androids-bouncer-can-be-pwned/, 2012.

[25] KELION, L. Android adware 'infects millions' of phones and tablets. http://www.bbc.com/news/technology-31129797, 2015.

[26] LINDORFER, M., NEUGSCHWANDTNER, M., WEICHSELBAUM, L., FRATANTONIO, Y., VAN DER VEEN, V., AND PLATZER, C. Andrubis-1,000,000 apps later: A view on current android malware behaviors. In *Proceedings of the the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)* (2014).

[27] LINDORFER, M., VOLANIS, S., SISTO, A., NEUGSCHWANDTNER, M., ATHANASOPOULOS, E., MAGGI, F., PLATZER, C., ZANERO, S., AND IOANNIDIS, S. Andradar: Fast discovery of android applications in alternative markets. In *DIMVA* (2014).

[28] LU, L., LI, Z., WU, Z., LEE, W., AND JIANG, G. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 229–240.

[29] NETWORKX. Python package for creating and manipulating graphs and networks. https://pypi.python.org/pypi/networkx/1.9.1, 2015.

[30] Oberheide, J., and Miller, C. Dissecting the android bouncer. *SummerCon2012, New York* (2012).

[31] Rastogi, V., Chen, Y., and Enck, W. Appsplayground: Automatic security analysis of smartphone applications. In *Proceedings of the third ACM conference on Data and application security and privacy* (2013), ACM, pp. 209–220.

[32] Rastogi, V., Chen, Y., and Jiang, X. Catch me if you can: Evaluating android anti-malware against transformation attacks. *Information Forensics and Security, IEEE Transactions on 9*, 1 (2014), 99–108.

[33] reading, I. . D. Google play exploits bypass malware checks. http://www.darkreading.com/risk-management/google-play-exploits-bypass-malware-checks/d/d-id/1104730?, 6 2012.

[34] Reina, A., Fattori, A., and Cavallaro, L. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *EuroSec, April* (2013).

[35] Ren, C., Chen, K., and Liu, P. Droidmarking: resilient software watermarking for impeding android application repackaging. In *ASE* (2014), ACM, pp. 635–646.

[36] smali. An assembler/disassembler for android's dex format. http://code.google.com/p/smali/, 2013.

[37] Square, G. Dexguard. https://www.saikoa.com/dexguard, 2015.

[38] Square, G. Proguard. https://www.saikoa.com/proguard, 2015.

[39] Statista. Statista : The statistics portal. http://www.statista.com/, 2014.

[40] Storm, A. Storm, distributed and fault-tolerant realtime computation. https://storm.apache.org/.

[41] Vidas, T., and Christin, N. Sweetening android lemon markets: measuring and combating malware in application marketplaces. In *Proceedings of the third ACM conference on Data and application security and privacy* (2013), ACM, pp. 197–208.

[42] Vidas, T., Tan, J., Nahata, J., Tan, C. L., Christin, N., and Tague, P. A5: Automated analysis of adversarial android applications. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices* (2014), ACM, pp. 39–50.

[43] VirusTotal. Virustotal - free online virus, malware and url scanner. https://www.virustotal.com/, 2014.

[44] VirusTotal. Virustotal for android. https://www.virustotal.com/en/documentation/mobile-applications/, 2015.

[45] Walenstein, A., and Lakhotia, A. *The software similarity problem in malware analysis*. Internat. Begegnungs-und Forschungszentrum für Informatik, 2007.

[46] Weichselbaum, L., Neugschwandtner, M., Lindorfer, M., Fratantonio, Y., van der Veen, V., and Platzer, C. Andrubis: Android malware under the magnifying glass. *Vienna University of Technology, Tech. Rep. TRISECLAB-0414-001* (2014).

[47] Wu, C., Zhou, Y., Patel, K., Liang, Z., and Jiang, X. Airbag: Boosting smartphone resistance to malware infection. In *NDSS* (2014).

[48] Yan, L. K., and Yin, H. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *In USENIX rSecurity 12'*.

[49] Yan, P. A look at repackaged apps and their effect on the mobile threat landscape. http://blog.trendmicro.com/trendlabs-security-intelligence/a-look-into-repackaged-apps-and-its-role-in-the-mobile-threat-landscape/, 7 2014. Visited on 11/10/2014.

[50] Zhang, F., Huang, H., Zhu, S., Wu, D., and Liu, P. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 7th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec 2014). ACM* (2014).

[51] Zhang, Q., and Reeves, D. S. Metaaware: Identifying metamorphic malware. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual* (2007), IEEE, pp. 411–420.

[52] Zhang, Y., Yang, M., Xu, B., Yang, Z., Gu, G., Ning, P., Wang, X. S., and Zang, B. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 611–622.

[53] Zheng, M., Lee, P. P., and Lui, J. C. Adam: An automatic and extensible platform to stress test android anti-virus systems. In *Detection of Intrusions and Malware, and Vulnerability Assessment* (2013), pp. 82–101.

[54] Zhou, W., Zhou, Y., Grace, M., Jiang, X., and Zou, S. Fast, scalable detection of piggybacked mobile applications. In *CODASPY* (2013).

[55] Zhou, W., Zhou, Y., Jiang, X., and Ning, P. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy* (2012), ACM, pp. 317–326.

[56] Zhou, Y., and Jiang, X. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on* (2012), IEEE, pp. 95–109.

[57] Zhou, Y., Wang, Z., Zhou, W., and Jiang, X. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS* (2012).

[58] Zorz, Z. 1.2info. http://www.net-security.org/secworld.php?id=15976, 11 2013. (Visited on 11/10/2014).

8 Appendix

Appstore	# of malicious apps	# of total apps studied	Percentage	Country
Anzhi	17921	46055	38.91	China
Yidong	1088	3026	35.96	China
yy138	828	2950	28.07	China
Anfen	365	1572	23.22	China
Slideme	3285	15367	21.38	US
AndroidLeyuan	997	6053	16.47	China
gfun	17779	108736	16.35	China
16apk	4008	25714	15.59	China
Pandaapp	1577	10679	14.77	US
Lenovo	9799	68839	14.23	China
Haozhuo	1100	8052	13.66	China
Dangle	2992	22183	13.49	China
3533_world	1331	9886	13.46	China
Appchina	8396	62449	13.44	China
Wangyi	85	663	12.82	China
Youyi	408	3628	11.25	China
Nduo	20	190	10.53	China
Sogou	2414	23774	10.15	China
Huawei	148	1466	10.1	China
Yingyongbao	272	2812	9.67	China
AndroidRuanjian	198	2308	8.58	China
Anji	3467	41607	8.33	China
AndroidMarket	1997	24332	8.21	China
Opera	4852	61866	7.84	Europe
Mumayi	6129	79594	7.7	China
Google	30552	401549	7.61	US
Xiaomi	832	12139	6.85	China
others	2377	38648	6.15	China
Amazon	59	1001	5.89	US
Baidu	831	21122	3.93	China
7xiazi	898	26195	3.43	China
Liqu	394	26392	1.49	China
Gezila	30	5000	0.6	China

Table 5: App Collection & Malware in Different Markets.

You Shouldn't Collect My Secrets:
Thwarting Sensitive Keystroke Leakage in Mobile IME Apps

Jin Chen[†], Haibo Chen[†], Erick Bauman[⋆], Zhiqiang Lin[⋆], Binyu Zang[†], Haibing Guan[†]
[†]Shanghai Key Kaboratory of Scalable Computing and Systems, Shanghai Jiao Tong University
[⋆]Department of Computer Science, The University of Texas at Dallas

Abstract

IME (input method editor) apps are the primary means of interaction on mobile touch screen devices and thus are usually granted with access to a wealth of private user input. In order to understand the (in)security of mobile IME apps, this paper first performs a systematic study and uncovers that many IME apps may (intentionally or unintentionally) leak users' sensitive data to the outside world (mainly due to the incentives of improving the user's experience). To thwart the threat of sensitive information leakage while retaining the benefits of an improved user experience, this paper then proposes I-BOX, an app-transparent oblivious sandbox that minimizes sensitive input leakage by confining untrusted IME apps to predefined security policies. Several key challenges have to be addressed due to the proprietary and closed-source nature of most IME apps and the fact that an IME app can arbitrarily store and transform user input before sending it out. By designing system-level transactional execution, I-BOX works seamlessly and transparently with IME apps. Specifically, I-BOX first checkpoints an IME app's state before the first keystroke of an input, monitors and analyzes the user's input, and rolls back the state to the checkpoint if it detects the potential danger that sensitive input may be leaked. A proof of concept I-BOX prototype has been built for Android and tested with a set of popular IME apps. Experimental results show that I-BOX is able to thwart the leakage of sensitive input for untrusted IME apps, while incurring very small runtime overhead and little impact on user experience.

1 Introduction

The Problem. With large touch screens, modern mobile devices typically feature software keyboards to allow users to enter text input. This is different compared to traditional desktops where we use the hardware keyboards. These soft keyboards are known as Input Method Editor (IME) apps, and they convert users' touch events to text. Since IME apps process almost all of a user's input in mobile devices, it is critical to ensure that they are not keyloggers and they do not leak any sensitive input to the outside world.

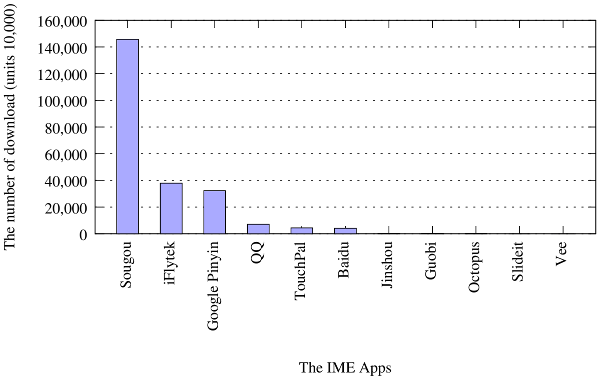


Figure 1: Download statistics of IME apps in our study.

While all mobile devices have a default IME app installed, users often demand third-party IME apps with expanded feature sets in order to gain a better user experience. This is especially common for non-Latin languages. In order to accommodate this need, mobile operating systems such as Android and iOS provide an extensible framework allowing alternate input methods. Due to the ease of making third-party IME apps and high demand for customization, there are currently thousands of IME apps in major App market like Google Play and Apple's App Store. Many of which have gained hundreds of millions downloads, as shown in Fig. 1. For instance, the Sogou IME apps has in total 1.6 billion downloads in Google Play and several third party app vendors such as 360, and Baidu. Meanwhile, a recent survey [13] found that 68.3% of smartphones in China are using third-party IME apps. This survey did not include statistics from Japan or Korea, where such apps are also very popular.

Unfortunately, despite these advantages, using a third-party IME app also brings security and privacy concerns (assume the default IME app does not have these problems). First, IME app developers have incentives to log and collect user input in order to improve the user's experience with their products, and user input is as valuable as email content, from which they can learn user's needs and push customized advertising or other business activities. Although an IME app may state a policy of not collecting certain input from a user, the policies implemented in the app may unintentionally send sensitive input outside the phone. In §2.3 we show that such a threat is real by observing the output of a popular IME app that periodically sends out user input to a remote server. In addition, we collected the network activities of a set of IME apps during a user input study and showed that they also likely send out private data. In light of this information leakage threat, the Japanese government's National Information Security Center has warned its central government ministries, agencies, research institutions and public universities to stop using IME apps offered by the search engine provider Baidu [1].

Even if a user trusts benign IME apps to properly secure private data, there is still a risk from repackaging attacks targeting benign apps. In fact, prior study has shown that around 86% of Android malware samples are repackaged from legitimate apps [49]. It is also surprisingly simple to repackage an IME app with a malicious payload, as we demonstrate in §2. Essentially, a repackaged malicious IME app is essentially a keylogger, which has been one of the most dangerous security threats for years [39]. Also, evidence has shown that IME apps are popular for attackers to inject malicious code [29].

Challenges. While it may seem trivial to detect these repackaged malicious IME apps by comparing a hash of the code with the corresponding vendor in the official market, the widespread existence of third-party markets makes such checks more difficult. It is also easy for attackers to plant repackaged malware into these markets, as is shown by the fact that a considerable amount of repackaged malware has been found in them [48].

Of further concern is the fact that it is very challenging to analyze whether even "benign" IME apps will leak any sensitive data or not. There are several reasons why detecting privacy leaks in IME apps is challenging. First, many commercial IME apps use excessive amounts of native code, which makes it very difficult to understand how they log and process user input. Second, many of the IME apps use unknown, proprietary protocols, which makes it especially hard to analyze how they collect and transform user input. Third, many of them utilize encryption, and their algorithms are also unknown. Therefore, we eventually must treat the IME apps as black boxes for current privacy-preserving techniques on mobile devices, and users must either trust them completely (and risk leaking their private data) or switch to the default IME app (and lose the improved user experience).

At a high level, it would seem that existing techniques such as taint tracking would be viable approaches to precisely tracking and containing sensitive input. For example, TaintDroid [16, 17] and its follow-up work have been shown to very effective to track sensitive input and detect when it is leaked. There will still be the following additional challenges to be overcome. First, current IME apps tend to use excessive native code in their core logic, and TaintDroid currently does not track tainted data in native code. Second, it is a well-known problem that data-flow based tracking for taint-tracking systems to capture control-based propagation. In fact, many of the keystrokes are generated through lookup tables, as reported in Panorama [46]. Third, sensitive information is often composed of a sequence of keystrokes, making it challenging to have a well-defined policy to differentiate between sensitive and non-sensitive keystrokes in TaintDroid. Therefore, we must look for new techniques.

Our approach. In this paper, we present I-BOX, an app-oblivious IME sandbox that prevents IME apps from leaking sensitive user input. In light of the opaque nature of third-party IME apps, the key idea of I-BOX is to make an IME app oblivious to sensitive input by running IME apps transactionally; I-BOX eliminates sensitive data from untrusted IME apps when there is sensitive input during this process. Specifically, I-BOX checkpoints the states of an IME app before an input transaction. It then analyzes the user's input data using a policy engine to detect whether sensitive input is flowing into an IME app. If so, I-BOX rolls back the IME app's states to the saved checkpoint, which essentially makes an IME app oblivious to what a user has entered. Otherwise, I-BOX commits the input transaction by discarding the checkpoint, which enables the IME app to leverage users' input to improve the user experience.

One key challenge faced when building I-BOX is how to make the checkpointing process efficient and consistent, which is unfortunately complicated by Android's design, especially its hybrid execution (of Java and C), multi-threading, and complex IPC mechanism (e.g., Binder). Fortunately, I-BOX addresses this challenge by leveraging the *event-driven* nature of an IME app. More specifically, we present a novel approach by creating the checkpoint at a *quiescent point*, in which its execution states are inactive. Such a design significantly simplifies many issues such as handling residual states in the local stack of native code, the Dalvik VM and IPCs.

We have implemented I-BOX based on Android 4.2.2 running on a Samsung Galaxy Nexus smartphone. Performance evaluations show that I-BOX can checkpoint and restore a set of third-party popular IME apps within a very tiny amount of time, and thus cause little impact on user experience. A security evaluation using a set of popular IME apps shows that I-BOX mitigates the leakage of sensitive input. Case studies using a popular "benign" IME app and a repackaged IME app confirm that I-BOX accurately conforms to the predefined security policies to prevent sending of sensitive input data.

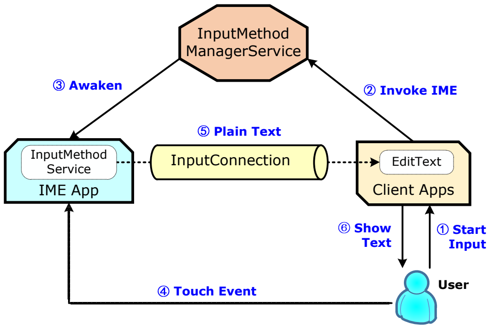


Figure 2: The workflow when using an IME app.

Contributions. In short, we make the following contributions:

- **New Problem**. This is the first attempt to systematically understand the threat caused by the leakage of private sensitive keystrokes in third-party IME apps. Our discovery shows the pervasive presence of such attacks, and the seriousness of the problem.

- **New Technique**. We introduce oblivious sandboxing for IME apps that embraces both security and usability and *quiescent points* based checkpoint/restore that significantly simplifies the design and implementation of I-Box.

- **New System**. We demonstrate a working prototype of the techniques and a set of evaluations confirming the security threat of commercial IME apps and the effectiveness of I-Box.

2 Background and Motivation

In this section, we first describe the necessary background on IME architecture in Android, and then discuss why commercial IME apps have the incentive to collect a user's data, followed by the case studies showing how IME apps can leak users' sensitive data to remote parties.

2.1 Input Method Editor

Though Android provides a default IME app for each language, many end users prefer using third-party IME apps for better user experiences, such as changing the screen layout for faster input, generating personalized phrases to provide intelligently associational input, and providing more accurate translation from keystrokes to the target languages. As a result, mobile operating systems such as Android provide an extensible IME infrastructure to allow third-party vendors to develop their own IME apps.

Figure 2 gives an overview of the involved IME components when entering text in a client app. Specifically, third-party IME apps must conform to the IME framework so that the Android Input Method Management Service (IMMS) can recognize and manage them. For example, every IME app contains a class that extends from `InputMethodService`, which helps Android recognize it as an input service and add it into the system as an IME app. When an end user clicks a textbox to invoke an IME app, Android IMMS will start the default IME activity and build an `InputConnection` between the IME app and the client app that helps the IME app to commit the user input to the client app. In particular, the IME app first gets the touch event containing the position data and translates it to meaningful characters or words based on its keyboard layout and internal logic. Then it sends the keystrokes to the client app through `InputConnection`.

The IME architecture is clean with well-defined classes. This not only significantly saves programmer's effort in developing a new IME app, but also makes it easy for attackers to locate key points of a victim IME app. For instance, our study found that simply hooking the function `BaseInputConnection.commitText` can intercept all the user's input in many IME apps. This can be done by simply searching for the keyword `BaseInputConnection.commitText` in the decompiled code to locate all of its occurrences.

2.2 Why IME Apps Collect Users' Input

Third-party IME apps usually extend the standard IME apps with lots of rich features to provide a better user experience. Such features usually require collecting users' input data to learn users' habits to allow personalizing IME apps. Further, such data may also collectively be used to improve experiences of other users, i.e., pushing phrases learned from a set of users to others. In fact, there are many features that require collecting user input data. The following lists a few of them:

- **Personal dictionary**. Commercial IME apps usually remember the words and phrases from user input to speed up follow-up input (especially for non-Latin languages) by prompting potential results when input is not finished. To achieve this, they need to maintain a personal dictionary for each user to save frequently typed or self-made words.

- **Cloud input**. As users usually have multiple devices and need to synchronize personal dictionary among them, IME apps utilize cloud-based services to store the dictionary and to synchronize the dictionary and personal settings between different devices.

  Meanwhile, some non-Latin languages such as those eastern languages differ from English in that IMEs need to translate users' keystrokes to words in those languages. To accelerate input speed, IMEs may usually need to leverage cloud services to analyze and predict users' intended words based on the current input.

In addition, for some latin-based languages, some IME apps provide a feature that leverages the current input to predict the intended phrases and adjust the layout of the soft keyboard to make the soft key of the next character close to users' current figure. To better predict user intent, some IME apps usually leverage the abundant resources in cloud to analyze and predict user input. Meanwhile, they also collect users' habits to improve the accuracy of prediction.

- **Search mediation**. Some IME apps have a new feature named "search mediation", which intercepts user input and returns some search result back to the user. However, this means that user inputs will be unrestrictedly sent to the search engine.

Note that due to the unstable network connectivity of mobile devices, almost all IME apps can work properly with and without network connections. When network is disconnected, an IME app may store current input (like frequently used phrases) for later use when the network connection is on. Besides, Android's configurable permission model indicates that an IME app usually works normally even without grants of certain permissions.

2.3 Possible Threats Posed by IME Apps

While third-party IME apps do offer useful features and better user experiences, they may unduly collect user data or be repackaged to be malicious. Next, we study the possible threats an IME app could impose.

Privacy leakage in "benign" IME apps. Conventional wisdom is to trust a respected service provider, in the hope that the provider will enforce policies in the cloud to faithfully provide user secrecy [30]. Unfortunately, this exposes users' sensitive keystrokes from two threats. First, a curious or malicious operator may stealthily steal such data [47, 41], which has been evidenced by numerous insider data theft incidents even from reputed companies [40]. Second, even reputed cloud providers provide no guarantee on the security of user data, which is evidenced by their user agreements. Hence, it is reasonable to not trust an IME app to securely protect users' data.

More specifically, a severe threat from "benign" IME apps is that they may have unduly collected user data without users' awareness. Given that we do not have their source code and they often use proprietary protocols with encryption, it thus remains opaque to end users how the IME apps really handle the sensitive input data. At a high level, since they have been collecting user data for better experiences (especially the personal dictionary and cloud input), it is highly likely that much of a user's sensitive input has been leaked to these IME providers.

To confirm our hypothesis, we conducted an experimental study by performing a man-in-the-middle attack on a popular IME app, namely TouchPal Keyboard (in version chubao 5.5.5.67049, cootek). This IME app provides multiple rich functionalities such as cloud input and a personal dictionary and has been installed more than 7.09 million times from a third-party market. By intercepting its network packages using Wireshark[1], we found that its cloud input is implemented using an HTTP POST command which carries several parameters in plain text. Therefore, we are able to see how it works without any protocol reverse engineering and packet decryption. A deep investigation revealed that these parameters include a `userid`, the `keycode` that a user just entered, and *the existing words* of the target input control that user is focusing on. This contradicts its privacy statement of "No collection of personal information that you type" in a prior statement[2], and thus poses a serious threat to user privacy.

We suspect there may be many other commercial IME apps that also leak users' sensitive input. Currently, we only used side-channel analysis [11] to analyze the packet size between the IME apps and their servers. We did notice there are notable differences in the number of packets (as reported in §5.2).

Privacy leakage in malicious IME apps. Even if all third-party IME apps did not leak any user's private data, there are still other attack vectors such as repackaging attacks. In fact, a prior study uncovers that repackaged malware samples account for 86% of all malware [49]. Moreover, there are also trojans that serve as key loggers but masquerade as IME apps [29]. Finally, IME apps may also be vulnerable to component-hijacking attacks. It has been shown that input methods have been a popular means to inject malicious code [29]. While currently we are not aware of any repackaged malicious IME apps in Android, we envision that there will be such malware given the large popularity of the official apps and the easiness of repackaging them as shown below.

To understand the repackaging threat of IME apps, we conducted an attack study by repackaging a popular commercial IME app called Baidu IME, which has been downloaded more than 100 million times in a third-party market. In this study, we repackage the IME app by inserting a malicious payload into the original program. The payload records all user input and sends them to a specific server.

While the core logic of the Baidu IME app is written

[1] http://www.wireshark.org/

[2] We noted that the newer versions of TouchPal changed their privacy statement indicating that they will collect user privacy data.

using C, the other components are written in Java which enables an easy reverse engineering of the bytecode especially with existing tools. Specifically, we used baksmali [2], a popular Dalvik disassembler to reverse `classes.dex` into an intermediate representation in the form of smali files. Then we directly modified smali code to insert our payload, which captures the text committed by the function `BaseInputConnection.commitText` and then sends the data out. A caveat in this study is that we found it would not work if we simply repackaged the app because the IME app has a checksum protection. However, the protection mechanism is rather simple, as it just calls a self-crash function when detecting repackaging. However, the self-crash function is not self-protected and thus we rewrote it to return directly to disable the protection.

We conducted our experiment in a contained environment and did not upload this repackaged IME app to any third-party Android market, but attackers can easily do this, as reported before [49, 48]. We installed this repackaged IME app on our test smartphone and all data we input through it was divulged. Our attack study shows all critical data that a user inputs will be compromised if the IME app is malicious. The popularity of third-party markets aggravates this problem, especially considering that 5% to 13% of apps are repackaged in a number of third-party markets [48].

3 Overview

The goal of I-BOX is to protect users' sensitive input, while still preserving the usability of (curious or malicious) IME apps such that users can still benefit from the rich features. One possible approach might be letting users switch to a trusted IME app when they want to type some sensitive information. While this may work for simple sensitive data like passwords, some users' sensitive input (like addresses and diseases) is scattered in a long conversation. It is cumbersome for users to constantly keep this in mind and do the switch. Another intuitive approach would be to block all network connections during user input, but doing so will negatively affect the user experience. Besides, there are also other channels like third-party content providers and external storages that an IME app may temporally store input data to be leaked later. Therefore, we have to look for new approaches.

Approach overview. As discussed, the key challenges of securely using third-party IME apps are that such apps are usually closed-source and they may do arbitrary processing and transformation of users' input data before sending it out. It is thus hard to model or predict their behavior. Hence, I-BOX instead treats an IME app as a black box and makes it *oblivious* to users' sensitive input data. To achieve this, I-BOX borrows the idea from execution transactions by running an IME app *transactionally*. Consequently, if an IME app touches users' sensitive input data, I-BOX will roll back the IME app's states to make it oblivious to what it has observed so as to address the problem where an IME app stores and transforms users' input data.

I-BOX regards the user input process as a transaction, which begins when a user starts to enter the input and ends when the input session ends. A clean snapshot of an IME app will be saved before an input transaction starts. For normal input transactions without touching sensitive input data, I-BOX will commit the IME app's state such that the IME app can use these data to improve the user experience. To prevent malicious IME apps from sending private data out during the input transaction, the network connection of the IME app will be restricted when the current transaction is marked as sensitive. When an input session ends and thus the client app has received all user input, I-BOX will abort the input transaction from the view of the IME app, by restoring the IME app's state to a most-recent checkpoint. This makes the IME app oblivious to the sensitive data it observed. Hence, even if the IME app locally saves a user's input to be sent later, the input data will be swiped during restoring.

As input data is provided in a streaming fashion by a user, there is no general way to know the input stream in advance. Because the IME app gets the input data prior to I-BOX, it would be too late to stop an IME app's leaking channels like network connection after it gets the whole input since it may have sent it out or store it locally. Hence, it is generally impossible for an approach not leaking any user input before I-BOX can determine if the current input stream is sensitive or not.

As a result, I-BOX chooses to use a combination of context-based and policy-driven approaches based on the state of the IME app, with the goal of striking a balance between user experience and privacy. For specific input such as passwords, which I-BOX can determine through input context, I-BOX can immediately know they are sensitive and thus constrains IME app's behavior (like blocking networking for the app). For general input, I-BOX uses a state-machine based policy engine to predict whether the current input transaction is sensitive. This is done continuously during the input process, where I-BOX uses the current partial input stream to determine if the next string is sensitive or not.

An architectural overview of I-BOX is presented in Figure 3. I-BOX consists of an isolated user-level policy engine that decides whether I-BOX shall commit or roll back the execution of an IME app's state. The sandbox module is implemented as a kernel module, which saves and restores the states of an IME app as needed.

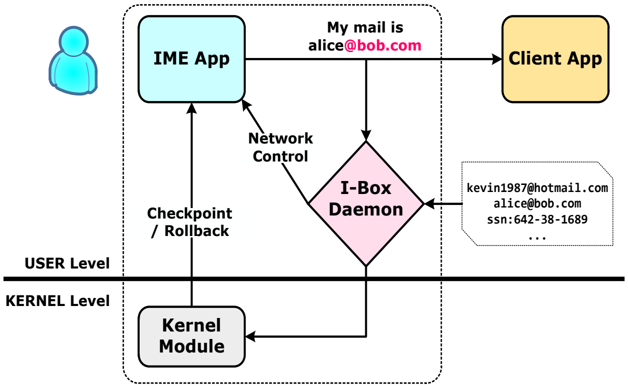


Figure 3: An Architectural Overview of I-BOX

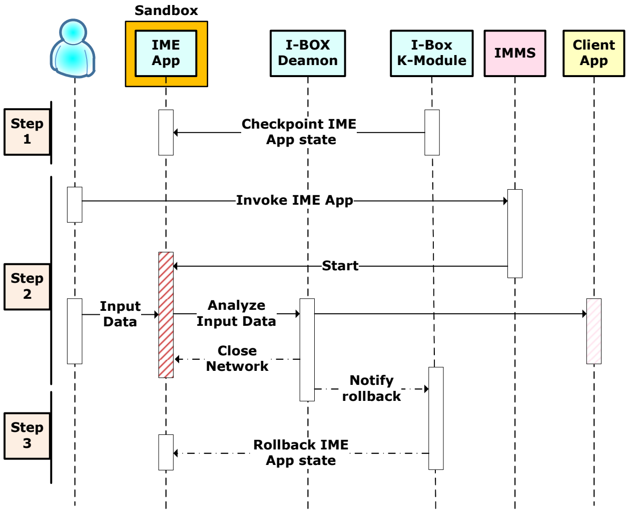


Figure 4: I-BOX work flow.

Challenges. To realize I-BOX, we are facing several challenges. In particular:

- **How to express and enforce security policies?** As users' privacy policies are usually vague, it is critical to efficiently represent users' policies such that there won't be a state explosion problem. This is especially challenging to handle for non-Latin languages as they usually require an additional layer of translation to represent them. Further, once the policies are represented, it should also be relatively easy to check the current input against the policies, which is critical to the latency of the checking.

- **How to efficiently perform the checkpoint and rollback?** As checkpoint and rollback are triggered during input, lengthy checkpoint and rollback may extend the latency of users' input. However, traditional checkpoint and rollback usually require either expensive copying of applications' states, or heavyweight recording of applications' execution. For example, prior checkpointing on server platforms takes around 600ms without copying files [28].

- **How to ensure consistency upon rollback?** By considering the user's input process as a transaction, I-BOX can ignore the implementation details of different IME apps and take them as normal processes from the kernel's viewpoint. However, there also intensive cross-layer and cross-component interactions between an IME app and the rest of the environment, like the Dalvik VM, the application framework and the client app. Further, the IME app is essentially multi-threaded. Hence, consistently checkpointing and rolling back an IME app's states while preserving the states of other components is another key technical challenge for I-BOX.

Threat model and assumptions. As third-party IME apps have the incentive to collect and send out users' data and some IME apps are even malicious (repackaged or even faked), I-BOX considers all third-party IME apps as untrusted. However, I-BOX trusts the underlying smartphone OS, including the OS kernel, system services and any process with root or system privileges. Also, we assume the user's smartphone has not been rooted such that the untrusted IME app cannot break the default security isolation between different apps, especially for system and user-level apps.

I-BOX relies on input contexts and a user's policy to distinguish private data from normal input data. It is possible that I-BOX may leak sensitive user input if the policy is incomplete or inaccurate, or the user's intent has changed after specifying the policy. Further, depending on the state machine, I-BOX may leak a prefix of some sensitive input.

I-BOX also trusts the end user and rely on her as a witness to prevent a malicious IME app from tampering with the user's input during typing. This should be easy as she can tell the difference between what she typed and what she observed from the input screen.

We consider the client app that uses the services from an IME app as trusted. While a rogue or malicious client app may also steal users' sensitive input, a malicious IME app causes more security impact than a malicious client app as it leaks all user input to all client apps (including system apps) in contrast to only input to a specific (third-party) client app. How to protect third-party client apps is out of the scope of this work and many prior efforts have intensively studied solutions to prevent information leakage from apps [24, 51, 34].

4 Design and Implementation

The work flow of how I-Box works is illustrated in Figure 4. Specifically, I-Box intercepts a user's input data by placing hooks into Android Input Method Service (IMS) and detects the sensitive data from the input stream based on the policy engine. I-Box uses both context-based and prefix-matching policies (§4.1) and enforces them using transactional execution (§4.2) to protect sensitive data such as passwords. Before diving into the details how we design and implement I-Box, we first use a running example to illustrate how it really works.

A running example. Assuming a sensitive string "IsUsenixSec2015" is being typed by a user through an IME app, I-Box first makes a checkpoint of the IME app as a clean snapshot before input. If this string is being typed to a password textbox (*context-based policy*), I-Box immediately knows that the string to type is sensitive and will restrict the IME app's behavior (such as stopping network connections). Otherwise, I-Box intercepts the characters and runs the analysis through the policy engine. After getting the characters 'I', 's', 'U', and 's', I-Box predicts that the user may be typing the sensitive string "IsUsenixSec2015" and I-Box restricts the IME app's behavior immediately to prevent it from sending further keystrokes out (*prefix-matching policy*). Afterwards, the IME app continues to accept input from users' typing and I-Box monitors the file operations of the IME app to record the files that may log the input data. After the user finishes typing, I-Box confirms that a sensitive string was typed into the IME app and restores the states of the IME app with the checkpoint to clean the sensitive string out.

4.1 Policy Engine

The policy engine of I-Box separates sensitive input from normal input such that different policies can be applied to different types of data. I-Box uses both context-based and prefix matching strategies to derive policies, with the first strategy having higher priority.

Context-based policy. We first provide an automated approach to deriving which input would be sensitive based on the type of the input and execution context of an app. Specifically, Android uses text fields to help the user type text into client apps. Text fields can have different input types, such as numbers, dates, passwords, or email addresses. In fact, the type information of text fields in the client app has been used to help an IME app to optimize its layout for frequently used characters. I-Box also leverages the type information of the text fields to decide whether the input is sensitive or not, and passwords and email addresses are by default sensitive. In addition, based on the user defined per-client app policy (e.g., an IME app is providing services to a banking application), I-Box will automatically treat all the input consumed by a sensitive app according to context [44] as sensitive.

Prefix-matching policy. For general input streams, I-Box leverages prefix matching to distinguish which input stream is sensitive or not. One challenge for defining policies for I-Box is that IME apps may need to handle multiple languages, including both Latin languages and non-Latin languages. For non-Latin languages, I-Box can only get the text in the target languages after an IME app has translated the keystrokes for the corresponding text. Hence, it is not viable to simply use keystrokes to represent the current input. To address this problem, I-Box instead uses the UTF-8 (8-bit Unicode Transformation Format) of the translated keystrokes to represent current keystrokes as well as those in the policy engine.

As there may eventually be multiple data instances that should be considered as sensitive, I-Box uses a trie-like structure to maintain which data should be considered as sensitive. A trie-like structure is very space-efficient for data with a common prefix and is very efficient for look-up. I-Box maintains a global trie structure to represent the global policy. I-Box may also provide an application-specific trie structure if an end user demands more strict policy. During a query, I-Box queries the global and application-specific trie structures in parallel but prefers application-specific policies over the global one.

While much of the sensitive data like contacts and cookies can be automatically translated to the trie structure as the default policies, I-Box also allows end users to use regular-expressions when they manually specify the policy. For example, user may define "abc*" to indicate any word starting with "abc" as sensitive input. Associated with the regular expression, there is also an *acceptable disclosure rate* (ADR), which defines how many characters can be exposed in an input stream. The larger the ADR, the more information may be leaked but the more chances are allowed for cloud assistance. Using regular expression is easy for experienced users to specify sensitive input, as it does not require them to fully remember all such sensitive data and thus matches users' ambiguous and incomplete memory. This also avoids asking users to input full secrets to I-Box. Alternatively, average users may also specify full secret names (i.e., a special case of regular expression) to I-Box.

I-Box provides a simple script to add such regular-expressions to the trie-like structure and report any conflicts if they occur. For example, for a sensitive string of 15 characters (such as 'IsUsenixSec2015') and an ADR of 0.2, I-Box will restrict an IME app's behavior when the first three characters ('IsU') are typed. I-BOX runs the trie-structure as a state machine to predict the input stream by matching the typed characters with the trie structures. Since any substring in the input data may be sensitive, I-BOX needs to check all of them. To speed up this process, I-BOX searches all possible substrings when a new character is typed. Intermediate states are maintained so that only new characters need to be handled instead of new substrings constructed by the character.

Note that currently I-BOX directly searches over the plain text of the policy file and relies on the Android permission system to protect it for simplicity. This can be further enhanced by encrypting the policy file and using regular expressions to search over the encrypted file, which was shown to have small runtime and space overhead [36].

Prefix-substitution attacks. At first glance, the prefix-matching policy used by I-BOX would appear to be vulnerable to a prefix-substitution attack by a malicious IME App. Specifically, a malicious IME app might first replace the prefix of a typed string with a non-sensitive one so that I-BOX wouldn't recognize this prefix and thus no oblivious sandbox would be applied for this input session. Fortunately, we note that users, the ultimate *witness*, would immediately notice this by observing the difference between what they typed and what was displayed on the screen.

Note that as I-BOX monitors all keystrokes sent from IME apps to user apps, I-BOX will adjust the state machine accordingly for any cursor movement and special characters like deletion. This can detect the case where a malicious IME app stealthily moves the cursor to deceive I-BOX on the input sensitivity.

Overall, I-BOX requires users' awareness of what she types from what she observes to detect malicious behavior from an IME app. If a user does not pay enough attention to the input process, a malicious IME app may still have the chance to fool I-BOX about the sensitivity of the input streams.

4.2 Enabling Transactional Execution

To enable transactional execution of an IME app, I-BOX needs to provide a checkpoint and rollback mechanism. The key challenges here lie in how to provide *low-latency* and ensure *consistency*, which are made especially difficult by Android's unique design. For example, Android uses a Dalvik virtual machine (VM) to run the Java code of the IME app, which interacts intensively with the application framework. Further, the native code of an IME app also interacts with the Dalvik VM through the Java Native Interface (JNI). Finally, Android intensively uses Binder, a complex IPC mechanism for communication among isolated apps. Such hybrid execution and complex communication make it hard to efficiently and consistently checkpoint the states of an IME app.

I-BOX addresses the above challenges by leveraging a set of *quiescent points*. A quiescent point is a point such that all threads of an application have stopped execution and there are no pending states and requests to be processed. Doing checkpointing at quiescent points frees I-BOX from handling a number of subtle states like residual states in stack or other communication peers. Further, it also requires less states to be checkpointed. Finally, when I-BOX rolls back the states of an IME app, the states can be restored consistently without having to deal with some subtle residual states in other apps.

In the following, we describe in greater detail how we choose the quiescent points (§4.2.1), how I-BOX performs the checkpoint and restore of the local states of an IME app (§4.2.2), and how I-BOX handles interactions of an IME app with others through IPCs (§4.2.3).

4.2.1 Quiescent Points

Our key observation is that an IME app is essentially an event-driven app that provides services to the client app. Consequently, it shall be usually in a quiescent point when a user is not typing, as no event will be delivered to the IME app at that time. At this state, the IME app's states are stable and consistent. Thus, I-BOX can be relaxed from handling a lot of complex and subtle local states. To achieve this, I-BOX first checks if an IME app is in a quiescent point by checking the process and thread states (sleeping or not) and the IPC states. The checking result is very likely to be true for most cases. Even if the IME app refuses to cooperate with I-BOX and keeps itself busy, I-BOX can first wait a short time and then enforce a quiescent point by blocking new requests and then forcing the IME app to sleep to do the checkpoint. Here, a non-cooperative IME app could also be a sign of being malicious. However, we never encountered this case as the IME apps we tested always conform to Android IME architecture. Even if so, I-BOX may always roll back the IME app to a clean state checkpointed early.

4.2.2 Checkpointing and Restoring Local States

Since data typed by a user can be stored into any place of the IME app in any form, it requires that all process states restore in order to wipe out any sensitive data. The traditional way of doing checkpoints is copying all related process states into storages, which is very heavyweight and would incur long latency. As the main purpose of I-BOX's checkpoint is to either rollback or discard later, I-BOX chooses a lightweight approach to checkpointing, which creates a shadow process and then tracks all later changes by using copy-on-write (COW) features provided by Linux.

Saving and restoring file states. As typical IME apps usually only modify a small amount of files during one input transaction, I-BOX currently records and copies such files during checkpointing and restores them during rollback for simplicity. Another option is using a COW file system like Btrfs or ext3cow to avoid copying. This requires replacing Android's file system with one with a COW feature, which will be our future work.

Android provides several options to save persistent application data. Based on the position where the data is stored, we can divide these options into two categories: internal and external storage. Every Android app will be assigned with a private directory in the internal storage to store files and data. By default, data saved to the internal storage are private to an app and other apps cannot access them (nor can the user). I-BOX just copies all files in the IME app's private directory and then restores the files modified during the input transaction upon rollback. Since there are usually only a small number of files in the internal storage for an IME app and the modified ones are even less, the time cost is negligible.

For external storage, any Android apps with proper permissions (e.g., `android.permission.WRITE_EXTERNAL_STORAGE`) can access the whole external storage. It would be very lengthy if I-BOX scanned the whole external storage to find the modified files. Hence, I-BOX records all the files modified by the IME app during the input transaction and then restores them as needed. Specifically, once I-BOX detects the IME app tries to write some data into a file, it duplicates the file for subsequent restoring.

Note that as the checkpointed files are created by I-BOX, which runs as a system process, the files are with system privilege and thus cannot be read/written by the IME app itself. This ensures that an IME app cannot first save sensitive key logs into such files and later read them out. Actually, I-BOX also removes the checkpointed files after rolling back an IME app.

Saving and restoring memory states. Memory states include the IME app process's data in memory and process-related metadata maintained by the OS kernel (i.e., Linux). Linux uses a lot of data structures to manage a process and maintain its state, such as `task_struct`, `thread_info` and others. I-BOX relies on a kernel module to save and restore such data structures. Specifically, this module maintains a shadow process in the kernel to store the data of each running IME app. The shadow process duplicates the process states of the original IME app by copying the metadata of the IME app into its own `task_struct` but with some modifications for consistency. For example, it has its own kernel stack and redirects the stack pointer in the `task_struct` to its own one, although the content on the stack is the same as in the original IME app. For independent states like process ID or kernel stack, I-BOX just copies the data into a buffer and writes them back later. As for other states connected with other processes or other events like a pipe or waitqueue, I-BOX needs to record the states and the relationships so that it can recover it correctly later. Besides this, I-BOX also needs to save the process memory. Instead of really copying the memory pages, I-BOX simply creates a shadow page table that shares the memory with the target IME app process and marks the page table of the target process as COW. This omits lots of unnecessary page copying since most pages will not be modified during the input transaction and it just needs to switch the page table root to restore the memory, which is very fast. This helps reduce the stop-time of each IME app process when I-BOX tries to do checkpoint and restore.

Multi-thread rollback. Most Android applications run in the Dalvik virtual machine and have multiple threads for different purposes. Besides the main thread for UI and the core logic of the IME app, there are about another 10 threads for garbage collection, event handling, Binder IPC, and so on. To roll back the process states of an IME app correctly, I-BOX needs to deal with such threads properly. Linux assigns `task_struct` to a thread just like a process to maintain its state and groups all threads belonging to one process together through a list. So I-BOX saves each thread with a separated shadow process and groups these processes together through a list to maintain their parent-child relationships just like the original one. The sharing resources between threads will be duplicated too. For example, I-BOX will save the pipe states between two threads and restore it later.

4.2.3 Handling IPCs

One major challenge I-BOX faces in checkpoint and rollback is how to deal with the IPC states of an IME app process. An IPC involves multiple processes or even multiple machines, but I-BOX can only control one end in the communication. One potential problem is that the other side of an IPC may wait for a reply that will never be sent, since the IME app process has forgotten this request after rollback. Another serious problem is that the client app may communicate with an inactive IPC that has been erased from the IME app process due to rollback.

As a result, I-BOX needs to find proper timing to do checkpoint and rollback such that the consistency of an IPC is not violated. Proper timing requires several conditions. First, there should not be any data in transmission between two processes; otherwise it will lead to a corrupted request with incorrect semantics. Second, there should be no pending IPC requests. This means an IME app shall wait for all replies before doing checkpoint and ensure that no request is pending to the process before rollback. Fortunately, it is not hard for I-BOX to find suitable timing because I-BOX only does checkpoint and rollback when a user does not input. In most cases, the IME app processes should be sleeping at that point. If not, we can safely enforce it without disturbing other client apps since the user is not typing.

Inter-threads IPC. Linux provides a set of IPC mechanisms such as pipe, socket, and shared memory. Android inherits such mechanisms but only uses them as a method for communication between threads within a single process. Hence, I-BOX can control both ends of these inter-threads IPC, which avoids the inconsistency issues due to unilateral actions. For example, the two communicating parties of a pipe in a single process have a pair of pipe `fd`; the OS kernel allocates a buffer for them to pass the message. To restore the pipe correctly, I-BOX just keeps a record of current pipe status and its buffered data, then restores it as needed. There is no restriction on the timing for checkpoint and rollback. Other IPCs within the same process are done similarly to this.

Android Binder. Android heavily uses its own IPC mechanism: *Binder*, which helps the Android permission system to provide access control to Android services and resources. By mapping kernel memory into user space, Binder IPC only requires one data copy for one transmission, i.e., from the sender's user space to the kernel buffer of the Binder driver. Then the receiver can directly read the data from its read-only user space mapping, which is more performance-friendly. There are two issues I-BOX needs to take care for a consistent restore of the Binder. More specifically:

- **Reference counting for Binder proxies**. An Android app uses Binder proxies (e.g., BBinder, BpBinder) as the reference to remote processes instead of simple file descriptors. The Binder driver in the kernel needs to manage the reference counter for such proxies so that it can know whether a binder instance is useless or not. I-BOX needs to track and record modifications to references to Binder proxies so that it can keep the consistency of the reference counters.
- **Conversation between the Binder request and response**. I-BOX also needs to keep the conversation between the Binder transaction request and response. As an Android service provider, an IME app process will accept a Binder transaction request from the client app and it will send back the transaction response after disposing the request. To achieve this, I-BOX tracks the transaction request and response to find a right timing when all requests have been handled. It is not hard to find such a point because usually I-BOX tries to do checkpoint or rollback when IME is idle without new requests.

Content Provider. An IME app may also interact with both third-party and system content providers. For example, our analysis with TouchPal IME app reveals that this app accesses third-party content providers like `content://com.tencent.mm.sdk.plugin.provider/sharedpref` and `content://com.facebook.katana.provider.AttributionIdProvider`; our analysis with Guobi IME app shows that this app accesses `content://com.iflytek.speechcloud.providers.LocalResourceProvider` and `content://com.tencent.mm.sdk.plugin.provider/sharedpref`. TouchPal accesses the system content provider like `content://sms/inbox` and both TouchPal and Guobi access `content://telephony/carriers/preferapn`. In Android, all requests to content providers are issued through the Binder mechanism, we rely on the Binder mechanism to detect a quiescent point. Fortunately, we note that accesses to content providers are request-oriented and thus connection-less. Thus, there is no request on-the-fly and thus I-BOX can checkpoint such states accordingly.

Network. Different from Binder, the network driver does not expose any semantic information to an upper layer's connections. Hence, it seems hard to maintain the consistency of request and response between an IME app process and its cloud-based server. Fortunately, there are two observations that help relax the strict consistency requirement. First, network connections between an IME app and the cloud-based server, like fetching the words by sending the keystrokes, synchronizing the user's library, and downloading news or advertisements, are usually stateless and non-transactional; a redo operation does not cause any consistency issues. Second, network connections during input transactions are mostly short-time synchronized requests that are finished when input is done; hence they will not be affected by rollback.

Lessons Learned. While it is generally hard to checkpoint a complex app like an IME app, the event-driven nature of I-BOX greatly helps simplify the design and implementation of I-BOX. By leveraging a quiescence-point based approach and conduct checkpointing at the time at which an IME app likely to be quiescent (e.g., before an input session start), I-BOX enjoys both less implementation complexity and runtime overhead.

4.3 Restricting IME Apps' Behavior

When I-BOX detects a sensitive input session, it needs to restrict an IME app's behavior such that no sensitive data should be leaked during this process. A malicious IME app may leverage various means to store and transform the data during this process. For example, it may directly send input data to the network, or store the input data to a content provider to be restored and sent out later. To this end, I-BOX needs to restrict an IME app's behavior to stop such channels for a sensitive input stream.

I-BOX constrains an IME app from using network and accesses to content provider and services during a sensitive input session. Specifically, during a sensitive input session, I-BOX only grants an IME app with read accesses (like query) to such content providers and services. This is done by interposing the *binder_transaction* and acts according to the access types from the transaction code (i.e., query, insert, update or delete).

One potential issue would be that the IME app may not function correctly without such accesses. Fortunately, most Android apps (including IME apps) are designed to work gracefully with different permissions, due to the fact that the user may grant different permissions and an IME app may work without network accesses. As a result, it is non-intrusive to dynamically deprive the IME app from certain accesses as evidenced by prior research on dynamic permissions on Android [32]. After a rollback, as all residual states inside an IME app have been cleaned, any pending actions like insertion or deletion will not cleared as if they never happen. Thus, there won't be any confusions to the content provider and services.

5 EVALUATION

We have implemented I-BOX based on Android 4.2.2 and Linux kernel OMAP 3.0.72. It consists of two main parts: i) a user-level modification of the Android application framework to insert the I-BOX policy engine and network control module; ii) a kernel module to handle checkpoints and rollback of IME apps.

Experimental Setup. All of our experiments were performed on a Samsung Galaxy Nexus smartphone with a 1.2 GHz TI OMAP4460 CPU, a 1GB memory and 16GB internal storage. We evaluate I-BOX using 11 popular IME apps to measure the performance overhead of I-BOX. The 11 IME apps (as shown in the first column of Table 1) are ranked among the highest in popularity in a large third-party market[3]. Many of these IME apps have been installed more than millions of times (Figure 1). In our testing, we set the security policies to include all contacts in the phone and all commonly used accounts and passwords. This forms a trie containing around 400 words.

5.1 Performance Evaluation

The time overhead of I-BOX comes from three parts: (1) time to find the quiescent points; (2) time to perform memory checkpoint and rollback, and (3) time to perform file save and restore. To measure the performance overhead, we asked a volunteer with an average typing speed of about 100 characters per minute to enter a 10 word paragraph in an SMS app using the tested IME apps. We did not use an automation tool like an Android Monkey as it cannot handle the complex UI interface of these IME apps.

Latency. As shown in Table 1, the time to find a quiescent point is very small (less than 14ms). This confirms our observation that it is very easy and fast to find or force a quiescent point to do checkpoint and rollback on an IME app. The time of saving and restoring an IME app's memory state is also very small (less than 29ms) since I-BOX does not really copy the whole memory but just mark them as COW. Based on the files touched by the IME app process during the typing, I-BOX needs to restore a few files to prevent the IME app from concealing the secret inside files. Hence, the time for file save and restore is a little bit lengthy (60 ms), which can further be improved by using a copy-on-write file system. In total, the maximum total time to do a checkpoint (including finding a quiescent point) is less than 103ms (14ms + 29ms + 60ms). In contrast, the world record of texting is typing a complicated 25 word message (159 characters) in 25.94 seconds [5], which corresponding to 163 ms/character and 1.0376 second/word. Hence, the time to do checkpointing is very small compared to user typing. As the time to search the trie is negligible, we didn't report it here.

Power. To measure the power overhead incurred by I-BOX, we used the TouchPal IME to input an article and its non-Latin translation to a text-note app called Catch and count its power status. The total input process spans 30 minutes for both unmodified Android and I-BOX-capable Android. We found that in both cases the power dropped from 100% to 99%, whose differences were indistinguishable. This is probably because the IME app is not power-hungry and the additional power consumed by I-BOX was evened by the reduced network transmissions, which is thus hard to be distinguished without a highly accurate power meter. In our future, we plan to further characterize the power consumption using an accurate power meter.

5.2 Security Evaluation

Here, we evaluate whether I-BOX indeed has mitigated the leakage of a user's sensitive keystrokes. We still use the IME apps in our performance testing, along with a

[3] http://www.wandoujia.com/

	Quiescent		Memory		File	
IME app	C (ms)	R (ms)	C (ms)	R (μs)	C (ms)	R (ms)
Sogou	13.3	14.1	22.8	91	30	30
Baidu	8.2	11.1	22.6	275	40	40
QQ	12	11.8	24.3	31	60	30
Pinyin	11.8	12	20.8	122	10	10
Vee	5.9	10.3	0.022	61	20	20
Guobi	7.4	9.5	25.5	61	10	10
Octopus	11.4	11	28.9	245	30	20
iFlytek	4.6	9.7	13.9	92	10	10
Slideit	13.2	15.2	13.5	152	20	30
Jinshou	3.1	6.5	28	91	60	50
TouchPal	7.8	13.3	22.1	183	30	30
Baidu*	3.3	10.9	9	61	30	40
Average	8.4	11.3	22.5	140	28.3	26.7

Table 1: Time overhead for finding a quiescent point, doing checkpoint (C) and rollback (R).

repackaged malicious IME app (described in §2.3), to evaluate its effectiveness. According to the accessibility of these IME apps, we conducted three sets of experiments to determine effectiveness: black-box testing, gray-box testing, and white-box testing.

5.2.1 *Black-box Testing*

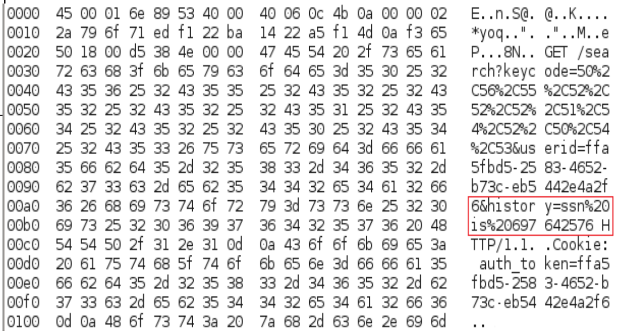
Since most of the IME apps use proprietary unknown protocols with unknown encryptions, we cannot directly trace the network packets to confirm our effectiveness. Therefore, we take a black-box approach to approximating our result. That is, instead of inspecting the packet contents, we inspect the packet differences sent by the IME-apps with I-BOX and without I-BOX, within an identical experiment setup and time window.

In particular, we ran all these apps using a two-minute time window, and we typed around 30 non-Latin words with "aa@usenix.org" as the sensitive word and then observed the packet differences using the Wireshark tool. Usually, these IME apps will send some packages out when a user types something that triggers the cloud input function. Interestingly, we found 6 out of the 11 tested apps have a different number of packages, as shown in Table 2. With I-BOX being enabled, there are less packages to be sent out compared to normal ones. This is because I-BOX controls the network of the target IME app when it detects sensitive input data and prevents the target IME app from leaking the data out.

While such side-channel based black-box testing cannot fully confirm that we have prevented all leaks, we believe it is highly likely that I-BOX has stopped them, even for the other 5 apps that we did not observe package differences for. (It is highly likely that these IME apps have buffered the input with the intent to send the data out later. However, our oblivious sandboxing mechanism will clear the buffered sensitive data).

IME app	w/o I-BOX	w/ I-BOX
Baidu	17	6
Sogou	44	30
QQ	37	20
Octopus	32	16
TouchPal	70	28
Baidu*	30	18

Table 2: #packages observed for the testing apps.



```
0000  45 00 01 6e 89 53 40 00  40 06 0c 4b 0a 00 00 02   E..n.S@. @..K....
0010  2a 79 6f 71 ed f1 22 ba  14 22 a5 f1 4d 0a f3 65   *yoq..". ."..M..e
0020  50 18 00 d5 38 4e 00 00  47 45 54 20 2f 73 65 61   P...8N.. GET /sea
0030  72 63 68 3f 6b 65 79 63  6f 64 65 3d 35 30 25 32   rch?keyc ode=50%2
0040  43 35 36 25 32 43 35 35  25 32 43 35 32 25 32 43   C56%2C55 %2C52%2C
0050  35 32 25 32 43 35 32 25  32 43 35 31 25 32 43 35   52%2C52% 2C51%2C5
0060  34 25 32 43 35 32 25 32  43 35 30 25 32 43 35 34   4%2C52%2 C50%2C54
0070  25 32 43 35 33 26 75 73  65 72 69 64 3d 66 66 61   %2C53&us erid=ffa
0080  35 66 62 64 35 2d 32 35  38 33 2d 34 36 35 32 2d   5fbd5-25 83-4652-
0090  62 37 33 63 2d 65 62 35  34 34 32 65 34 61 32 66   b73c-eb5 442e4a2f
00a0  36 26 68 69 73 74 6f 72  79 3d 73 73 6e 25 32 30   6&histor y=ssn%20
00b0  69 73 25 32 30 36 39 37  36 34 32 35 37 36 20 48   is%20697 642576 H
00c0  54 54 50 2f 31 2e 31 0d  0a 43 6f 6f 6b 69 65 3a   TTP/1.1. .Cookie:
00d0  20 61 75 74 68 5f 74 6f  6b 65 6e 3d 66 66 61 35    auth_to ken=ffa5
00e0  66 62 64 35 2d 32 35 38  33 2d 34 36 35 32 2d 62   fbd5-258 3-4652-b
00f0  37 33 63 2d 65 62 35 34  34 32 65 34 61 32 66 36   73c-eb54 42e4a2f6
0100  0d 0a 48 6f 73 74 3a 20  7a 68 2d 63 6e 2e 69 6d   ..
```

Figure 5: Hexdump of the traced Touchpal package. The leaked SSN is highlighted.

5.2.2 *Gray-box Testing*

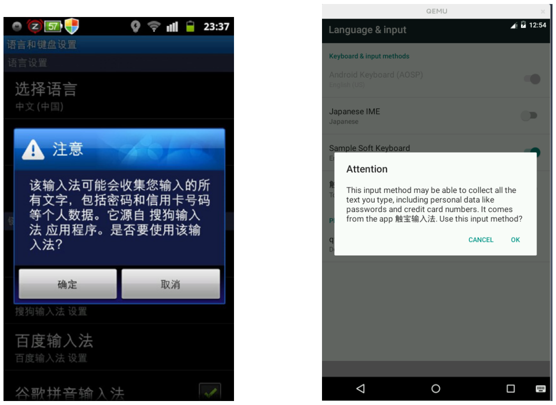
Among these 11 IME apps, we are able to observe the packet payload of TouchPal (as in discussed in §2.3) because it uses a plain-text protocol. Therefore, we conducted gray-box testing to confirm I-BOX indeed mitigated the privacy leakage. In this experiment, we open a client "SMS" app to send a short message to one friend with a social security number (SSN), which is private and sensitive by default. The text to send is a mixture of both Latin and non-Latin languages, as well as the number. Cloud input functionality will be triggered in this case.

Interestingly, without I-BOX's protection, we found that Touchpal uploaded not only the keycodes the user typed as arguments of cloud input, but also the text message before the current input cursor that includes the sensitive social security number to the cloud through an HTTP POST method. We intercepted this packet using a man-in-the-middle attack. Part of the packet is displayed in Figure 5. However, with I-BOX's protection, we found that I-BOX successfully detected the critical number and shutdown its network to stop the leakage of data, and we did not observe any network trace.

We also studied the privacy warnings generated by Android on which data an IME may collect. Figure 6 shows that Android generates privacy warnings for two popular IME apps, Sogou and TouchPal, indicating that they may collect users' passwords, credit card number, etc. This further confirms our conclusion that they collect users' privacy data.

Apps	Without I-BOX	With I-BOX
SMS (phone number)	`6204562244`	`62045`
SMS (message)	`Let’s meet tomorrow noon at room 302`	`Let’s meet tomorrow noon at room 302`
Instagram (account)	`thisisfortest@gmail.com`	`thisisf`
Instagram (password)	`fakepassword`	
Facebook (account)	`thisisfortest@gmail.com`	`thisisf`
Facebook (password)	`dontbelieveit`	
Alipay	`nomoney@yahoo.com`	`nomo`
Gmail	`tosomeone@hotmail.com`	`tosom`
Google Play	`Ingress`	`Ingress`
browser	`How much is this PS3?`	`How much is this PS3?`

Table 3: Evaluation result w/ repackaged Baidu IME using different client apps.



注意

该输入法可能会收集您输入的所有文字，包括密码和信用卡号码等个人数据。它源自 搜狗输入法 应用程序。是否要使用该输入法?

(a) Sogou IME App (in Chinese)

Attention

This input method may be able to collect all the text you type, including personal data like passwords and credit card numbers. It comes from the app 触宝输入法. Use this input method?

(b) TouchPal IME App (in English)

Figure 6: Privacy Warning by Android for two popular IME apps. The left is shown in Chinese and the right is shown in English; the essential meanings are the same.

5.2.3 *White-box Testing*

As discussed in §2.3, we repackaged a very popular Baidu IME app to log all of the user input data and send them out to a malicious server we controlled. Hence, this repackaged IME app is essentially a keylogger. We were able to perform white-box testing by inspecting the packet payloads and confirming them with the source code of our malicious payload. We installed this IME app on our test phone and then used this phone to enter some user-defined private sensitive data with different client apps ranging from SMS, Facebook, and Gmail, etc. Table 3 shows the data we collected at the server side with and without I-BOX’s protection.

From this table we can clearly observe that without I-BOX, the malicious IME app will steal all the data that a user enters. Consequently, all sensitive data has been leaked out; with I-BOX, it automatically blocks the network connection so that the server cannot receive any complete sensitive information. For instance, for passwords, the malicious server cannot receive anything as shown in the Instagram and Facebook case. As I-BOX shuts down the malicious IME app’s network when it finds character sequences that have matched part of the sensitive phrase in our security policy, the server side can only receive the parts of the typed characters. For example, when a user tries to type her Facebook account *thisisfortest@gmail.com*, the server side can only receive a part of it, i.e. *thisisf*[4]. While partial sensitive input is still being leaked, we believe it is still hard for attackers to guess the original message.

5.3 Users Experience

One principal goal of I-BOX is to limit the negative influence on an end user’s experience as little as possible. To evaluate this, we tested latency by determining how an end user would feel when typing characters on devices protected by I-BOX. For this, we invited a dozen students (6 undergraduate and 6 master students) in our Lab to install I-BOX on their phones, and asked them to use our system and provide us with feedback. By default, I-BOX uses the context-based policy and derives all sensitive data from the contacts and cookies. Two of them also tried to input their girl-friend’s names and birth dates into I-BOX.

To our pleasure, none of the users complained of any latency imposed by our system. As shown in Table 4, there is only 0.4 milliseconds (ms) overhead per character imposed by our policy checking. While network shutdown takes about 180 ms, it is not executed per word and is instead triggered only when certain sensitive words are going to be formed. Therefore, the additional overhead added by I-BOX cannot be detected by end users. This is because the typing speed for a normal user is 625ms per character, and the world fast record is 160 ms per character, as shown in Table 4.

One complaint we received so far is that the users now need to manually type their account instead of using the automation features provided by the IME apps. We believe this is worthwhile for better privacy protection. Another complaint is that they need to specify their additional secrets manually; this will motivate us to design better UI interface in our future work.

[4] Note that we regard the sequence after @ as one character because an attacker can guess the rest by the first character most of the time.

Policy Checking	0.4ms/char
Network Shutdown	180ms
Checkpoint/Restore	103ms
Guinness World Records of fastest texter	160ms/char
normal user speed	625ms/char

Table 4: Statistics regarding the usage latency of I-BOX.

6 DISCUSSIONS AND LIMITATIONS

While I-BOX has made a first step to mitigate keystroke leakage against untrusted IME apps, there are still a number of limitations in its design and implementation.

Side-channel attacks It has been viable to use side channels to infer some keystroke information [9, 4]. I-BOX currently cannot prevent such side channel attacks. However, such threats are usually less severe than those of malicious IME apps, which can accurately observe all user input. We leave it as our future work to address issues related to the side-channel leakages.

Colluding malware As I-BOX currently only runs an IME app inside in a sandbox transactionally, it is still possible that an IME app could collude with another malware to leak information (i.e., the colluding attack [8]). For example, an IME app could first save the user input in a local file, and inform a colluding malware to read the file when the transaction has not been rolled back and then divulge the input. This essentially violates the policies of I-BOX. However, it is challenging for sandboxing to reliably prevent this, as studied by TxBox [25].

Security of I-BOX Any new security tools may bring new security implications as they usually touch security-sensitive data and I-BOX is of no exception. As I-BOX can essentially touch all users' sensitive data, it is essentially a key logger as well. Yet, I-BOX is much simpler than close-sourced proprietary IME apps (1,700 LOCs vs. hundreds of thousands LOCs). Regarding whether to trust I-BOX or other IME apps, third-party agents need to only audit the code of I-BOX instead of using gray-box based approaches to auditing the behavior of dozens of third-party IME apps. Meanwhile, I-BOX is completely a local service and will not send any private data out of the phone.

Permission Attacks As I-BOX's security is based on Android permission systems, it cannot defend against attacks against the permissions like component hijacking attacks and confused deputy attacks [23]. We consider this out of the scope of this paper; actually there have been a number of prior systems that statically and dynamically detect and prevent such attacks (e.g., [12, 43]). Actually, Android has significantly improved its permission systems since version 4.2 [3].

Voice input Currently we limit input data protection to handwriting input and keystroke input and do not consider voice input as it does not have keystrokes. Yet, users usually use dedicated system services like Apple Siri, Google Now and Microsoft voice recognition. How to handle voice input and preserve its privacy is very challenging and will be our future work.

Beyond Mobile IME Apps Note that the approach of I-BOX does not necessarily only apply to mobile platforms; Similar techniques can also be applied to desktops, which suffer from a similar dilemma between privacy and usability. We may provide a similar oblivious sandbox for each IME app, which should be straightforward as Android actually runs atop Linux. We leave this as our future work. Besides, other applications that requires a tradeoff between privacy and usability may use execution transaction like I-BOX.

7 RELATED WORK

Privacy leakage detection in mobile devices. Recently, there have been significant efforts on the detection of privacy leakage in mobile devices. Early attempts include TaintDroid [16, 17] and PiOS [15], and recent efforts include such as Woodpecker [22], AndroidLeaks [20], ContentScope [50], and Appprofiler [35]. In particular, TaintDroid [16] uses dynamic taint analysis to track whether sensitive information (e.g., address book) can be leaked through the network. PiOS [15] uses static analysis and focuses on the privacy leakage in iOS apps. Woodpecker [22] leverages an inter-procedural data-flow analysis to inspect whether an untrusted app can obtain unauthorized access to sensitive data. ContentScope [50] detects passive content leak vulnerabilities, by which in-app sensitive data can be leaked.

AndroidLeaks [20] instead uses static analysis to detect data leakage in Android apps. Chan et al. [10] further leverages mobile forensics to correlate user actions with privacy leakages. Appprofiler [35] creates a mapping between high-level API calls and low-level privacy-related behavior, which is then used to provide a high-level profile of App's privacy behavior. Besides, there have also been interests in detecting privacy leakage due to mobile ads [38]. In contrast, I-BOX focuses on preventing leakage of sensitive keystrokes.

Privacy leakage prevention in mobile devices. Other than detecting privacy leakage, there are also a number of systems that prevent private data from being leaked. By extending TaintDroid [16], AppFence [24] prevents applications from accessing sensitive information using data shadowing, and it also blocks outgoing communications tainted by sensitive data. While I-BOX and AppFence both block network communications when sensitive data is to be leaked, there are substantial differences: AppFence uses shadowing to provide an illusion to the app such that it can continue performing its taint tracking, whereas I-BOX does not use any illusion nor any instruction-level taint tracking, due to the pervasive existence of native code. Meanwhile, AppFence does not encounter the challenges we faced such as consistent rollback, and it only simply blocks the network communication, whereas I-BOX still has to keep the connection and allow other data to be transferred.

TISSA [51] tames information stealing apps to stop possible privacy leakage. SpanDex [14] further uses symbolic execution to quantify and limit the implicit flows through a sandbox, to prevent an untrusted application from leaking passwords. Through automatic repackaging of Android apps, Aurasium [43] attaches sandboxing and policy enforcement atop existing apps, to stop malicious behaviors such as attempts to retrieve users' sensitive information. Unlike Aurasium that adds a sandbox to an app, πBox [30] shifts the sandboxing protection of private data from the app level to the system level, and offers a platform for privacy-preserving apps. However πBox trusts a few app vendors to protect users' privacy data, while I-BOX treats the vendor of IME apps as untrusted, due to their incentives to collect users' input. TinMan [42] instead completely offload passwords-like secret to a remote cloud, but only handles a class of special secrets that are not necessary to be displayed in mobile devices. ScreenPass [31] leverages a trusted software keyboard to input and tag passwords and uses taint tracking to ensure that a password is only used within a specific domain. In contrast, while I-BOX also uses a trusted software keyboard for password input, it focuses more on preventing a malicious IME from leaking sensitive data (not only passwords).

Checkpoint and restore. I-BOX employs a checkpoint and restore mechanism to prevent privacy leakage. Such a mechanism has been built for transactional memory [6], execution transactions [37], as well as whole-system transactions [33]. Retro [26] leverages selective re-execution for intrusion recovery. Storage Capsules [7] also use checkpoint and restore to wipe off residual data after an application has viewed data in a desktop. I-BOX is an instance of a system transaction but designed specially for untrusted IME apps.

Sandboxing. There have been a large number of efforts in building sandboxes to execute untrusted programs, web applications, and native code. These tools were built using a variety of approaches such as kernel-based systems [19], user-level approaches [27], system call interpositions [21], or binary code translation [18], and recompilation [45].

A sandbox that also contains transactions is the TxBox [25], a tool built atop TxOS [33] for speculative execution and automatic recovery. While I-BOX and TxBox share the similarity of using transactions to build a sandbox, there are still significant differences: the goal of TxBox is to confine the execution of native x86 programs atop Linux kernel, whereas I-BOX is to confine the IME apps atop Android OS. Consequently, I-BOX faces additional challenges including resolving IPC bindings. Further, using quiescent points in I-BOX significantly simplifies the design and implementation.

8 CONCLUSION

This paper made a first systematic study on the (in)security of third-party (trusted or untrusted) IME apps, and revealed that these apps tend to leak users' sensitive input (due to their incentives of improving user's experience). To enjoy the rich-experiences offered by such apps while mitigating information leakages, this paper described I-BOX as a first step towards this direction. In light of the opaque nature of an IME app, I-BOX leverages the idea of transactions to run an IME app to make it oblivious to users' sensitive input. Experiments showed that I-BOX is efficient, incurs little impact on users' experiences and successfully thwarted the leakage of sensitive user input.

ACKNOWLEDGMENTS

We thank our shepherd William Enck and the anonymous reviewers for their insightful comments, Xiaojuan Li and Yutao Liu for helping prepare the final version. This work is supported in part by the Program for New Century Excellent Talents in University, Ministry of Education of China (No. ZXZY037003), a foundation for the Author of National Excellent Doctoral Dissertation of PR China (No. TS0220103006), the Shanghai Science and Technology Development Fund for high-tech achievement translation (No. 14511100902), Zhangjiang Hi-Tech program (No. 201501-YP-B108-012), and the Singapore NRF (CREATE E2S2).

REFERENCES

[1] Free Chinese-made software poses security risk. http://www.japantimes.co.jp/news/2013/12/26/national/chinese-made-computer-input-system-banned-in-government-agencies/#.U21w5_aPUS0.

[2] smali-An assembler/disassembler for Android's dex format. https://code.google.com/p/smali/.

[3] Security enhancements in jelly bean. http://android-developers.blogspot.jp/2013/02/security-enhancements-in-jelly-bean.html, 2013.

[4] A. J. Aviv, B. Sapp, M. Blaze, and J. M. Smith. Practicality of accelerometer side channels on smartphones. In *ACSAC*, 2012.

[5] BBC News. Salford woman makes bid for fastest text title. http://news.bbc.co.uk/local/manchester/hi/people_and_places/newsid_8939000/8939790.stm, 2010.

[6] A. Birgisson, M. Dhawan, U. Erlingsson, V. Ganapathy, and L. Iftode. Enforcing authorization policies using transactional memory introspection. In *CCS*, pages 223–234, 2008.

[7] K. Borders, E. Vander Weele, B. Lau, and A. Prakash. Protecting confidential data on personal computers with storage capsules. In *Usenix Security*, 2009.

[8] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *NDSS*, 2012.

[9] L. Cai and H. Chen. Touchlogger: inferring keystrokes on touch screen from smartphone motion. In *HotSec*, 2011.

[10] J. J. K. Chan, K. W. Tan, L. Jiang, and R. K. Balan. The case for mobile forensics of private data leaks: Towards large-scale user-oriented privacy protection. In *APSYS*, 2013.

[11] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *Oakland*, pages 191–206, 2010.

[12] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *MobiSys*, pages 239–252. ACM, 2011.

[13] China IT Research Center. Third-part IMEs usage stats in China for 2014 Q1. http://www.cnit-research.com/content/201405/303.html, 2014.

[14] L. P. Cox, P. Gilbert, G. Lawler, V. Pistol, A. Razeen, B. Wu, and S. Cheemalapati. Spandex: Secure password tracking for android. In *USENIX Security*, 2014.

[15] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. Pios: Detecting privacy leaks in ios applications. In *NDSS*, 2011.

[16] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.

[17] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM TOCS*, 32(2):5, 2014.

[18] B. Ford and R. Cox. Vx32: Lightweight user-level sandboxing on the x86. In *USENIX ATC*, 2008.

[19] T. Fraser, L. Badger, and M. Feldman. Hardening cots software with generic software wrappers. In *Oakland*, pages 2–16, 1999.

[20] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale. In *Trust*, 2012.

[21] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications confining the wily hacker. In *USENIX Security*, 1996.

[22] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *NDSS*, 2012.

[23] N. Hardy. The confused deputy:(or why capabilities might have been invented). *SIGOPS Oper. Sys. Review*, 22(4):36–38, 1988.

[24] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *CCS*, 2011.

[25] S. Jana, D. E. Porter, and V. Shmatikov. Txbox: Building secure, efficient sandboxes with system transactions. In *Oakland*, 2011.

[26] T. Kim, X. Wang, N. Zeldovich, M. Kaashoek, et al. Intrusion recovery using selective re-execution. In *OSDI*, 2010.

[27] T. Kim and N. Zeldovich. Practical and effective sandboxing for non-root users. In *USENIX ATC*, pages 139–144, 2013.

[28] O. Laadan and J. Nieh. Transparent checkpoint-restart of multiple processes on commodity operating systems. In *USENIX ATC*, pages 323–336, 2007.

[29] W. S. Labs. Fake input method editor(ime) trojan. http://community.websense.com/blogs/securitylabs/archive/2010/07/05/trojan-using-input-method-inject-technology.aspx.

[30] S. Lee, E. L. Wong, D. Goel, M. Dahlin, and V. Shmatikov. πbox: a platform for privacy-preserving apps. In *NSDI*, 2013.

[31] D. Liu, E. Cuervo, V. Pistol, R. Scudellari, and L. P. Cox. Screenpass: Secure password entry on touchscreen devices. In *MobiSys*, pages 291–304, 2013.

[32] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *ASIACCS*, pages 328–332, 2010.

[33] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating system transactions. In *SOSP*, 2009.

[34] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: Automatic security analysis of smartphone applications. In *ACM conference on Data and application security and privacy*, 2013.

[35] S. Rosen, Z. Qian, and Z. M. Mao. Appprofiler: a flexible method of exposing privacy-related behavior in android applications to end users. In *ACM conference on Data and application security and privacy*, pages 221–232. ACM, 2013.

[36] M. A. Salehi, T. Caldwell, A. Fernandez, E. Mickiewicz, E. W. Rozier, S. Zonouz, and D. Redberg. Reseed: Regular expression search over encrypted data in the cloud. In *CCGrid*, 2014.

[37] S. Sidiroglou, O. Laadan, A. D. Keromytis, and J. Nieh. Using rescue points to navigate software recovery. In *Oakland*, 2007.

[38] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen. Investigating user privacy in android ad libraries. In *Workshop on Mobile Security Technologies (MoST)*, 2012.

[39] K. Subramanyam, C. E. Frank, and D. F. Galli. Keyloggers: The overlooked threat to computer security. http://www.keylogger.org/articles/kishore-subramanyam/keyloggers-the-overlooked-threat-to-computer-security-7.html.

[40] TechSpot News. Google fired employees for breaching user privacy. `http://www.techspot.com/news/40280-google-fired-employees-for-breaching-user-privacy.html`, 2010.

[41] Y. Xia, Y. Liu, and H. Chen. Architecture support for guest-transparent vm protection from untrusted hypervisor and physical attacks. In *HPCA*, 2013.

[42] Y. Xia, Y. Liu, C. Tan, M. Ma, H. Guan, B. Zang, and H. Chen. Tinman: eliminating confidential mobile data exposure with security oriented offloading. In *EuroSys*, 2015.

[43] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *USENIX Security*, 2012.

[44] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *ICSE*, 2015.

[45] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. *Commun. ACM*, 53(1):91–99, Jan. 2010.

[46] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *CCS*, 2007.

[47] F. Zhang, J. Chen, H. Chen, and B. Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *SOSP*, 2011.

[48] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *ACM conference on Data and Application Security and Privacy*, pages 317–326. ACM, 2012.

[49] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Oakland*, 2012.

[50] Y. Zhou and X. Jiang. Detecting passive content leaks and pollution in android applications. In *NDSS*, 2013.

[51] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming information-stealing smartphone applications (on android). In *Conference on Trust and Trustworthy Computing*, 2011.

Boxify: Full-fledged App Sandboxing for Stock Android

Michael Backes
CISPA, Saarland University & MPI-SWS
backes@cs.uni-saarland.de

Sven Bugiel
CISPA, Saarland University
bugiel@cs.uni-saarland.de

Christian Hammer
CISPA, Saarland University
hammer@cs.uni-saarland.de

Oliver Schranz
CISPA, Saarland University
schranz@cs.uni-saarland.de

Philipp von Styp-Rekowsky
CISPA, Saarland University
styp-rekowsky@cs.uni-saarland.de

Abstract

We present the first concept for full-fledged app sandboxing on stock Android. Our approach is based on application virtualization and process-based privilege separation to securely encapsulate untrusted apps in an isolated environment. In contrast to all related work on stock Android, we eliminate the necessity to modify the code of monitored apps, and thereby overcome existing legal concerns and deployment problems that rewriting-based approaches have been facing. We realize our concept as a regular Android app called Boxify that can be deployed without firmware modifications or root privileges. A systematic evaluation of Boxify demonstrates its capability to enforce established security policies without incurring a significant runtime performance overhead.

1 Introduction

Security research of the past five years has shown that the privacy of smartphone users—and in particular of Android OS users, due to Android's popularity and open-source mindset—is jeopardized by a number of different threats. Those include increasingly sophisticated malware and spyware [63, 39, 62], overly curious libraries [25, 32], but also developer negligence and absence of fail-safe defaults in the Android SDK [33, 29]. To remedy this situation, the development of new ways to protect the end-users' privacy has been an active topic of Android security research during the last years.

Status quo of deploying Android security extensions. From a deployment perspective, the proposed solutions followed two major directions: The majority of the solutions [26, 44, 45, 16, 21, 64, 52, 56] extended the UID-centered security architecture of Android. In contrast, a number of solutions [38, 59, 23, 49, 22, 15] promote inlined reference monitoring (IRM) [28] as an alternative approach that integrates security policy enforcement directly into Android's application layer, i.e., the apps' code.

However, this dichotomy is unsatisfactory for end-users: While OS security extensions provide stronger security guarantees and are preferable in the long run, they require extensive modifications to the operating system and Android application framework. Since the proposed solutions are rarely adopted [54, 53] by Google or the device vendors, users have to resort to customized aftermarket firmware [4, 6] if they wish to deploy new security extensions on their devices. However, installing a firmware forms a technological barrier for most users. In addition, fragmentation of the Android ecosystem [46] and vendor customizations impede the provisioning of custom-built ROMs for all possible device configurations in the wild.

In contrast, solutions that rely on inlined reference monitoring avoid this deployment problem by moving the reference monitor to the application layer and allowing users to install security extensions in the form of apps. However, the currently available solutions provide only insufficient app sandboxing functionality [36] as the reference monitor and the untrusted application share the same process space. Hence, they lack the strong isolation that would ensure tamper-protection and non-bypassability of the reference monitor. Moreover, inlining reference monitors requires modification and hence re-signing of applications, which violates Android's signature-based same-origin model and puts these solutions into a legal gray area.

The sweet spot. The envisioned app sandboxing solution provides immediate strong privacy protection against rogue applications. It would combine the security guarantees of OS security extensions with the deployability of IRM solutions, while simultaneously avoiding their respective drawbacks. Effectively, such a solution would provide an OS-isolated reference monitor that can be deployed entirely as an app on stock Android without modifications to the firmware or code of the monitored applications.

Our contributions. In this paper we present a novel concept for Android app sandboxing based on *app virtualization*, which provides tamper-protected reference monitoring without firmware alterations, root privileges or modifications of apps. The key idea of our approach is to encapsulate untrusted apps in a restricted execution environment within the context of another, trusted sandbox application. To establish a restricted execution environment, we leverage Android's *"isolated process"* feature, which allows apps to totally de-privilege selected components—a feature that has so far received little attention beyond the web browser. By loading untrusted apps into a de-privileged, isolated process, we shift the problem of sandboxing the untrusted apps from *revoking* their privileges to *granting* their I/O operations whenever the policy explicitly allows them. The I/O operations in question are syscalls (to access the file system, network sockets, bluetooth, and other low-level resources) and the Binder IPC kernel module (to access the application framework). We introduce a novel app virtualization environment that proxies all syscall and Binder channels of isolated apps. By intercepting any interaction between the app and the system (i.e., kernel and app framework), our solution is able to enforce established and new privacy-protecting policies. Additionally, it is carefully crafted to be transparent to the encapsulated app in order to keep the app agnostic about the sandbox and retain compatibility to the regular Android execution environment. By executing the untrusted code as a de-privileged process with a UID that differs from the sandbox app's UID, the kernel securely and automatically isolates at process-level the reference monitor implemented by the sandbox app from the untrusted processes. Technically, we build on techniques that were found successful in related work (e.g., libc hooking [59]) while introducing new techniques such as Binder IPC redirection through ServiceManager hooking. We realize our concept as a regular app called Boxify that can be deployed on stock Android. To the best of our knowledge, Boxify is the first solution to introduce *application virtualization* to stock Android.
In summary, we make the following contributions:

1. We present a novel concept for application virtualization on Android that leverages the security provided by *isolated processes* to securely encapsulate untrusted apps in a completely de-privileged execution environment within the context of a regular Android app. To retain compatibility of isolated apps with the standard Android app runtime, we solved the key technical challenge of designing and implementing an efficient app virtualization layer.

2. We realize our concept as an app called Boxify, which is the first solution that ports app virtualization to the Android OS. Boxify is deployable as a regular app on stock Android (no firmware modification and no root privileges required) and avoids the need to modify sandboxed apps.

3. We systematically evaluate the efficacy and efficiency of Boxify from different angles including its security guarantees, different use-cases, performance penalty, and Android API version dependence across multiple Android OS versions.

The remainder of this paper is structured as follows. In §2, we provide necessary technical background information on Android. We define our objectives and discuss related work in §3. In §4, we present our Boxify design and implementation, which we evaluate in §5. We conclude the paper in §6.

2 Background on Android OS

Android OS is an open-source software stack (see Figure 1) for mobile devices consisting of a Linux kernel, the Android application framework, and system apps. The application framework together with the pre-installed system apps implement the Android application API. The software stack can be extended with third-party apps, e.g., from Google Play.

Android Security Model. On Android, each application runs in a separate, simple sandboxed environment that isolates data and code execution from other apps. In contrast to traditional desktop operating systems where applications run with the privileges of the invoking user, Android assigns a unique Linux user ID (UID) to every application at installation time. Based on this UID, the components of the Android software stack enforce access control rules that govern the app sandboxing. To understand the placement of the enforcement points, one has to consider how an app can interact with other apps (and processes) in the system:

Like any other Linux process, an app process uses syscalls to the Linux kernel to access low-level resources, such as files. The kernel enforces discretionary access control (DAC) on such syscalls based on the UID of the application process. For instance, each application has a private directory that is not accessible by other applications and DAC ensures

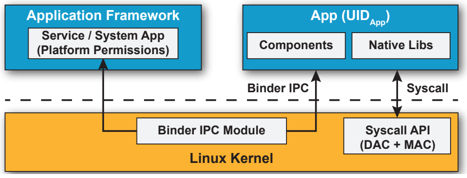


Figure 1: High-level view of interaction between apps, application framework, and Linux kernel on Android.

that applications cannot access other apps' private directories. Since Android version 4.3 this discretionary access control is complemented with SELinux mandatory access control (MAC) to harden the system against low-level privilege escalation attacks and to reinforce this UID-based compartmentalization.

The primary channel for inter-application communication is Binder *Inter-Process Communication* (IPC). It is the fundamental building block for a number of more abstract inter-app communication protocols, most importantly *Inter-Component Communication* (ICC) [27] among apps and the application framework. For sandboxing applications at the ICC level, each application UID is associated with a set of platform permissions, which are checked at runtime by reference monitors in the system services and system apps that constitute the app framework (e.g. `LocationService`). These reference monitors rely on the Binder kernel module to provide the UID of IPC senders to the IPC receivers.

In general, both enforcement points are implemented *callee-sided* in the framework and kernel, and hence agnostic to the exact call-site within the app process. This means that enforcement applies equally to all code executing in a process under the app's UID, i.e., to both Java and native code.

Additionally, Android verifies the integrity of application packages during installation based on their developer signature. The corresponding developer certificate is afterwards used to enforce a same-origin policy for application updates, i.e., newer app versions must be signed with the same signing key as the already installed application.

Isolated Process. The *Isolated Process*, introduced in Android version 4.1, is a security feature that has received little attention so far. It allows an app developer to request that certain service components within her app should run in a special process that is isolated from the rest of the system and has no permissions of its own [2]. The isolated process mechanism follows the concept of *privilege separation* [48], which allows parts of an application to run at different levels of privilege. It is intended to provide an additional layer of protection around code that processes content from untrusted sources and is likely to have security holes. Currently, this feature is primarily geared towards web browsers [35] and is most prominently used in the Chrome browser to contain the impact of bugs in the complex rendering code.

An isolated process has far fewer privileges than a regular app process. An isolated process runs under a separate Linux user ID that is randomly assigned on process startup and differs from any existing UID. Consequently, the isolated process has no access to the private app directory of the application. More precisely, the process' filesystem interaction is limited to reading/writing world readable/writable files. Moreover, the isolated process' access to the Android middleware is severely restricted. The isolated process runs with no permissions, regardless of the permissions declared in the manifest of the application. More importantly, the isolated process is forbidden to perform any of the core Android IPC functions: Sending Intents, starting Activities, binding to Services or accessing Content Providers. Only the core middleware services that are essential to running the service component are accessible to the isolated process. This effectively bars the process from any communication with other apps. The only way to interact with the isolated process from other application components is through the Service API (binding and starting). Further, the transient UID of an isolated process does not belong to any privileged system groups and the kernel prevents the process from using low-level device features such as network communication, bluetooth or external storage. As of Android v4.3, SELinux reinforces this isolation through a dedicated process type. With all these restrictions in place, code running in an isolated process has only minimal access to the system, making it the most restrictive runtime environment Android has to offer.

3 Requirements Analysis and Existing Solutions

We first briefly formulate our objectives (see §3.1) and afterwards discuss corresponding related work (see §3.2 and Table 1).

3.1 Objectives and Threat Model

In this paper, we aim to combine the security benefits of OS extensions with the deployability benefits of application layer solutions. We identify the following objectives:

- **O1 No firmware modification:** The solution does not rely on or require customized Android firmware, such as extensions to Android's middleware, kernel or the default configuration files (e.g., policy files), and is able to run on stock Android versions. This also excludes availability of root privileges, since root can only be acquired through a firmware modification on newer Android versions due to increasingly stringent SELinux policies.

- **O2 No app modification:** The solution does not rely on or require any modifications of monitored apps' code, such as rewriting existing code.

- **O3 Robust reference monitor:** The solution provides a robust reference monitor. This encompasses: 1) the presence of a strong security boundary, such as a process boundary, between the reference monitor and untrusted code; and 2) the monitor cannot be bypassed, e.g., using a code representation that is not monitored, such as native code.

- **O4 Secure isolation of untrusted code:** This objective encompasses fail-safe defaults and complete mediation by the reference monitors. The solution provides a reference monitor that mediates all interaction between the untrusted code and the Android system, or, in case no complete mediation can be established, enforces fail-safe defaults that isolate the app on non-mediated channels in order to prevent untrusted code from escalating its privileges.

Threat model. We assume that the Android OS is trusted, including the Linux kernel and the Android application framework. This includes the assumption that an application cannot compromise the integrity of the kernel or application framework at runtime. If the kernel or application framework were compromised, no security guarantees could be upheld. Protecting the kernel and framework integrity is an orthogonal research direction for which different approaches already exist, such as trusted computing, code hardening, or control flow integrity.

Furthermore, we assume that untrusted third-party applications have full control over their process and the associated memory address space. Hence the attacker can modify its app's code at runtime, e.g., using native code or Java's reflection interface.

3.2 Existing Solutions

We systematically analyze prior solutions on app sandboxing.

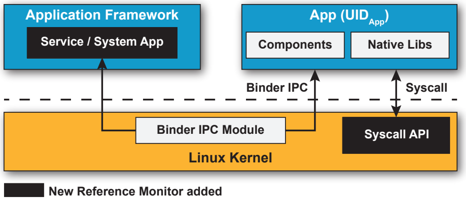


Figure 2: Instrumentation points for operating system security extensions.

Objectives	**OS ext.**	**IRM**	**Sep. app**	**Boxify**
O1: No system modification	✘	✔	✔	✔
O2: No application modification	✔	✘	✘	✔
O3: Robust reference monitor	✔	✘	✔	✔
O4: Secure isolation of untrusted code	✔	✘	✘	✔

✔= applies; ✘= does not apply.

Table 1: Comparison of deployment options for Android security extensions based on desired objectives.

3.2.1 Android Security Extensions

Many improvements to Android's security model have been proposed in the literature, addressing a variety of shortcomings in protecting the end-user's privacy. In terms of deployment options, we can distinguish between solutions that extend the Android OS and solutions that operate at the application layer only.

Operating system extensions. The vast majority of proposals from the literature (e.g. [26, 44, 45, 16, 21, 58]) statically enhance Android's application framework and Linux kernel with additional reference monitors and policy decision points (see Figure 2). The proposed security models include, for instance, context-aware policies [21], app developer policies [45], or Chinese wall policies [16]. More recent approaches [52, 43, 56] avoid static changes to the OS by dynamically instrumenting core system services (like Binder and Zygote) or the Android bootup scripts in order to interpose [47] untrusted apps' syscalls and IPC. Since in all approaches the reference monitors are part of the application framework and kernel, there inherently exists a strong security boundary between the reference monitor and untrusted code (O3: ✔). Moreover, this entails that these reference monitors are by design part of the callee-side of all interaction of the untrusted app's process with the system and cannot be by-

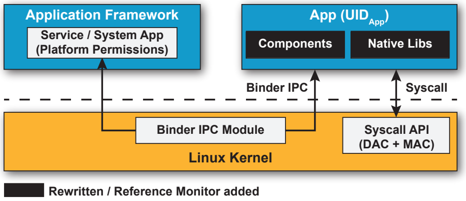


Figure 3: Instrumentation points for application code rewriting and inlining reference monitors.

passed (O4: ✔). On the downside, these solutions require modification of the Android OS (image) or root privileges to be deployed (O1: ✘; O2: ✔).

Additionally, a number of solutions exist that particularly target higher-security deployments [17, 51, 40, 13], such as government and enterprise. Commercial products exist that implement these solutions in the form of tailored mobile platforms (e.g., Blackphone[1], GreenHills[2], or Cryptophone[3]). These products target specialized user groups with high security requirements—not the average consumer—and are thus deployed on a rather small scale.

Application layer solutions. At the application layer, the situation for third-party security extensions is bleak. Android's UID-based sandboxing mechanism strictly isolates different apps installed on the same device. Android applications run with normal user privileges and cannot elevate to root in order to observe the behavior of other apps, e.g., like classical trace or anti-virus programs on desktop operating systems [31]. Also, Android does not offer any APIs that would allow one app to monitor or restrict the actions of another app at runtime. Only static information about other apps on the device is available via the Android API, i.e., application metadata, such as the package name or signing certificate, and the compiled application code and resources.Consequently, most commercially available security solutions are limited to *detecting* potentially malicious apps, e.g. by comparing metadata with predefined blacklists or by checking the application code for known malware signatures, but they lack the ability to observe or influence the runtime behavior of other applications. As a result, their effectiveness is, at best, debatable [50, 62].

Few proposals in the academic literature [38, 59, 23, 49, 15] focus on application layer only solutions (see Figure 3). Existing systems mostly focus on access control by interposing security-sensitive APIs to redirect the control flow to an additionally inlined reference monitor within the app (e.g., Aurasium [59], I-ARM-Droid [23], RetroSkeleton [22], AppGuard [15]). DroidForce [49] additionally preprocesses target apps with static data flow analysis to identify strategic policy enforcement points and to redirect policy decision making to a separate app.

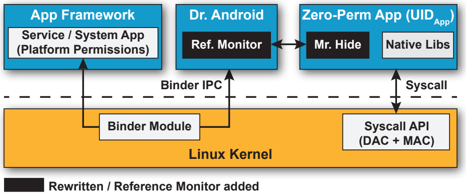


Figure 4: *Dr. Android and Mr. Hide* approach [38].

All these systems are based on rewriting the application code to inline reference monitors or redirect control flows, which works without modifications to the firmware and is thus suitable for large-scale deployment (O1: ✔; O2: ✘). However, app rewriting causes security problems and also a couple of practical deployment problems. First, inlining the reference monitor within the process of the untrusted app itself might be suitable for "benign-but-buggy" apps; however, apps that actively try to circumvent the monitor will succeed as there exists no strong security boundary between the app and the monitor. In essence, this boils down to an arms race between hooking security critical functions and finding new ways to compromise or bypass the monitor [36], where currently native code gives the attacker the advantage (O3: ✘; O4: ✘). Moreover, re-writing application code requires re-signing of the app, which breaks Android's signature-based same origin policy and additionally raises legal concerns about illicit tampering with foreign code. Lastly, re-written apps have to be reinstalled. This is not technically possible for pre-installed system apps; other apps have to be uninstalled in order to install a fresh, rewritten version, thereby incurring data loss.

Separate app. Dr. Android and Mr. Hide [38] (see Figure 4) is a variant of inlined reference monitoring (O1: ✔; O2: ✘) that improves upon the security of the reference monitor by moving it out of the untrusted app and into a separate app. This establishes a strong security boundary between the untrusted app and the reference monitor as they run in separate processes with different UIDs (O3: ✔). Additionally, it revokes all Android platform permissions from the untrusted app and applies code rewriting techniques to replace well-known security-sensitive Android API calls in the monitored app with calls to the separate

[1] https://blackphone.ch
[2] http://www.ghs.com/mobile/
[3] http://esdcryptophone.com

reference monitor app that acts as a proxy to the application framework. The benefit of this design is that in contrast to inlined monitoring, the untrusted, zero-permission app cannot gain additional permissions by tampering with the inlined/rewritten code. However, this enforcement only addresses the platform permissions. The untrusted app process still has a number of Linux privileges (such as access to the Binder interface or file system), and it has been shown that even a zero-permission app is still capable of escalating its privileges and violate the user's privacy [30, 33, 19, 18, 60, 42, 11, 12] (O4: ✗).

3.2.2 Sandboxing on traditional OSes

Restricting the access rights of untrusted applications has a longstanding tradition in desktop and server operating systems. Few solutions set up user-mode only sandboxes without relying on operating system functionality by making strong assumptions about the interface between the target code and the system (e.g., absence of programming language facilities to make syscalls or direct memory manipulation). Among the most notable user-space solutions are *native client* [61] to sandbox native code within browser extensions and the *Java virtual machine* [5] to sandbox untrusted Java applications.

Other solutions, which loosen the assumptions about the target interface to the system rely on operating system security features to establish process sandboxes. For instance, *Janus* [31], one of the earlier approaches, introduced an OS-supported sandbox for untrusted applications on Solaris 2.4, which was based on syscall monitoring and interception to restrict the untrusted process' access to the underlying operating system. The monitor was implemented as a separate process with necessary privileges to monitor and restrict other processes via the `/proc` kernel interface. Modern browsers like *Chromium* [9, 3, 8] employ different sandboxing OS facilities (e.g, seccomp mode) to mitigate the threat of web-based attacks against clients by restricting the access of untrusted code.

App virtualization. Sandboxing also plays a role in more recent *application virtualization* solutions [34, 10, 20, 41], where applications are transparently encapsulated into execution environments that replace (parts of) the environment with emulation layers that abstract the underlying OS and interpose all interaction between the app and the OS. App virtualization is currently primarily used to enable self-contained, OS-agnostic software, but also provides security benefits by restricting the interface and view the encapsulated app has of the system.

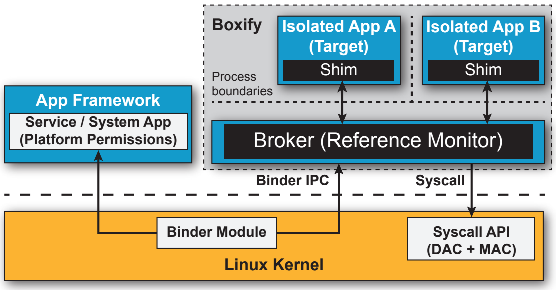


Figure 5: Architecture overview of Boxify.

Similarly to these traditional sandboxes and in particular to app virtualization, Boxify forms a user-mode sandbox that builds on top of existing operating system facilities of Android. Thereby, it establishes app sandboxes that encapsulate Android apps without the need to modify the OS and without the need to make any assumptions about the apps' code.

4 Boxify Architecture

We present the Boxify design and implementation.

4.1 Design Overview

The key idea of Boxify is to securely sandbox Android apps, while avoiding any modification of the OS and untrusted apps. Boxify accomplishes this by dynamically loading and executing the untrusted app in one of its own processes. The untrusted application is not executed by the Android system itself, but runs completely encapsulated within the runtime environment that Boxify provides and that can be installed as a regular app on stock Android (see Figure 5). This approach eliminates the need to modify the code of the untrusted application and works without altering the underlying OS (O1: ✔; O2: ✔). It thus constitutes the first solution that ports the concept of app virtualization to the stock Android OS.

The primary challenge for traditional application sandboxing solutions is to completely mediate and monitor all I/O between the sandboxed app and the system in order to restrict the untrusted code's privileges. The key insight for our Boxify approach is to leverage the security provided by *isolated processes* in order to isolate the untrusted code running within the context of Boxify by executing it in a completely de-privileged process that has no platform permissions, no access to the Android middleware, nor the ability to make persistent changes to the file system.

However, Android apps are tightly integrated within the application framework, e.g., for lifecycle management and inter-component communication. With the restrictions of an isolated process in place, encapsulated apps are rendered dysfunctional. Thus, the key challenge for Boxify essentially shifts from constraining the capabilities of the untrusted app to now gradually permitting I/O operations in a controlled manner in order to securely re-integrate the isolated app into the software stack. To this end, Boxify creates two primary entities that run at different levels of privilege: A privileged controller process known as the Broker and one or more isolated processes called the Target (see Figure 5).

The Broker is the main Boxify application process and acts as a mandatory proxy for all I/O operations of the Target that require privileges beyond the ones of the isolated process. Thus, if the encapsulated app bypasses the Broker, it is limited to the extremely confined privilege set of its isolated process environment (*fail-safe defaults*; O4: ✔). As a consequence, the Broker is an ideal control-flow location in our Boxify design to implement a reference monitor for any privileged interaction between a Target and the system. Any syscalls and Android API calls from the Target that are forwarded to the Broker are evaluated against a security policy. Only policy-enabled calls are then executed by the Broker and their results returned to the Target process. To protect the Broker (and hence reference monitor) from malicious app code, it runs in a separate process under a different UID than the isolated processes. This establishes a strong security boundary between the reference monitor and the untrusted code (O3: ✔). To transparently forward the syscalls and Android API calls from the Target across the process boundary to the Broker, Boxify uses Android's Binder IPC mechanism. Finally, the Broker's responsibilities also include managing the application lifecycle of the Target and relaying ICC between a Target and other (Target) components.

The Target hosts all untrusted code that will run inside the sandbox. It consists of a shim that is able to dynamically load other Android applications and execute them. For the encapsulated app to interact with the system, it sets up interceptors that interpose system and middlware API calls. The interceptors do *not* form a security boundary but establish a compatibility layer when the code inside the sandbox needs to perform otherwise restricted I/O by forwarding the calls to the Broker. All resources that the Target process uses have to be acquired by the Broker and their handles duplicated into the Target process.

By encapsulating untrusted apps and interposing all their (privileged) I/O operations, Boxify is able to effectively enforce security- and privacy-protecting policies. Based on syscall interposition, Boxify has fine-grained control over network and filesystem operations. Intercepting Binder IPC enables the enforcement of security policies that were so far only achievable for OS extensions, but at application layer only.

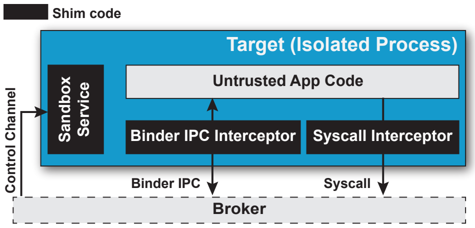


Figure 6: Components of a Target process.

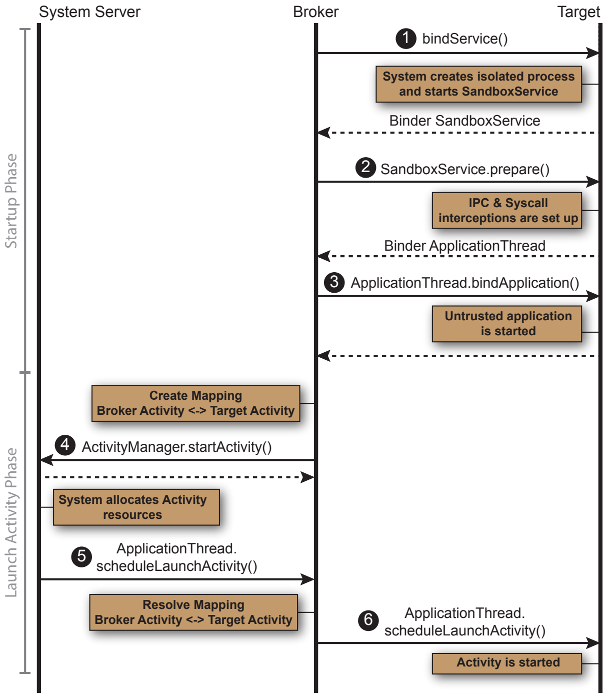
Moreover, with this architecture, Boxify can provide a number of interesting novel features. Boxify is capable of monitoring multiple (untrusted) apps at the same time. By creating a number of Target processes, multiple apps can run in parallel yet securely isolated in a single instance of Boxify. Since the Broker fully controls all inter-component communication between the sandboxed apps, it is able to not only separate different apps from one another but also to allow controlled collaboration between them. Further, Boxify has the ability to execute apps that are not regularly installed on the phone: Since Boxify executes other apps by dynamically loading their code into one of its own processes and handles all the interaction between the sandboxed application and the OS, there is no need to register the untrusted app with the Android system. Hence, applications can be installed into, updated, or removed from Boxify without involving the `PackageInstaller` or having system privileges. A potential application of these features are application containers (e.g., enterprise app domain, see §5.4).

4.2 Target

The Target process contains four main entities (see Figure 6): The SandboxService (1) provides the Broker with a basic interface for starting and terminating apps in the sandbox. It is also responsible for setting up the interceptors for Binder IPC (2) and syscalls (3), which transparently forward calls issued by the untrusted application to the Broker.

1) SandboxService. Isolated processes on Android are realized as specifically tagged Service components (see §2). In Boxify each Target is implemented as such a tagged SandboxService component of the Boxify app. When a new Target should be spawned, a new, dedicated SandboxService is spawned. The SandboxService provides an IPC interface that enables the Broker to communicate with the isolated process and to call two basic lifecycle operations for the Target: `prepare` and `terminate`. The Broker invokes the `prepare` function to initialize the sandbox environment for the execution of a hosted app. As part of this preparation, the Broker and Target exchange important configuration information for correct operation of the Target, such as app meta-information and Binder IPC handles that allow bi-directional IPC between Broker and Target. The `terminate` function shuts down the application running in the sandbox and terminates the Target process.

The biggest technical challenge at this point was *"How to execute another third-party application within the running isolated service process?"* Naïvely, one could consider, for instance, a warm-restart of the app process with the new application code using the `exec` syscall. However, we discovered that the most elegant and reliable solution is to have the Broker initially imitate the `ActivityManager` by instructing the Target process to load (i.e., *bind*) another application to its process and afterwards to relay any lifecycle events between the actual application framework and the newly loaded application in the Target process. The *bind* operation is supported by the standard Android application framework and used during normal app startup. The exact procedure is illustrated in Figure 7. The Broker first creates a new SandboxService process (❶), which executes with the privileges of an isolated process. This step actually involves multiple messages between the Broker process, the Target process and the system server, which we omitted here for the sake of readability. As a result, the Broker process receives a Binder handle to communicate with the newly spawned SandboxService. Next, the Broker uses this handle to instruct the SandboxService to prepare the loading of a sandboxed app (❷) by setting up the Binder IPC interceptor and syscall interceptor (using the meta-information given as parameters of the `prepare` call). The SandboxService returns the Binder handle to its `ApplicationThread` to the Broker. The application thread is the main thread of a process containing an Android runtime and is used by the `ActivityManager` to issue commands to Android application processes. At this point, the Broker emulates the behavior of the `ActivityManager` (❸) by instructing the `ApplicationThread` of the Target with the `bindApplication` call to load the target app into its Android runtime and start its execution. By default, it would be the `ActivityManagerService` as part of the application framework that uses this call to instruct newly forked and specialized Zygote processes to load and execute an application that should be started. After this step, the sandboxed app is executing.



System Server
Broker
Target
Startup Phase
Launch Activity Phase
1 bindService()
System creates isolated process and starts SandboxService
Binder SandboxService
2 SandboxService.prepare()
IPC & Syscall interceptions are set up
Binder ApplicationThread
3 ApplicationThread.bindApplication()
Untrusted application is started
Create Mapping Broker Activity <-> Target Activity
4 ActivityManager.startActivity()
System allocates Activity resources
5 ApplicationThread. scheduleLaunchActivity()
Resolve Mapping Broker Activity <-> Target Activity
6 ApplicationThread. scheduleLaunchActivity()
Activity is started

Figure 7: Process to load an app into a Target process and to launch one its Activities.

As an example how a sandboxed app can be used, we briefly explain how an Activity component of the sandboxed app can be launched, e.g., as result of clicking its entry in a launcher. As explained in §4.3, the Virtualization Layer creates a mapping from generic Boxify components to Target components. In this case, it maps the Activity component of Target to an Activity component of Boxify. The Broker requests the Activity launch from the `ActivityManager` in the `SystemServer` (❹), which allocates the required resources. After allocation, it schedules the launch of the Activity component by signaling the `ApplicationThread` of the targeted app (❺), which in this case is the Boxify app. Thus, the Virtualization Layer resolves the targeted Activity component and relays the signal to the corresponding Target process (❻).

2) Binder IPC Interceptor. Android applications use the Binder IPC mechanism to communicate with the (remote) components of other applications, including the application framework services and apps. In order to interact via Binder IPC with a remote component, apps must first acquire a Binder handle that connects them to the desired component.

To retrieve a Binder handle, applications query the ServiceManager, a central service registry, that allows clients to lookup system services by their common names. The ServiceManager is the core mechanism to bootstrap the communication of an application with the Android application framework. Binder handles to non-system services, such as services provided by other apps, can be acquired from the core framework services, most prominently the ActivityManager.

Boxify leverages this choke point in the Binder IPC interaction to efficiently intercept calls to the framework in order to redirect them to the Broker. To this end, Boxify replaces references to the ServiceManager handle in the memory of the Target process with references to the Binder handle of the Broker (as provided in the prepare function). These references are constrained to a few places and can be reliably modified using the Java Reflection API and native code. Consequently, all calls directed to the ServiceManager are redirected to the Broker process instead, which can then manipulate the returned Binder objects in such a way that any subsequent interactions with requested services are also redirected to the Broker. Furthermore, references to a few core system services, such as the ActivityManager and PackageManager, that are passed by default to new Android app runtimes, need to be replaced as well. By modifying only a small number of Binder handles, Boxify intercepts all Binder IPC communication. The technique is completely agnostic of the concrete interface of the redirected service and can be universally applied to all Binder interactions.

3) Syscall Interceptor. For system call interception, we rely on a technique called libc hooking (used, for instance, also in [59]). Applications use Android's implementation of the Standard C library *Bionic libc* to initiate system calls. With libc hooking, we efficently intercept calls to libc functions and redirect these calls to a service client running in the Target process. This client forwards the function calls via IPC to a custom service component running in the Broker. Due to space constraints, we refer to [7] for a detailed technical explanation of libc hooking.

In contrast to the IPC interception, which redirects all IPC communication to the Broker, the syscall interception is much more selective about which calls are forwarded: We do not redirect syscalls that would be anyway granted to an isolated process, because there is no security benefit from hooking these functions: a malicious app could simply remove the hook and would still succeed with the call. This exception applies to calls to read world-readable files and to most system calls that operate purely on file descriptors (e.g. read, write). Naturally, by omitting the indirection via our Broker, these exempted calls perform with native performance. However, Boxify still hooks calls that are security-critical and that are not permitted for isolated processes, such as system calls to perform file system operations (e.g. open, mkdir, unlink) and network I/O (socket, getaddrinfo). For a few calls, such as file operations, whose success depends on the given parameter, the syscall interception is parameter-sensitive in its decision whether or not to forward this operation to the Broker.

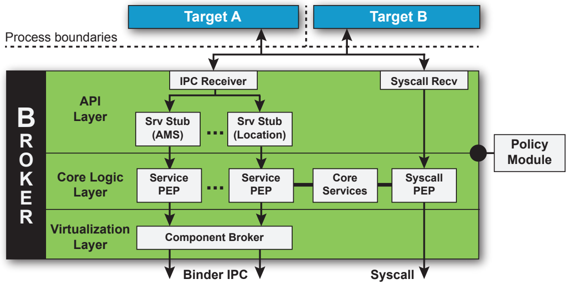


Figure 8: Architecture of the Broker.

4.3 Broker

The Broker is the main application process of Boxify and is thus not subject to the restrictions imposed by the isolated process. It holds all platform permissions assigned to the Boxify app and can normally interact with the Android middleware. The Broker acts as a mandatory proxy for all interactions between the Target processes and the Android system and thus embodies the reference monitor of Boxify. These interactions are bi-directional: On the one hand, the untrusted app running in the Target process issues IPC and syscalls to the system; on the other hand, the Android middleware initiates IPC calls to Target (e.g., basic lifecycle operations) and the Broker has to dispatch these events to the correct Target.

The Broker is organized into three main layers (see Figure 8): The API Layer (4) abstracts from the concrete characteristics of the Android-internal IPC interfaces to provide compatibility across different Android versions. It bridges the semantic gap between the raw IPC transactions forwarded by the Target and the application framework semantics of the Core Logic Layer (5), which implements the fundamental mechanics of the virtual runtime environment that Boxify provides. All interaction with the system happens through the Virtualization Layer (6), which translates between the virtual environment inside of

Boxify and the Android system on the outside. In the following, we will look at every layer in more detail.

4) API Layer. The API Layer is responsible for receiving and unwrapping the redirected syscall parameters from the Syscall Interceptor in the Target and relaying them to the Core Logic Layer for monitoring and execution. More importantly, it transforms the raw Binder IPC parcels received from the IPC Interceptor into a representation agnostic of the Android version.

In order to (efficiently) sandbox applications at the Binder IPC boundary, Boxify must semantically interpret the intercepted Binder parcels. However, intercepted parcels are in a raw representation that consists only of native types that the kernel module supports and the sender marshalled all higher-level objects (e.g., Java classes) to this representation. This impedes an efficient sandboxing. To solve this problem, Boxify leverages the default Android toolchain for implementing Binder-based RPC protocols: To ensure that sender and receiver can actually communicate with each other, the receiver must know how to unmarshal the raw parcel data (exactly like Boxify). Android supports the developers in this process through the *Android Interface Definition Language* (AIDL), which allows definitions of Binder interfaces very similar to Java interfaces, including the names and signatures of remotely callable functions. The Android SDK toolchain generates the required boilerplate marshalling code from AIDL definitions both for the receiver (*Stub*) and the sender (*Proxy*). For system services, these Stubs are automatically generated during system build and Boxify uses the generated Stubs (which ship with Android OS and are conveniently accessible to third-party application) to unmarshal the raw Binder IPC parcel back to their application framework semantic (i.e., Java objects, etc). In essence, this allows us to generate the API layer of the Broker in an almost fully-automatic way for each Android version on which Boxify is deployed. Since Boxify is in full control of the Binder handles of the encapsulated app (i.e., calls to the `ServiceManager`, `ActivityManager`, etc.), it can efficiently determine which Binder handle of the app addresses which system service and hence which Stub must be used to correctly unmarshal the raw Binder parcel intercepted from each handle.

However, the exact structure of the unmarshalled data and the functions (name and signature) depend entirely on the AIDL file. Since the system service interfaces describe the internal Android API, these interfaces change frequently between Android versions. Hence Boxify would have to implement each possible version of a Stub for every available Android version. Since this Stub implementation, in contrast to the marshalling logic, can not be automated, this complicates efficient sandboxing of apps across multiple Android versions. Consequently, it is desirable to transform the unmarshalled IPC data into a version-agnostic representation and then implement each Stub once and for all for this version. To accomplish this in Boxify, we borrow ideas from Google's proprietary *SafeParcel* class: In contrast to the regular Binder parcel, the SafeParcel carries structural information about the data stored in it, which allows the receiver of an IPC request to selectively read parts of the payload without knowing its exact structure. We achieve the same result by transforming the version-dependent parcel into a version-agnostic key-value store (where keys are the parameter names of methods declared in the interface definitions) and adapting the Core Logic Layer and Stub implementations to work with these version-agnostic data stores. Thus, while the API layer is version-dependent and automatically generated for each Android version, the remaining layers of Broker are version-agnostic and implemented only once.

5) Core Logic Layer. The Core Logic Layer provides essential functionality required to run apps on Android by replicating a small subset of the functionality that Android's core system services provide. Most prominently, this layer provides a minimal implementation of the `PackageManager`, which manages the packages installed into the Boxify environment. Every call to a system service that is not emulated by the Core Logic Layer is passed on to the Virtualization Layer and thus to the underlying Android system. Other system services, such as the `LocationManager`, which are not necessarily required, can be instantiated at this layer as well, in case encapsulated apps are supposed to use the local, Boxify service implementation instead of the pristine Android service (e.g., servicing sandboxed apps with fake location data [64]). Hence, this layer decides whether an Android API call is emulated using a replicated service or forwarded to the system (through the Virtualization Layer). This layer is therefore responsible for managing the IPC communication between different sandboxed apps (abstractly like an *"ICC switch"*).

Furthermore, the Core Logic Layer implements the policy enforcement points (PEP) for Binder IPC services and syscalls. Because the API Layer already bridges the semantic gap between kernel-level IPC and Android application framework semantics, this removes the burden for dealing with low-level semantics in the IPC PEPs. We emulate the integration of enforcement points into pristine Android services by integrating these points into our mandatory service proxies. This allows us to instantiate security models from the area of OS security extensions (see §3.2), but at the application layer. One default security model that Boxify provides is the permission enforcement and same origin model of Android. For instance, the replicated `ActivityManager` will enforce permissions on calls between components of two sandboxed apps. We present further security models from related work on OS security extensions that we integrated at this layer in §5.4 and for future work we consider a programmable interface for extending Core Logic Layer security in the spirit of ASM [37] and ASF [14]. For calls that are not protected by a permission, the Broker can also choose to enable direct communication between the target app and the requested Android system service. This can improve performance for non-critical services such as the `SurfaceFlinger` (for GUI updates) at the cost of losing the ability to mediate calls to these services.

The syscall PEP enforces system call policies in the spirit of [47] with respect to network and filesystem operations. Its responsibilities are twofold: First, it functions as a transparent compatibility layer by emulating the file-system structure of the Android data partition (e.g., `chroot` of sandboxed apps by emulating a home directory for each sandboxed app[4] within the home directory of the Boxify app). Second, it emulates the access control of the Linux kernel, i.e., compartmentalization of sandboxed apps by ensuring that they cannot access private files of other apps as well as enforcing permissions (e.g., preventing a sandboxed app without Internet permission from creating a network socket).

6) Virtualization Layer. The sandbox environment must support communication between sandboxed apps and the Android application framework, because certain system resources cannot be efficiently emulated (e.g., `SurfaceFlinger` for GUI) or not emulated at all (e.g., hardware resources like the camera). However, the sandbox must be transparent to the Target and all interaction with the application framework must appear as in a regular app. At the same time, the sandbox must be completely opaque to the application framework and sandboxed apps must be hidden from the framework; otherwise, this leads to inconsistencies that the framework considers as runtime (security) exceptions.

In Boxify, the Virtualization Layer is responsible for translating the bi-directional communication between the Android application framework and the Target. It achieves the required semi-transparent communication with a technique that can be abstractly described as *"ICC Network Address Translator"*: On outgoing calls from Target to framework, it ensures that all ICC appears as coming from the Boxify app instead of the sandboxed app. As described earlier, all Binder handles of a Target are substituted with handles of the Broker, which relays the calls to the system. During relay of calls, the Virtualization Layer manipulates the call arguments to hide components of sandboxed apps by substituting the component identifiers with identifiers of components of the Boxify app. On incoming calls from the framework, the Virtualization Layer substitutes the addressed Boxify component with the actually addressed component of the sandboxed app and dispatches the call. In order to correctly substitute addressed components, the Virtualization Layer maintains a mapping between Target and Boxify component names, or in case the Target component is not addressed by a name but a Binder handle that was given prior to the framework, the mapping is between the released Binder handle and its owning Target component.

A concrete example where this technique is applied is requesting the launch of a Target Activity component from the application framework (see Figure 7). The Virtualization Layer substitutes the Activity component with a generic Activity component of Boxify if a call to the `ActivityManager` occurs. When the service calls back for scheduling the Activity launch, the Virtualization Layer dispatches the scheduling call to the corresponding Target Activity component.

Lastly, we hook the application runtime of Boxify's Broker process (using a technique similar to [55]) in order to gain control over the processing of incoming Binder parcels. This enables the Broker to distinguish between parcels addressed to Boxify itself and those that need to be forwarded to the Target processes.

4.4 System Integration

Lastly, we discuss some aspects of integrating sandboxed apps into the default application framework.

Launcher. Since sandboxed apps have to be started through Boxify (and are not regularly installed on the system), they cannot be directly launched from the default launcher. A straightforward solution is to provide a custom launcher with Boxify in form of a dedicated Activity. Alternatively, Boxify could register as a launcher app and then run the default launcher (or any launcher app of the user's choice) in the sandbox, presenting the union of the regularly installed apps and apps installed in the sandbox environment; or Boxify launcher widgets could be placed

[4] Recall that sandboxed apps are not installed in the system but only in the Boxify environment, and hence do not have a native home directory.

Table 2: Microbenchmarks Middleware (200 runs)

API Call	Native	on Boxify	Overhead
Open Camera	103.24 ms	104.48 ms	1.24ms (1.2%)
Query Contacts	7.63 ms	8.55 ms	0.92 ms (12.0%)
Insert Contacts	66.49 ms	67.51 ms	1.02 ms (1.5%)
Delete Contacts	75.86 ms	76.81 ms	0.95 ms (0.9%)
Create Socket	120.83 ms	121.58 ms	0.75 ms (0.6%)

on the regular home screen to launch sandboxed apps from there.

App stores. Particularly smooth is the integration of Boxify with app store applications, such as the Google Play Store. Since no special permissions are required to install apps into the sandbox, we can simply run the store apps provided by Google, vendors, and third-parties in Boxify to install new apps there. For example, clicking install in the sandboxed Play Store App will directly install the new app into Boxify. Furthermore, Play Store (and vendor stores) even take care of automatically updating all apps installed in Boxify, a feature that IRM systems have to manually re-implement.

Statically registered resources. Some resources of apps are statically registered in the system during app installation. Since sandboxed apps are not regularly installed, the system is unaware of their resources. This concerns in particular Activity components that can receive Intents for, e.g., content sharing, or package resources like icons. However, some resources like Broadcast Receiver components can be dynamically registered at runtime and Boxify uses this as a workaround to dynamically register the Receivers declared statically in the Manifests of sandboxed apps.

5 Evaluation

We discuss the prototypical implementation of Boxify in terms of performance impact, security guarantees, and app robustness, and present concrete use-cases of Boxify. Our prototype comprises 11,901 lines of Java code, of which 4,242 LoC are automatically generated (API Layer), and 3,550 lines of additional C/C++ code. All tests described in the following were performed on an LG Nexus 5 running Android 4.4.4, which is currently the most widely used version in the Android ecosystem.

5.1 Performance Impact

To evaluate the performance impact of Boxify on monitored apps, we compare the results of common benchmark apps and of custom micro-benchmarks for encapsulated and native execution of apps.

Table 3: Microbenchmarks Syscalls (15k runs)

Libc Func.	Native	on Boxify	Overhead
`create`	47.2 μs	162.4 μs	115.2 μs
`open`	9.5 μs	122.7 μs	113.2 μs
`remove`	49.5 μs	159.6 μs	110.1 μs
`mkdir`	88.4 μs	199.4 μs	111.0 μs
`rmdir`	71.2 μs	180.7 μs	109.5 μs

Table 4: Benchmark Tools (10 runs)

Tool	Native	on Boxify	Loss
CF Bench v1.3	16082 Pts	15376 Pts	4.3%
Geekbench v3.3.1	1649 Pts	1621 Pts	1.6%
PassMark v1.0.4	3674 Pts	3497 Pts	4.8%
Quadrant v2.1.1	7820 Pts	7532 Pts	3.6%

Table 2 and Table 3 present the results of our microbenchmarks for common Android API calls and for syscall performance. Intercepting calls to the application framework imposes an overhead around 1%, with the exception of the very fast `Query Contacts` (12%). For syscalls, we measured the performance of calls that request file descriptors for file I/O in private app directories (or external storage) and that are proxied by the Broker. We observe a constant performance overhead of $\approx 100\mu s$, which corresponds to the required time of the additional IPC round trip for the communication with the Broker on our test platform. However, the syscall benchmarks depict a worst-case estimation: The overall performance impact on apps is much lower, since high-frequent follow up operations on acquired file descriptors (e.g., `read`/`write`) need not to be intercepted and therefore run with native speed. We measured the overall performance penalty by excecuting several benchmarking apps on top of Boxify, which show an acceptable performance degradation of 1.6%–4.8% (see Table 4).

5.2 Runtime Robustness

To assess the robustness of encapsulated apps, we executed 1079 of the most popular, free apps from Google Play (retrieved in August 2014) on top of Boxify. For each sandboxed app we used the *monkeyrunner* tool[5] to exercise the app's functionality by injecting 500 random UI events. From the 1079 apps, 93 (8.6%) experienced a crash during testing. Manual investigation of the dysfunctional apps revealed

[5] `http://developer.android.com/tools/help/monkeyrunner_concepts.html`

Table 5: Android versions supported by Boxify.

Version	< 4.1	4.1	4.2	4.3	4.4	5.0	5.1
Supported	✗[†]	✓	✓	✓	✓	✓	✓

✓: supported; ✗: not supported

[†]: no *isolated proccess*

that most errors were caused by apps executing exotic syscalls or rarely used Android APIs which are not covered by Boxify yet and thus fail due to the lack of privileges of the Target process (fail-safe defaults). This leads to a slightly lower robustness than reported for related work (e.g., [59, 15]) where bypassed hooks do not cause the untrusted app to crash but instead silently circumvent the reference monitor. The remaining issues were due to unusual application logic that relies on certain OS features (e.g., the process information pseudo-filesystem `proc`), which the current prototype of Boxify does not yet support. However, all of these are technical and not conceptual shortcomings of the current implementation of Boxify.

5.3 Portability

Table 5 summarizes the Android versions currently supported by our prototypical Boxify implementation. Our prototype supports all Android versions 4.1 through 5.1 and can be deployed on nine out of ten devices in the Android ecosystem [1]. Android versions prior to 4.1 are not supported due to the lack of the *isolated process* feature.

5.4 Use-cases

Boxify allows the instantiation of different security models from the literature on Android security extensions. In the following, we present two selected use-cases on fine-grained permission control and domain isolation that have received attention before in the security community.

Fine-Grained Permission Control. The TISSA [64] OS extension empowers users to flexibly control in a fine-grained manner which personal information will be accessible to applications. We reimplemented the TISSA functionality as an extension to the Core Logic Layer of the Boxify Broker. To this end, we instrumented the mandatory proxies for core system services (e.g. `LocationManager`, `TelephonyService`) so that they can return a filtered or mock data set based on the user's privacy settings. Users can dynamically adjust their privacy preferences through a management Activity added to Boxify. In total, the TISSA functionality required additional 351 lines of Java code to Core Logic Layer.

Domain Isolation. Particularly for enterprise deployments, container solutions have been brought forward to separate business apps from other (untrusted) apps [56, 17, 53].

We implemented a domain isolation solution based on Boxify by installing business apps into the sandbox environment. The Broker provides its own version of the `PackageManager` to directly deliver inter-component communication to sandboxed applications without involving the regular `PackageManager`, enabling controlled collaboration between enterprise apps while at the same time isolating and hiding them from non-enterprise apps and the OS.

To separate the enterprise data from the user's private data, we exploit that the Broker is able to run separate instances of system services (e.g., Contacts, Calendar) within the sandbox. Our custom `ActivityManager` proxy now selectively and transparently redirects `ContentProvider` accesses by enterprise apps to the sandboxed counterparts of those providers.

Alternatively, the above described domain isolation concept was used to implement a privacy mode for end users, where untrusted apps are installed into a Boxify environment with empty (or faked) system ContentProviders. Thus, users can test untrusted apps in a safe environment without risking harm to their mobile device or private data. The domain isolation extension required 986 additional lines of code in the Core Logic Layer of Boxify.

5.5 Security Discussion

Our solution builds on *isolated processes* as fundamental security primitive. An isolated process is the most restrictive execution environment that stock Android currently has to offer, and it provides Boxify with better security guarantees than closest related work [38]. In what follows, we identify different security shortcomings and discuss potential future security primitives of stock Android that would benefit Boxify and defensively programmed apps in general.

Privilege escalation. A malicious app could bypass the syscall and IPC interceptors, for instance, by statically linking `libc`. For IPC, this does not lead to a privilege escalation, since the application framework apps and services will refuse to cooperate with an isolated process. However, the kernel is unaware of the concept of an "isolated process" and will enforce access control on syscalls according to the process' UID. Although the transient UIDs of isolated processes are very restricted in their filesystem access (i.e., only world readable/writable files), a malicious process has the entire kernel API as an attack vector and might escalate its privileges through a root or kernel exploit. In this sense, Boxify is not more secure than existing approaches that rely on the assumption that the stock Android kernel is hardened against root and kernel exploits.

To remedy this situation, additional layers of security could be provided by the underlying kernel to further restrict untrusted processes. This is common practice on other operating systems, e.g., on modern Linux distributions, where Chromium—the primary user of isolated process on Android—uses the *seccomp-bpf* facility to selectively disable syscalls of renderer processes and we expect this facility to become available on future Android versions with newer kernels. Similarly, common program tracing facilities could be made available in order to interpose syscalls more securely and efficiently [31, 47, 52].

Violating Least-Privilege Principle. The Broker must hold the union set of all permissions required by the apps hosted by Boxify in order to successfully proxy calls to the Android API. Since it is hard to predict a reasonable set of permissions beforehand, this means that the Broker usually holds all available permissions. This contradicts the principle of least privilege and makes the Broker an attractive target for the encapsulated app to increase its permission set. A very elegant solution to this problem would be a Broker that drops all unnecessary permissions. This resembles the privilege separation pattern [48, 57] of established Linux services like ssh, which drop privileges of sub-processes based on setting their UIDs, capabilities, or transitioning them to seccomp mode. Unfortunately, Android does not (yet) provide a way to *selectively* drop permissions *at runtime*.

Red Pill. Even though Boxify is designed to be invisible to the sandboxed app, it cannot exclude that the untrusted app gathers information about its execution environment that allow the app to deduce that it is sandboxed (e.g., checking its runtime UID or permissions). A malicious app can use this knowledge to change its runtime behavior when being sandboxed and thus hide its true intentions or refuse to run in a sandboxed environment. Prevention of this information leak is an arms race that a resolute attacker will typically win. However, while this might lead to refused functionality, it cannot be used to escalate the app's privileges.

6 Conclusion

We presented the first application virtualization solution for the stock Android OS. By building on isolated processes to restrict privileges of untrusted apps and introducing a novel app virtualization environment, we combine the strong security guarantees of OS security extensions with the deployability of application layer solutions. We implemented our solution as a regular Android app called Boxify and demonstrated its capability to enforce established security policies without incurring significant runtime performance overhead.

Availability and Future Work. We will make the Boxify source code freely available. Beyond the immediate privacy benefits for the end-user presented in this paper (see §5.4), Boxify offers all the security advantages of traditional sandboxing techniques and is thus of independent interest for future Android security research. As future work, we are currently investigating different application domains of Boxify, such as application-layer only taint-tracking for sandboxed apps [24], programmable security APIs in the spirit of ASM [37]/ASF [14] to facilitate the extensibility of Boxify, as well as Boxify-based malware analysis tools.

References

[1] Android developer dashboard. `https://developer.android.com/about/dashboards/`. Last visited: 06/20/15.

[2] Android developer's guide. `http://developer.android.com/guide/index.html`. Last visited: 02/19/15.

[3] Chromium: Linux sandboxing. `https://code.google.com/p/chromium/wiki/LinuxSandboxing`. Last visited: 02/10/15.

[4] Cyanogenmod. `http://www.cyanogenmod.org`.

[5] Java SE Documentation: Security Specification. `http://docs.oracle.com/javase/7/docs/technotes/guides/security/spec/security-specTOC.fm.html`. Last visited: 02/10/15.

[6] OmniROM. `http://omnirom.org`. Last visited: 02/19/15.

[7] Redirecting functions in shared elf libraries. `http://www.codeproject.com/Articles/70302/Redirecting-functions-in-shared-ELF-libraries`.

[8] The Chromium Projects: OSX Sandboxing Design. `http://dev.chromium.org/developers/design-documents/sandbox/osx-sandboxing-design`. Last visited: 02/10/15.

[9] The Chromium Projects: Sandbox (Windows). `http://www.chromium.org/developers/design-documents/sandbox`. Last visited: 02/10/15.

[10] Wine: Run Windows applications on Linux, BSD, Solaris and Mac OS X. `https://www.winehq.org`. Last visited: 02/13/15.

[11] Zero-Permission Android Applications. https://www.leviathansecurity.com/blog/zero-permission-android-applications/. Last visited: 02/11/15.

[12] Zero-Permission Android Applications (Part 2). http://www.leviathansecurity.com/blog/zero-permission-android-applications-part-2/. Last visited: 02/11/15.

[13] Andrus, J., Dall, C., Hof, A. V., Laadan, O., and Nieh, J. Cells: A virtual mobile smartphone architecture. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP'11)* (2011), ACM.

[14] Backes, M., Bugiel, S., Gerling, S., and von Styp-Rekowsky, P. Android Security Framework: Extensible multi-layered access control on Android. In *Proc. 30th Annual Computer Security Applications Conference (ACSAC'14)* (2014), ACM.

[15] Backes, M., Gerling, S., Hammer, C., Maffei, M., and von Styp-Rekowsky, P. Appguard - enforcing user requirements on Android apps. In *Proc. 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13)* (2013), Springer.

[16] Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.-R., and Shastry, B. Towards Taming Privilege-Escalation Attacks on Android. In *Proc. 19th Annual Network and Distributed System Security Symposium (NDSS'12)* (2012), The Internet Society.

[17] Bugiel, S., Davi, L., Dmitrienko, A., Heuser, S., Sadeghi, A.-R., and Shastry, B. Practical and lightweight domain isolation on Android. In *Proc. 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'11)* (2011), ACM.

[18] Cai, L., and Chen, H. Touchlogger: inferring keystrokes on touch screen from smartphone motion. In *Proc. 6th USENIX conference on Hot topics in security (HotSec'11)* (2011), USENIX Association.

[19] Chen, Q. A., Qian, Z., and Mao, Z. M. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *Proc. 23rd USENIX Security Symposium (SEC'14)* (2014), USENIX Association.

[20] Citrix. Xenapp. http://www.citrix.com/products/xenapp/how-it-works/application-virtualization.html. Last visited: 02/13/15.

[21] Conti, M., Nguyen, V. T. N., and Crispo, B. CRePE: Context-Related Policy Enforcement for Android. In *Proc. 13th International Conference on Information Security (ISC'10)* (2010).

[22] Davis, B., and Chen, H. Retroskeleton: Retrofitting android apps. In *Proc. 11th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'13)* (2013), ACM.

[23] Davis, B., Sanders, B., Khodaverdian, A., and Chen, H. I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications. In *Proc. Mobile Security Technologies 2012 (MoST'12)* (2012), IEEE Computer Society.

[24] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2010)* (2010), pp. 393–407.

[25] Enck, W., Octeau, D., McDaniel, P., and Chaudhuri, S. A Study of Android Application Security. In *Proc. 20th USENIX Security Symposium (SEC'11)* (2011), USENIX Association.

[26] Enck, W., Ongtang, M., and McDaniel, P. On lightweight mobile phone application certification. In *Proc. 16th ACM Conference on Computer and Communication Security (CCS'09)* (2009), ACM.

[27] Enck, W., Ongtang, M., and McDaniel, P. Understanding android security. *IEEE Security and Privacy 7*, 1 (2009), 50–57.

[28] Erlingsson, Ú. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, January 2004.

[29] Fahl, S., Harbach, M., Muders, T., Smith, M., Baumgärtner, L., and Freisleben, B. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *Proc. 19th ACM Conference on Computer and Communication Security (CCS'12)* (2012), ACM.

[30] Felt, A. P., Wang, H. J., Moshchuk, A., Hanna, S., and Chin, E. Permission re-delegation: Attacks and defenses. In *Proc. 20th USENIX Security Symposium (SEC'11)* (2011), USENIX Association.

[31] Goldberg, I., Wagner, D., Thomas, R., and Brewer, E. A. A secure environment for untrusted helper applications confining the wily hacker. In *Proc. 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography (SSYM'96)* (1996), USENIX Association.

[32] Grace, M., Zhou, W., Jiang, X., and Sadeghi, A.-R. Unsafe exposure analysis of mobile in-app advertisements. In *Proc. 5th ACM conference on Security and Privacy in Wireless and Mobile Networks (WISEC'12)* (2012), ACM.

[33] Grace, M. C., Zhou, Y., Wang, Z., and Jiang, X. Systematic detection of capability leaks in stock android smartphones. In *Proc. 19th Annual Network and Distributed System Security Symposium (NDSS'12)* (2012), The Internet Society.

[34] Guo, P. J., and Engler, D. Cde: Using system call interposition to automatically create portable software packages. In *Proc. 2011 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'11)* (2011), USENIX Association.

[35] Hackborn, D. Android Developer Group: Advantage of introducing Isolatedprocess tag within Services in JellyBean. https://groups.google.com/forum/?fromgroups=#!topic/android-developers/pk45eUFmKcM, 2012. Last visited: 02/19/15.

[36] Hao, H., Singh, V., and Du, W. On the Effectiveness of API-level Access Control Using Bytecode Rewriting in Android. In *Proc. 8th ACM Symposium on Information, Computer and Communication Security (ASIACCS'13)* (2013), ACM.

[37] Heuser, S., Nadkarni, A., Enck, W., and Sadeghi, A.-R. ASM: A Programmable Interface for Extending Android Security. In *Proc. 23rd USENIX Security Symposium (SEC'14)* (2014), USENIX Association.

[38] Jeon, J., Micinski, K. K., Vaughan, J. A., Fogel, A., Reddy, N., Foster, J. S., and Millstein, T. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. In *Proc. 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'12)* (2012), ACM.

[39] Kaspersky Lab, and INTERPOL. Mobile cyber-threats. http://securelist.com/analysis/publications/66978/mobile-cyber-threats-a-joint-study-by-kaspersky-lab-and-interpol/, 2014. Last visited: 02/19/15.

[40] Lange, M., Liebergeld, S., Lackorzynski, A., Warg, A., and Peter, M. L4android: A generic operating system framework for secure smartphones. In *Proc. 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'11)* (2011), ACM.

[41] Microsoft. Application Virtualization (App-V). http://www.microsoft.com/en-us/windows/enterprise/products-and-technologies/mdop/app-v.aspx. Last visited: 02/13/15.

[42] Moulu, A. Android OEM's applications (in)security and backdoors without permission. http://www.quarkslab.com/dl/Android-OEM-applications-insecurity-and-backdoors-without-permission.pdf. Last visited: 02/19/15.

[43] Mulliner, C., Oberheide, J., Robertson, W., and Kirda, E. PatchDroid: Scalable Third-party Security Patches for Android Devices. In *Proc. 29th Annual Computer Security Applications Conference (ACSAC'13)* (2013), ACM.

[44] Nauman, M., Khan, S., and Zhang, X. Apex: Extending android permission model and enforcement with user-defined runtime constraints. In *Proc. 5th ACM Symposium on Information, Computer and Communication Security (ASIACCS'10)* (2010), ACM.

[45] Ongtang, M., McLaughlin, S. E., Enck, W., and McDaniel, P. Semantically Rich Application-Centric Security in Android. In *Proc. 25th Annual Computer Security Applications Conference (ACSAC'09)* (2009), ACM.

[46] Open Signal. Android Fragmentation Visualized (July 2013). http://opensignal.com/reports/fragmentation-2013/. Last visited: 02/06/2015.

[47] Provos, N. Improving host security with system call policies. In *Proc. 12th Conference on USENIX Security Symposium - Volume 12 (SSYM'03)* (2003), USENIX Association.

[48] Provos, N., Friedl, M., and Honeyman, P. Preventing privilege escalation. In *Proc. 12th Conference on USENIX Security Symposium - Volume 12 (SSYM'03)* (2003), USENIX Association.

[49] Rasthofer, S., Arzt, S., Lovat, E., and Bodden, E. DroidForce: Enforcing Complex, Data-Centric, System-Wide Policies in Android. In *Proc. 9th International Conference on Availability, Reliability and Security (ARES'14)* (2014), IEEE Computer Society.

[50] Rastogi, V., Chen, Y., and Jiang, X. DroidChameleon: Evaluating Android Anti-malware Against Transformation Attacks. In *Proc. 8th ACM Symposium on Information, Computer and Communication Security (ASIACCS'13)* (2013), ACM.

[51] Russello, G., Conti, M., Crispo, B., and Fernandes, E. MOSES: supporting operation modes on smartphones. In *Proc. 17th ACM Symposium on Access Control Models and Technologies (SACMAT'12)* (2012), ACM.

[52] Russello, G., Jimenez, A. B., Naderi, H., and van der Mark, W. FireDroid: Hardening Security in Almost-stock Android. In *Proc. 29th Annual Computer Security Applications Conference (ACSAC'13)* (2013), ACM.

[53] Samsung Electronics. White paper: An overview of samsung KNOX. http://www.samsung.com/se/business-images/resource/2013/samsung-knox-an-overview/%7B3%7D/Samsung_KNOX_whitepaper-0-0-0.pdf, 2013. Last visited: 02/19/15.

[54] Smalley, S., and Craig, R. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Proc. 20th Annual Network and Distributed System Security Symposium (NDSS'13)* (2013), The Internet Society.

[55] von Styp-Rekowsky, P., Gerling, S., Backes, M., and Hammer, C. Idea: Callee-site rewriting of sealed system libraries. In *Proc. 5th International Symposium on Engineering Secure Software and Systems (ESSoS'13)* (2013), Springer.

[56] Wangy, X., Sun, K., and Jing, Y. W. J. DeepDroid: Dynamically Enforcing Enterprise Policy on Android Devices. In *Proc. 22nd Annual Network and Distributed System Security Symposium (NDSS'15)* (2015), The Internet Society.

[57] Watson, R. N. M., Anderson, J., Laurie, B., and Kennaway, K. Capsicum: Practical capabilities for unix. In *Proc. 19th USENIX Security Symposium (SEC'10)* (2010), USENIX Association.

[58] Wu, C., Zhou, Y., Patel, K., Liang, Z., and Jiang, X. Airbag: Boosting smartphone resistance to malware infection. In *Proc. 21st Annual Network and Distributed System Security Symposium (NDSS'14)* (2014), The Internet Society.

[59] Xu, R., Saïdi, H., and Anderson, R. Aurasium – Practical Policy Enforcement for Android Applications. In *Proc. 21st USENIX Security Symposium (SEC'12)* (2012), USENIX Association.

[60] Xu, Z., Bai, K., and Zhu, S. Taplogger: inferring user inputs on smartphone touchscreens using on-board motion sensors. In *Proc. 5th ACM conference on Security and Privacy in Wireless and Mobile Networks (WISEC'12)* (2012), ACM.

[61] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native client: A sandbox for portable, untrusted x86 native code. In *Proc. 30th IEEE Symposium on Security and Privacy (Oakland'09)* (2009), IEEE Computer Society.

[62] Zhou, Y., and Jiang, X. Dissecting Android malware: Characterization and evolution. In *Proc. 33rd IEEE Symposium on Security and Privacy (Oakland'12)* (2012), IEEE Computer Society.

[63] Zhou, Y., Wang, Z., Zhou, W., and Jiang, X. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proc. 19th Annual Network and Distributed System Security Symposium (NDSS'12)* (2012), The Internet Society.

[64] Zhou, Y., Zhang, X., Jiang, X., and Freeh, V. Taming information-stealing smartphone applications (on Android). In *Proc. 4th International Conference on Trust and Trustworthy Computing (TRUST'11)* (2011), Springer.

Cookies Lack Integrity: Real-World Implications

Xiaofeng Zheng[1,2,3], Jian Jiang[7], Jinjin Liang[1,2,3], Haixin Duan[1,3,4], Shuo Chen[5], Tao Wan[6], and Nicholas Weaver[4,7]

[1]Institute for Network Science and Cyberspace, Tsinghua University
[2]Department of Computer Science and Technology, Tsinghua University
[3]Tsinghua National Laboratory for Information Science and Technology
[4]International Computer Science Institute
[5]Microsoft Research Redmond
[6]Huawei Canada
[7]UC Berkeley

Abstract

A cookie can contain a "secure" flag, indicating that it should be only sent over an HTTPS connection. Yet there is no corresponding flag to indicate how a cookie was set: attackers who act as a man-in-the-midddle even temporarily on an HTTP session can inject cookies which will be attached to subsequent HTTPS connections. Similar attacks can also be launched by a web attacker from a related domain. Although an acknowledged threat, it has not yet been studied thoroughly. This paper aims to fill this gap with an in-depth empirical assessment of cookie injection attacks. We find that cookie-related vulnerabilities are present in important sites (such as Google and Bank of America), and can be made worse by the implementation weaknesses we discovered in major web browsers (such as Chrome, Firefox, and Safari). Our successful attacks have included privacy violation, online victimization, and even financial loss and account hijacking. We also discuss mitigation strategies such as HSTS, possible browser changes, and present a proof-of-concept browser extension to provide better cookie isolation between HTTP and HTTPS, and between related domains.

1 Introduction

The same-origin policy (SOP) is a corner stone of web security, guarding the web content of one domain from the access from another domain. The most standard definition of "origin" is a 3-tuple, consisting of the scheme, the domain and the port number. However, the notion of "origin" regarding cookies is fairly unusual – cookies are not separated between different schemes like HTTP and HTTPS, as well as port. The domain isolation of cookie is also weak: different but related domains can have a shared cookie scope. A cookie may have a "secure" flag, indicating that it should only be presented over HTTPS, ensuring confidentiality of its value against a network man-in-the-middle (MITM). However, there is no similar measure to protect its integrity from the same adversary: an HTTP response is allowed to set a secure cookie for its domain. An adversary controlling a related domain is also capable to disrupt a cookie's integrity by making use of the shared cookie scope. Even worse, there is an asymmetry between cookie's read and write operations involving pathing, enabling more subtle form of cookie integrity violation.

The lack of cookie integrity is a known problem, noted in the current specification [2]. However, the real-world implications are under-appreciated. Although the problem has been discussed by several previous researchers [4, 5, 30, 32, 24, 23], none provided in-depth and real-world empirical assessment. Attacks enabled by merely injecting malicious cookies could be elusive, and the consequence could be serious. For example, a cautious user might only visit news websites at open wireless networks like those at Starbucks. She might not know that this is sufficient for a temporary MITM attacker to inject malicious cookies to poison her browser, and compromise her bank account when she later logs on to her bank site at home.

We aim to understand how could attackers launch cookie inject attacks, and what are the damaging consequences to real-world websites. Our study shows that most websites are potentially susceptible to cookie injection attacks by network attackers. For example, only one site in the Alexa top 100 websites has fully deployed HTTP Strict Transport Security (HSTS) on its top-level domain, a sufficient server-side protection to counter cookie injection attacks by network attackers (Section 3). We also found a number of browser vulnerabilities and implementation quirks that can be exploited by cookie injection attacks (Section 4). Notably, all major browsers, except Internet Explorer (IE), respect the "Set-Cookie" header in a 407-response (i.e., an Authentication Required Response) when configured to use a proxy. Because of this vulnerability, even websites adopting sufficient HSTS are subject to cookie injection attacks by a malicious proxy.

Our study also shows that current cookie practices have widespread problems when facing cookie injection attacks (Section 5). We demonstrate multiple exploitations against large websites. For example, we show that an attacker can put his Gmail chat gadget on a victim's screen without affecting the victim's use of Gmail and other Google services. We also demonstrate that an attacker can hijack a victim's online deposit to his account, or even deliver the victim's online purchase to his address. Other exploitations include user tracking, cross-site scripting (XSS) attacks against large financial sites embedded in injected cookies, *etc.*.

We have developed a mitigation strategy (Section 6). By modifying how browsers treat secure cookies, it is possible to largely mitigate cookie injection attacks by network attackers. We have also considered possible browser enhancements to mitigate cookie injection from web attackers. We implement our proposals as a proof-of-concept browser extension. A preliminary evaluation does not encounter compatibility issues.

In summary, this work makes the following main contributions:

- We provided an evaluation of potential susceptible websites to cookie injection attacks, including a detailed measurement of full HSTS adoption and an assessment of shared domains used by Content Delivery Networks (CDNs).

- We examined both browser-side and server-side cookie implementation, in which we found several browser vulnerabilities and a number of non-conforming and/or inconsistent implementations that could be exploited in cookie injection attacks.

- We demonstrated the severity and prevalence of cookie injection attacks in the real world. In particular, our exploitations against a variety of large websites show that cookie injection enables complicated interactions among implements, applications, and various known attacks.

- We developed and implemented browser-side enhancements to provide better cookie isolation. Our evaluation showed promising results in compatibility.

Together, this work provides a close-up picture of the cookie integrity problem and the threats of cookie inject attacks. We intend to provide a context for motivating further discussion in research community and industry.

2 Background

2.1 Cookies

Cookies are a browser-side assisted state management mechanism that are pervasively used by web applications [2]. Cookies can be set by either HTTP servers using "`Set-Cookie:`" header or client side JavaScript with a write to "`document.cookie`". A cookie can have five optional attributes: `domain` and `path` specifying the cookie's scope; `expires` stating when it should be discarded; `secure` specifying that it should only be sent over HTTPS connections, and `HTTPOnly` preventing browser-side scripts from reading the cookie. When sending a request to a server, a web browser includes all unexpired cookies whose domains and paths match the requested URL, excluding those marked as `secure` from the inclusion in an HTTP request.

Cookies have two fairly unusual behaviors. First, there is a critical disconnection between cookie storage and reading. Cookies are set and stored as a name/domain/path to value attributes mapping, but only name-value pairs are presented to both JavaScript and web servers. This asymmetry allows cookies with the same name but different domain and/or path scopes to be written into browser; a subsequent reader can read out all same name cookies together, yet cannot distinguish them because the other attributes such as path are not presented in the reading process. Another complication occurs when writing a cookie, the writer can specify arbitrary value for the `path` attribute, not limited by the URL of the writer's context.

Moreover, the security policy for cookies is not as stringent as the classic SOP. In web security, the SOP is the most important access control mechanism to segregate static contents and active scripts from different origins [3]. An origin for a given URL is defined by a 3-tuple: scheme (or protocol), *e.g.* HTTP or HTTPS, domain (or host), and port (not supported by IE (Internet Explorer)). However, the security policy guarding cookies does not provide separation based on either scheme or port but only on domain [2]. In addition, a website can set cookies with flexible domain scopes: 1) not shared (*i.e.*, host-only), 2) shared with its subdomains, or 3) shared with its sibling domains (i.e., using its parent domain as the scope). For the third case, a restriction is enforced by browser to ensure that a cookie domain scope is not "too wide". For example, `www.example.com` can set a cookie with the scope of `.example.com`, but it cannot set a cookie with `.com` as the scope because `.com` is a public top level domain (TLD). Unfortunately, there is no clear definition of whether a domain scope is "too wide" (See Section 3.2).

The combination of the read/write asymmetry and the lack of domain or scheme segregation implies that a domain cannot protect the integrity of its cookie from an active MITM or a malicious/compromised *related domain* that shares some cookie domain scope with it. There are two forms of cookie integrity violations:

- **Cookie Overwriting.** If a cookie shares the domain scope with a related domain, it can be directly overwritten by that domain using another cookie with the exactly same name/domain/path. Of particular note, although a `secure` cookie can only be read by an HTTPS process, it can be written or overwritten by an HTTP request.
- **Cookie Shadowing.** Alternatively, an attacker with the control of a related domain can intentionally *shadow* a cookie by injecting another one that has the same name, but different domain/path scope. For example, to shadow a cookie with "value=good; domain=www.example.com; path=/; secure", a related domain `evil.example.com` can write a cookie with "value=bad; domain=.example.com; path=/home". Later, when browser issues a request to `https://www.example.com/home`, both cookies match the URL and are included. For most browsers, the cookie header will be "Cookie: value=bad; value=good;". The "good" cookie could be shadowed by the "bad" one if a website happens to prefer the value of "bad" over "good".

  Note while the "good" cookie has a `secure` flag and is sent over HTTPS, it can still be shadowed with a cookie set from an HTTP connection.

2.2 HSTS

HSTS (HTTP Strict Transport Security) allows a server to inform a client to only initiate communications over HTTPS. It was originally proposed by Jackson and Barth to address a number of MITM threats such as cookie sniffing and SSL stripping [18], and is now standardized in RFC6797 as a HTTP response header `Strict-Transport-Security` [15].

The HSTS header requires a `max-age` attribute indicating how long a browser should keep the HSTS policy for that domain. An optional attribute `includeSubDomains` tells a browser to apply the HSTS policy to its all subdomains. After receiving an HSTS header, a conforming browser ensures that all subsequent connections to that domain always take place over HTTPS until the policy expires. Chrome and Firefox also support a preloaded list that contains self-declared websites supporting HSTS. For more information on HSTS, please see [22].

HSTS coverage can often be incomplete. For example, if `example.com` does not specify `includeSubDomains` in its HSTS header, a browser will allow HTTP connection to `foo.example.com`. Worse, even if the HSTS policy of `example.com` specifies `includeSubDomains`, this will not be checked by a browser if a user only visits `bar.example.com` unless the page includes a reference to `example.com`.

2.3 Cookie Injection Attacks

It is a known vulnerability that cookies can be injected by HTTP response into subsequent HTTPS request, and from one domain to another related domain. Johnston and Moore reported such problem in 2004 [19]. Their report already pinpointed the root cause: the loosely defined SOP for cookies. Unfortunately browsers vendors did not fix the problem probably because they were concerned of potential incompatibility issues. In 2008, Evans described an attack called *cookie forcing* that exploits cookie integrity deficiency to overwrite cookies in HTTPS sessions [7]. In 2013, GitHub migrated their domain for hosting users' homepages from `github.com` to `github.io` after they recognized the threat of cookie injection from/to a *shared domain* whose subdomains belong to mutually untrusted users; they described detailed steps of several possible cookie injection exploits and referred to them as *cookie tossing* [11].

The problem was also noted in several more formal publications. Barth *et al.* discussed security implications of cookie overwriting on session initiation [4]. They also proposed a new header `Cookie-Integrity` to provide additional information so that web server can distinguish between cookies set from HTTP and those set from HTTPS. Bortz *et al.* also reviewed the problem and proposed a new header `Origin-Cookie` that guarantees integrity by enforcing a complete 3-tuple SOP [5]. Singh *et al.* referred the difference between the classic SOP and the cookie SOP as inconsistent principal labeling [30]. Both Zalewski's book [32] and the current cookie specification by Barth [2] explained the cookie integrity deficiencies in great detail. We also learned of two technical reports, one from Black Hat EU by Lundeen [23] and the other from Black Hat AD by Lundeen *et al.* [24], that illustrated several subtle attacks initiated by cookie injection.

Although a known threat, previous research fall short of in-depth empirical assessment of its real-world security implications. This work aims to fill this gap. We provide a detailed comparison in Section 7.

3 Threat Analysis

We first present the threat model for cookie injection attacks. For each type of attacker, we analyze its real-world threat. Table 1 gives an overview.

Attacker		Root Cause	Attack Surface	Mitigation
Network Attacker	Active MITM	SOP without protocol & complete domain isolation.	Websites and browsers that allow attackers to reply an unencrypted request to a related domain with forged response.	Full HSTS
	Malicious Proxy			
Web Attacker	Full control of related domain	SOP without complete domain isolation.	Websites using shared domains.	Public suffix list
	XSS on related domain		Websites with compromised related domains.	Out of scope

Table 1: Overview of the threats of cookie injection attacks

3.1 Threat Model

Two classes of attackers can manipulate a target site's cookies: an active network adversary or a remote adversary able to host or inject content on a related domain.

The active MITM attacker (including the classic MITM fully controlling the network and the Man-on-the-Side (*i.e.*, wiretapping and packet-injecting)) can load arbitrary cookies through HTTP into the target's cookie store. The attacker modifies an unrelated HTTP request to create a hidden iframe in a web page. The attacker's iframe then creates a series of HTTP fetches to the target domains, which the attacker responds to with `Set-Cookie` headers to poison the victim's cookie store.

A malicious proxy is at least as powerful as an active MITM in terms of manipulating network traffic. Moreover, because the browser has extra protocol interactions with the proxy, potential logic flaws or implementation bugs might give the malicious proxy additional chances to break in. Chen *et al.* highlighted this threat with a number of logic flaws [6]. Our study also targets this type of issues related to unexpected capabilities for a malicious proxy to inject cookies.

Finally, if an attacker controls a related domain directly, he may launch cookie injection remotely. The attacker does not need full control of the related server, just the ability to host JavaScript. This attacker cannot target arbitrary domains, but can target any other domain under the same "top level" domain.

One key property of all these adversaries is its ability to change state. For example, a victim might only visit her bank from known-good networks, but an attacker can poison the victim's browser when the victim is on an open wireless network. Only later, when the victim has now returned to the "safe" network and visits her bank, does the attack actually affect the victim.

3.2 Attack Surface

Network Attack Surface: The only current protection against an active network attacker requires that the victim's browser *never* issues an unencrypted HTTP connection to a target site or any related domain. This condition holds if 1) the target domain enables HSTS on its *base domain* [1] (*i.e.* the first upper-level domain that is considered "non-public") with the `includeSubDomains` option, which we refer to as *full HSTS*; 2) the browser supports HSTS; and 3) the browser has received the full HSTS policy from the base domain of the target domain.

	Domain Ranking					
	<10	10-10^2	10^2-10^3	10^3-10^4	10^4-10^5	$> 10^5$
Valid HTTPS	7	52	353	2,914	20,548	128,805
Full HSTS	0	1	7	35	212	997

Table 2: Ranking distribution of domains with valid HTTPS and full HSTS.

Unfortunately, the support and adoption of HSTS in the real world is unsatisfactory. First, all current versions of IE, a major browser with considerable marketshare, do not support HSTS (Microsoft announced that its new browser will support HSTS [16]). Second, there is limited adoption of full HSTS among sites. We scanned 961,857 base domains from the Alexa top one million websites and also examined if these domains present in the Chrome's preloaded HSTS list [28]. While we observed 152,679 (15.87%) domains have deployed HTTPS with valid certificates, we only found 1,252 (0.13%) domains have enabled full HSTS. Moreover, most of the full HSTS domains are low ranked domains (see Table 2). A recent study by Kranch and Bonneau also presented a similar total number of full HSTS domains among the Alexa top one million websites [22].

Because of the prevalence of unsafe networks like open wireless networks and the very limited deployment/availability of full HSTS protection, we consider cookie injection by active network attackers a pervasive and severe threat, especially for websites who have deployed HTTPS to prevent active network attackers from launching other possible attacks such as eavesdropping or active script injection, yet have not enabled full HSTS.

Web Attack Surface: Generally, a web attacker might be able to control a related domain in two ways. First, for large websites that all subdomains are used internally, an attacker can fully control one subdomain by compromising its DNS resolution or its hosting server. The attacker can also exploit a XSS vulnerability on a subdomain of a large website. A cookie injection attack can then be launched to target other subdomains.

A greater concern is when a website either hosts user content or shares a domain scope with other possibly untrustworthy sites. This problem is inherent from the

[1] We learned this term from Kranch and Bonneau's recent HSTS study [22].

weaker cookie SOP. As we previously discussed in Section 2.1, a domain is allowed to set cookies with wider domain scope as long as the scope is not considered public. Hence, a clear boundary between "public" and "non-public" domain scope is needed to prevent cookie injection from undesired shared cookie domain. However, this is not easy to define and implement clearly. First, many top-level domains (a.k.a., TLDs), especially country code top-level domains (a.k.a., ccTLDs) have their own reserved suffixes such as `.com.cn`, `.co.uk`, which are mostly TLD-specific. Second, many websites use *shared domains* to assign subdomains to their mutually untrusted clients. Such shared domain providers include cloud hosting providers, web hosting providers, blog providers, CDN providers etc. These shared domains should also be considered as non-public in terms of cookie domain scope.

The problem of cookie domain scope boundary is partially remedied by a community effort initiated by Mozilla called "public suffix list", which maintains an exceptional list containing TLDs, TLD-reserved suffixes, and self-declared shared domains [25]. Public information suggests that the list is enforced by major browser vendors including IE, Chrome, Firefox, and Opera, while our own tests confirm that Safari implements this list.

Our study of the public suffix list shows that the public shared domains list still exposes an attack surface for cookie injection. First, we empirically identified 45 shared domains from the Alexa top one million websites, among which only 10 Google domains and 3 non-Google domains are included in the public suffix list. Among the remaining domains, we found at least 4 domains (`sinaapp.com`, `weebly.com`, `myshopify.com`, and `forumotion.com`) allow customized server-side code or browser-side scripts. Websites hosting on these domains are vulnerable to cookie injection attacks.

Another easy-to-miss corner case is shared domains used by CDNs. CDNs commonly assign subdomains or sub-directories of shared domains to their customers. If a website directly uses a shared domain assigned by its CDN provider, and the CDN provider does not handle the shared domain carefully, then the website is subject to cookie injection attacks from malicious customers of the same CDN provider. While websites rarely use shared domains as their main domains, a common practice is to refer static resources (*e.g.*, JavaScript files, images) using shared domain URLs. Although cookies under these resource URLs are usually not processed by server-side code or browser-side scripts, cookie injection attacks could still cause serious consequences. For example, suppose both websites A and B host their static resource files under one shared domain from the same CDN. Website A can inject garbage cookies from the requests to his resource files with specific paths so that the injected cookies will be sent with the requests of resource files to website B. This type of cookie injection attack could cause performance downgrade, bandwidth consumption, and even denial-of-service (DoS) if the amount of injected cookies exceeds the server's header size limitation[2]. In worst case, DoS of a critical resource file like a JavaScript library could break the whole website.

Vendor	Domain	Publis Suffix List?	Vulnerable?
Akamai	akamai.net	No	n/a [1]
	akamaiedge.net	No	n/a
	akamaihd.net	No	n/a
	edgesuite.net	No	n/a
Azure	msecnd.net	No	Yes
	windows.net	No	Yes
BitGravity	bitgravity.com	No	n/a
CacheFly [2]	cachefly.net	No	Yes
CDN77	cdn77.net	No	Yes
CDNetworks	cdngc.net	No	n/a
CDN.net	worldcdn.net	No	n/a
ChinaCache	chinacache.net	No	n/a
ChinaNetCenter	wscloudcdn.com	No	n/a
CloudFlare [3]	cloudflare.net	No	Yes
CloudFront	cloudfront.net	Yes	No
EdgeCast	edgecasecdn.net	No	n/a
Exceda	expresscdn.com	No	Yes
Fastly [3]	fastly.net	Yes	Yes
Highwinds	hwcdn.net	No	n/a
Incapsula	incapsula.net	No	Yes
Internap	internapcdn.net	No	n/a
Jiasule	jiashule.com	No	Yes
KeyCDN [2]	kxcdn.com	No	Yes
Level3	footprint.net	No	n/a
Limelight	linwd.net	No	n/a
MaxCDN	netdna-cdn.com	No	Yes
Squixa	squixa.net	No	n/a

1: "n/a" refers to the case that we were not able to test.
2: CDNs attempting to defend cookie related attacks on shared domains by filtering the `Set-Cookie` header.
3: CDNs allowing shared cookie scopes in customer-specific prefixes of shared domains.

Table 3: Assessment of cookie injection attacks on shared domains used by CDNs.

We empirically collected 28 shared domains used by 23 CDNs [3], in which only 2 domains are registered in the public suffix list, as presented in Table 3. We were also able to sign up and test 13 shared domains from 12 CDNs. While we confirmed that `cloudfront.net` is immune because of its presence in the public suffix list, for each of the other 12 domains, we successfully launched DoS attack on one test URL by injecting 72KB cookies from another test URL. Our experiments also found two problematic behaviors. First, CacheFly and KeyCDN attempt to defend cookie related attacks by filtering the `Set-Cookie` header in response instead of utilizing the public suf-

[2] Although the current HTTP specification does not define any limitation on the size of request header [9], most of web server implementations do so by default. For example, nginx by default limits a single HTTP header not to exceed 8KB [26].

[3] We collected most of the CDNs from `http://www.cdnplanet.com/cdns/`.

fix list, which fails to prevent JavaScript from injecting cookies. Second, although Fastly has declared several subdomains of `fastly.net` as public suffix, its naming mechanism enables shared scopes in customer-specific prefixes, making its customers still vulnerable to cookie injection attacks. For example, for a customer `foo.com`, Fastly assigns a customer subdomain `foo.com.global.prod.fastly.net`. Although the suffix `global.prod.fastly.net` is present in the public suffix list, the prefix causes a cookie scope `com.global.prod.fastly.net` shared with other customer subdomains such as `bar.com.global.prod.fastly.net`. CloudFlare also has the same problem. We have reported this problem to all vulnerable vendors. CloudFlare and CDN77 have acknowledged our reports. The response from CloudFlare said that they are considering to disable direct access of all `cloudflare.net` URLs to defend against this problem.

4 Pitfalls in Cookie Implementations

Based on the threat model and the understanding of potential attack surfaces, we then turn to understand how cookie related mechanisms are implemented in browsers and web applications. Our study pinpointed a number of inconsistent and/or non-conforming behaviors in major browsers and web frameworks, as summarized in Table 4. We also identified several vulnerabilities in major browsers allowing an active network attacker to inject cookies even when the full HSTS is deployed. We have reported these vulnerabilities to browser vendors.

4.1 Uncovered Implementation Quirks

Browser-side Cookie Ordering. The current cookie specification [2] suggests that browsers should rank cookies first by path specificity and then by the creation time in ascending order. We found all major browsers follow this suggestion except Safari, which ranks cookies first by the specificity of the domain attribute then by the path specificity.

Server/script-side Cookie Preference. The cookie header is semantically a list. For the same name cookies in the list, the specification states that the server should not rely upon cookie's ordering presented by the browser. We examined popular web programming languages, web frameworks, and third-party libraries including PHP, Python, Java, Go, ASP, ASP.NET, JavaScript, Node.js, JQuery, JSF, SpringMVC. At the language level, only Java, JavaScript and Go provide built-in or standard library interfaces to read cookies as a list. Other languages, and all web frameworks and third-party libraries treat the cookie list as a name-value map that only returns one value for each cookie name in the list. For cookies with the same name, while the name-value map interface in Python standard library prefers the last-ordered cookie, all others prefers the first-ordered one. This explains why cookie shadowing is possible and the example given in Section 2.1 works in many cases.

Cookie Storage Limitation. The specification has several vague suggestions for browsers to limit the number and size of stored cookies. We found all major browsers set the maximum size of a single cookie to 4 KB. Chrome, Firefox, and Opera implements a cookie jar for every base domain, with the total numbers of cookies limited to 180, 150, and 180, respectively. IE's cookie jar implementation is per cookie domain scope, with the total number of cookies limited to 50. We did not reach Safari's cookie storage limit after writing and reading 1,000 cookies.

Cookie Header Size Limit. While Safari does not seem to have a limit for the number of cookies, it truncates the matching cookie list if the length of the cookie header exceeds 8 KB. We did not observe similar behaviors in other browsers.

Cookie Name. The cookie name can contain all US-ASCII values except control characters and separator characters (see definition in [2] and [8]). We found that Safari mistakenly stores cookie name in case-insensitive manner. Some programming languages also implement cookie names incorrectly. Previously Lundeen *et al.* reported that ASP.NET implements cookie names case-insensitively [24]. We found that ASP makes same mistake. In addition, PHP performs percent-decoding on cookie names. For these languages, different cookie names sent by browser are possibly recognized as same name cookies, which embraces another vector for cookie shadowing. For example, PHP interprets a cookie header "%76alue=bad; value=good;" as "value=bad; value=good;", causing the "good" cookie to be shadowed by the "bad" one.

Cookie Path. According to the specification, a cookie matches a URL only when the path scope of the cookie is a sub-directory of (or identical to) the URL path. When a cookie does not specify the `path` scope, the browser is required to set its default path as the directory-portion of the URL path without any trailing slash. We found 4 violations to the standard: 1) Safari[4] implements a substring other than sub-directory matching rule; 2) Firefox and IE match cookie path with not only the URL path, but also the URL query and the URL fragment portion match; 3) Firefox matches a cookie path with a URL path when the former has one more slash than the later; 4) Chrome, Safari, and Opera (Linux and iOS versions) include the trailing slash in default cookie path.

[4] Also Chrome on iOS, but as iOS browsers need to use Apple's rendering engine rather than their own, this is probably due to Apple's decision, not Google's

Cookie Property	Specification	Non-conforming/inconsistent behaviors
Browser-side priority	Cookies SHOULD be ranked by specificity of path then by creation time in ascending order.	1. Safari ranks cookies by specificity of domain then by specificity of path.
Server/script-side preference	Server SHOULD NOT rely on cookie's ordering presented by browsers.	1. Most standard libs and frameworks only provide name-value map interfaces; 2. For each name in the cookie list, Python prefers the last-ordered cookie, others prefer the first-ordered one.
Cookie storage limitation	Several vague suggestions	1. Safari seemingly does not have limitation on the number of stored cookies; 2. Chrome and Firefox limit the size of the cookie store per base domain, IE does so per specific domain scope.
Cookie header size limitation	Not specified	1. Safari truncates the cookie header not to exceed 8,192 bytes.
Cookie name	US-ASCII values except control characters and separator characters (see definition in [2] and [8])	1. Safari is case-insensitive with cookie name; 2. ASP and ASP.NET are case-insensitive; 3. PHP performs percent-decoding on cookie name.
Cookie path	1. Cookie path and URL path MUST be identical or sub-directory matching; 2. Trailing slash MUST NOT be included in default cookie path.	1. Firefox and IE matches cookie path not only with URL path, but also with URL query and URL fragments; 2. Safari implements sub-string matching other than sub-directory matching; 3. Firefox allows cookie path has one more slash than the URL path; 4. Chrome, Safari, and Opera under some platforms include trailing slash in the default cookie path.

Table 4: Summaries of non-conforming and inconsistent behaviors found in browser and web server cookie implementations.

4.2 Uncovered Vulnerabilities

Vulnerabilities in Handing Proxy Response. In [6], Chen *et al.* found a number of flaws in major browsers. The root problem resided in the handling of HTTPS responses. Essentially, all browsers at that time could not differentiate an HTTPS response from a proxy and an HTTPS response from the intended server. The flaws were patched after disclosure. However, we found the patches are incomplete: if a proxy replies to a HTTPS `CONNECT` request with an unencrypted 407 (proxy authentication required) response, all major browsers except IE accept the cookies set in 407 response. While some vulnerable browsers display a pop-up window, some accept cookies silently (Table 5).

	Windows	Mac OS	Linux	Android	iOS
IE	–	N/A	N/A	N/A	N/A
Chrome	①	①	①	①	①
Firefox	②	②	②	②	N/A
Safari	②	③	N/A	N/A	②
Opera	①	①	N/A	①	N/A

①: cookie injection with pop-up window.
②: cookie injection without pop-up window.
③: cookie injection and script injeciton.

Table 5: Browser vulnerabilities in handling 407 response by a malicious proxy.

These vulnerabilities allow a malicious proxy to launch cookie injection attacks against a full HSTS site. Users who use proxies or have them set automatically, these vulnerabilities can also be exploited by an active MITM between the victim and the proxy, even if a victim user does not intentionally use the attacker as the proxy.

Vulnerability in Handing Public Suffixes in Safari. As described in Section 3.2, the public suffix list enforces the boundary between public and non-public cookie domain scopes. However we found the implementation of Safari is vulnerable under certain conditions. When Safari issues a request `http://tld/`, it accepts cookies in the response with domain scope as `.tld`, which are shared by all `subdomains.tld`. Because HSTS is not enabled on an entire TLD (in general, there is no A record indicating a server at the TLDs), this vulnerability is exploitable by active network attackers who can forge a DNS response as well as an HTTP response.

Vulnerability in Safari's HSTS Implementation. We also found a vulnerability in Safari's HSTS implementation. When receiving a URL, Safari does percent-decoding and upper-to-lower case conversion on its domain name before issuing a request. However, the HSTS check is performed before the conversion process completes, enabling an attacker to bypass Safari's HSTS check if both capital and percent-encoding are used in the domain name.

5 Real-World Exploitations

Our study aims at understanding the prevalence and severity of potential exploitation by cookie injection in real-world websites. In particular, we are curious about how web developers use cookies, whether they are aware of this problem explicitly and have developed best practices accordingly. With these questions in mind, we conducted black box penetration tests on a number of popular websites with our test accounts. We also reviewed several well-known open source web applications. For penetration tests, we first used browser extensions like EditThisCookie [1] to test manually. For possible exploitations, we then implemented with Bro [27] (for packet sniffing and injection with the `rst` tool) in an open wireless network setting.

We found cookie injection attacks are possible with very large websites and popular open source applications including Google, Amazon, eBay, Apple, Bank of America, BitBucket, China Construction Bank, China UnionPay, JD.com, phpMyAdmin, and MediaWiki, among others. The consequences of attacks include, but are not limited to, XSS, privacy leakage, bypassing of cross-site request forgery (CSRF) defenses, financial loss, and account hijacking. The varieties of vulnerable web applications and exploitations suggest cookie injection is a serious threat in the real world, and deserves a greater attention from the web security community.

The exploitations we found indicate three common cookie usages: 1) using cookies as authentication tokens; 2) associating important and session independent states with cookies; 3) reflecting cookies into HTML. These cookie usages often lead to cookie injection attacks if specific defensive measures are not in place.

We present our exploitations based on these categories, along with the necessary background and additional observations. Please refer Section 4 and Table 4 for the details of different cookie implementations involved in some cases. We extensively make use of cookie shadowing. For these cases, unless otherwise specified, we assume that the web server has the common behavior of preferring the first-ordered cookie for each name in the cookie list.

5.1 Cookies as Authentication Tokens

A common practice in web development is to use a cookie to identify a user session. Many websites further set long expiration durations on session cookies to avoid having users sign in every time. This practice itself is somewhat questionable, because session cookies are sent along with HTTP requests automatically, which facilitate CSRF attacks. Nevertheless, Barth *et al.* showed that CSRF attacks can be defeated with specific defensive principles and techniques in web applications [4].

Also in [4], Barth *et al.* noted a special form of CSRF which they called *login CSRF*. In this attack, an attacker signs in with his own account on the victim's browser. If not noticed, the victim might visit targeted web site on behalf of the attacker's account, resulting in security and privacy consequences such as search history leakage, credit card stealing, and XSS. The authors also pointed out that login CSRF is a special form of a threat they called *Authenticated-as-Attacker*, which can also be carried out by injecting malicious session cookies to overwrite original ones.

In fact, the consequences of cookie injection on session cookies can go beyond those described in [4]. We found that, by using cookie shadowing, similar attacks could be carried out without noticeable evidences by the victim. We call our attacks *sub-session hijacking attacks*.

5.1.1 Exploiting Google Chat and Search

We first present two exploits targeting Google, which lead our observation of the sub-session hijacking attack. Google's base domain `google.com` is not protected with full HSTS, so in most cases it is subject to cookie injection by an active network attacker.

Case-1: Gmail chat gadget hijacking. The web interface of Gmail at `https://mail.google.com/` shows a chat gadget at the bottom left corner. If an attacker hijacks the gadget without affecting Gmail and other Google services, he can fake the victim's friend list and chat with the victim to initiate advanced phishing, intercept communication, or perform other disruptive activity. This could be particularly deceptive in a targeted attack scenario.

We have confirmed this attack. Although the browser displays everything as one page, the chat gadget and Gmail content are actually loaded with different URLs then composed together. Both the chat gadget and Gmail use cookies for authentication. If an attacker injects his Google session cookies in a way that the injected cookies shadow the original ones only at the chat gadget related URLs, then the attacker can put his chat gadget on the victim's screen, without disturbing the victim's use of Gmail and other Google services.

We demonstrated this attack by injecting a total of 25 cookies: five session cookies "SID/SSID/HSID/APISID/SAPISID", each with five specific paths. Meanwhile most Google services are not affected because the specific paths of the injected cookies do not match with their URL paths. This is sufficient to cause the chat window to load with the attacker's cookies, while all other components are loaded as the victim.

Case-2: "Invisible" Google search history stealing. Another attack is to use cookie shadowing to steal Google search history (which is automatically logged and retrievable with the login cookie) without being noticed. We assume that a user has visited `https://www.google.com/`, which shows the search box and her profile name and icon. When she types in the search box, browser-side script issues AJAX requests to `https://www.google.com/search` to get search results.

Our original goal was to only shadow the session cookies of the AJAX request, so that we could steal search history without affecting the web interface loaded by `https://www.google.com/`. But it turned out we could not achieve this. We first injected three relevant session cookies "SID/SSID/HSID" with path "/search". However, this attempt failed because we found the server unusually preferred the last-ordered cookie, and the injected cookies were ranked before the legitimate ones because of the specific path. We then found out a way to only shadow the session cookies of the AJAX request on Safari by exploiting its cookie header limitation (see Case-5 in Section 5.1.5 for the details). However, the server seemed to check whether session cookies under `https://www.google.com/search` are consistent with those under `https://www.google.com/`. Once receiving inconsistent session cookies from the AJAX request, it navigated the web interface to `https://www.google.com/search`, which still showed the attacker's profile name and icon.

Our final attack was to inject session cookies with domain scope "`www.google.com`" and path "/", so that for non-Safari browsers, the attacker could steal the victim's search history. Although this attack affects the web interface, causing to show the attacker's profile name and icon, it does not affect most other Google services. We also verified an invisible attack by spoofing the victim's profile name and icon.

5.1.2 Sub-session Hijacking Attacks

The two cases above show a common pattern: the attacker intends to limit the effective scope of injected session cookies as small as possible to reduce the visibility of his attack.

Essentially, web applications require one or more request-reply pairs with different URLs, which we view as different *sub-sessions*. In a normal case, when a user views a web page or performs a certain action through a series of pages, the corresponding sub-sessions carrying the same user authentication tokens are attributed to the user's account. However, when using cookies as authentication tokens, the cookie-URL matching rules and implementations often allow the attacker to selectively associate one or more sub-sessions to the attacker's account by cookie shadowing. That is why we call this type of attack sub-session hijacking attacks.

The impact of such attacks varies by the applications. In general, the attacker's strategy is to select a minimum set of sub-sessions that achieve his attack goals meanwhile keep the visibility of the attack as small as possible. However such attack could be made difficult by some implementation choices.

First, in general, a victim could notice a sub-session hijacking attack if she views abnormal changes of some visual elements on her screen. Typically such elements include username, email, a profile icon *etc.*, which we refer to as *ID-indicators*. If a website uses less URLs in one page or one certain functionality, and makes the important URLs related with the ID-indicators, the attacker is less likely to perform sub-session hijacking without being noticed. For example, in Case-2, the attacker has to hijack both of the URL that shows the search interface, and the AJAX request that performs the search. This limitation causes the expose of his profile name and icon, which may be noticed by the victim. However, if the attacker can only hijack an AJAX request which is not related to the interface, especially ID-indicators, the attack could be launched invisibly.

Second, explicit and session dependent verifiers could bind separate URLs together, so that the attacker needs to hijack more URLs. One example is using a session dependent nonce to counter CSRF attacks. Suppose the attacker wants to steal some sensitive information submitted by a form which is fetched from URL `GetForm` then submitted through URL `SubmitForm`. If the CSRF protection of the form is session dependent, *e.g.* a nonce associated with user session embedded in the form and verified when submitting, the attacker must hijack both `GetForm` and `SubmitForm` so that the CSRF verification does not fail. Otherwise he only needs to hijack `SubmitForm`.

It turns out that sub-session hijacking can be a powerful attack against today's websites. Because many web applications do not adopt mechanisms to bind sub-sessions together, and, for many operations, hijacking one sub-session is sufficient to cause serious consequences. Below we describe three common functionalities that are often vulnerable to sub-session hijacking, demonstrated with real-world cases.

5.1.3 Payment Account Stealing

Many websites require users to associate one or more payment accounts to pay their bills or online purchases. If the attacker hijacks the payment account submission form, he could get sensitive information, or even spend money using the victim's payment account.

Case-3: Credit card stealing on China UnionPay. China UnionPay, a government-owned financial corporation in China, has an online third-party payment service in which users can add their credit/debit cards. Although the process of adding a card involves four URLs as well as authentication via text message, all the URLs merely use one session cookie "uc_s_key" for authentication and the actual data submission is performed at one URL that is not related to any ID-indicator. We have verified that by shadowing the session cookie at the submission URL, the attacker can hijack China UnionPay's credit card association invisibly to obtain the victim's (obfuscated) credit card number and its spending history when the victim uses this interface in the future.

5.1.4 Online Deposit Hijacking

A common feature in many Chinese websites is the ability to deposit money from an online bank (or a third-party payment service like Alipay) into a website for future spending. We found this feature is particularly vulnerable to sub-session hijacking.

The process of online deposit usually includes six steps: 1) the user enters deposit amount; 2) the website generates an ID as a unique identifier of this transaction; 3) the website redirects the user to the selected online bank with the transaction ID; 4) the user authenticates and confirms to withdraw money from the online bank; 5) the online bank notifies the website with the transaction ID and redirects the user back to the website; 6) the website receives the notification from the online bank, and adds the corresponding amount on the user's account according to the transaction ID. The bank site only shows the transaction ID on its interface which is usually an unmeaningful string. If the attacker can hijack the step 2 to associate the transaction ID with his account without being noticed, the victim user is likely to finish all steps on the online bank because there is no abnormal visual indication. Once the victim does so, the money is deposited to the attacker's account.

Case-4: Deposit hijacking on JD.com. `JD.com` is a popular E-commerce website in China. In its implementation of the online deposit feature, the second step uses an AJAX request that is not related to any ID-indicator. We have verified that by shadowing `JD.com`'s session cookie "ceshi3.com" at the `AJAX` request, the attacker can hijack the online deposit invisibly, redirecting funds from the victim into the attacker's `jd.com` account.

5.1.5 Account Hijacking in SSO Association

Single Sign On (SSO) is a technique where an Identity Provider (IdP) provides authentication assertions for a logged-in user to relying parties (RP) for them to authenticate the user. SSO usually enables automatic login on the relying party, providing a better user experience and in some cases better security. This is a popular technique deployed by a number of large websites such as Google and Facebook as IdPs, and many other web sites as relying parties. Popular web protocols used for SSO implementation include OpenID [10] and OAuth [14].

Under certain conditions, SSO systems face a threat called *association violation* [31], in which a victim account on an RP is associated with an attacker's account on an IdP, so that the attacker gains control of the victim's account on the RP. This is likely to happen when 1) the victim is logged-in in the IdP as the attacker, 2) the RP has a feature for its users to associate with their accounts on the IdP, 3) the feature is implemented through redirections without further confirmation. The first condition can be mounted by cookie injection, and the websites satisfying the latter two conditions are not hard to find.

Case-5: Account hijacking against Google OAuth and BitBucket. BitBucket, a popular code hosting service, provides account association with Google by OAuth. If a user is already logged in with Google and has authorized BitBucket to access her Google profile through OAuth, the association is accomplished with two forth-and-back redirections with `https://accounts.google.com/o/oauth2/auth` without confirmation except a final message saying *"You've successfully connected your Google account"*.

Our goal is to hijack the Google OAuth URL to invisibly cause an association violation. There are 5 relevant session cookies: "SID/SSID/HSID" with domain scope ".google.com" and path "/", and "LSID/LSOSID" with domain scope ".accounts.google.com" and path "/". This is challenging because the server seemingly has deployed specific defense to counter cookie injection. First, `accounts.google.com` has enabled HSTS with `includeSubDomains`. Second, if we inject cookies with the same names, the server redirects us to a "CookieMismatch" warning page.

We successfully launch the attack on Safari by taking advantage of Safari's quirks. First, we exploit the HSTS implementation bug (Section 4.2) to inject the attacker's five session cookies with domain scope ".accounts.google.com" and path "/o/oauth2/". Recall that Safari ranks cookies by domain specificness then by path specificness, therefore the injected cookies shadows the legitimate ones. Then, we make use of Safari's 8 KB limitation on the cookie header (see Section 4.1) to get around the same name cookie detection. To achieve this, we inject a number of cookies with specific names and domain/path scopes, so that they are ranked between the injected session cookies and the legitimated session cookies. We control the length of these cookies to "overflow" the cookie list so that Safari truncates the legitimated session cookies when issuing requests to the OAuth URL. This allows us to bypassed all restrictions.

5.2 Cookies as References to Session Independent States

Session fixation is a well-known attack in which an attacker holds a session ID, then persuades a victim to authenticate with that ID so that he gains control of the victim's account [21]. Cookie injection can be used to exploit vulnerable websites that use cookies to store session IDs. Standard defenses, *e.g.* regenerating session ID after login, have been widely adopted.

However, we found that, although some websites have implemented defenses for typical session fixation attacks, they still have similar vulnerabilities for cookie injection. The root cause is that they associate important server-side states with long-term cookies. Moreover, they do not bind these states with user sessions.

The attacker can fixate such cookies by cookie injection (*e.g.*, through cookie overwriting) in order to access and manipulate the associated states. Interestingly, most of the vulnerable websites we found vulnerable are E-commerce websites.

Case-6: Shopping cart tracking/manipulation on popular E-commerce websites. We demonstrate this type of issues on 3 popular E-commerce websites: Apple, eBay, and JD.com. These websites allow unregistered visitors to add items in shopping carts. For better user experience, they never expire the shopping carts on the server side, and use long-term cookies on the browser side as references. We have verified that if the attacker fixates the corresponding cookies using cookie injection, he can track or manipulate shopping carts of the unregistered visitors.

We also found similar problems on Amazon, which are much more serious in terms of the real-world consequences, because they compromise security for registered users.[5]

Case-7: Browsing history and purchase tracking/hijacking on Amazon. Amazon's E-commerce websites use two long-term cookies "session-id" and "ubid-main" to associate with a user's browsing history and the ongoing purchase. Surprisingly, these important states are not associated with the user session (Not as its name suggests, "session-id" is not used for user authentication). Once the attacker fixates the two cookies, he can launch various attacks remotely.

The first exploitation is to track and manipulate the user's browsing history. Amazon keeps all previous viewed items in a user's browsing history. Upon fixating the two referencing cookies, the attacker can track what the victim have viewed on Amazon in real-time. He can also inject unwanted items into the browsing history, which affects the recommendation system.

Moreover, from what we observed, we infer that Amazon keeps a session independent data structure for an on-going purchase, which stores the user, the purchased items, the total amount, the delivery address, and other payment information. The structure is likely created by clicking the "proceed to checkout" button, and released after clicking of the "place your order" button. This structure is associated with the same two cookies referencing the browsing history. By fixating the two cookies and consequently gaining access of the data structure, the attacker has various ways to manipulate the victim's purchase remotely. Below we describe two exploitations:

- **Tracking of all purchases.** First, the attacker can track all purchases of the victim. To do so, he first creates an on-going purchase, of which the internal data structure is also shared with the victim. Later, when the victim makes a purchase, the information is updated to the shared data structure, consequently retrieved by the attacker. On Amazon China, the attacker can see all information of the victim's purchase including items, amount, the victim's name, delivery address, and cellphone number. On Amazon U.S., the delivery address and cellphone number are not visible by the attacker.

- **Potential hijacking of purchases.** When detecting an ongoing purchase by the victim, the attacker can change the delivery address so that the purchase is paid by the victim but sent to the attacker. If the victim confirms the hijacked purchase, she cannot even see where the purchase is hijacked to in her order history, because Amazon only shows "Gifting address". The attacker can even manipulate the purchase in such a way that it is paid by the victim, delivered to the attacker, and only recorded in the attacker's order history. The only limitation of the attack is that if the delivery address is new to the payment option, Amazon requires the victim to confirm the card number, however the interface is arguably not alarming. On Amazon China, this limitation does not apply if the victim chooses to pay with a third-party service like Alipay.

5.3 Cookies reflected into HTML

Another common practice is to store auxiliary variables like preferred language or username as cookies, and reflect these cookies into HTML or script snippets. If not implemented carefully, this practice could make websites vulnerable to various attacks in face of cookie injection.

5.3.1 XSS via Cookie Injection

A direct threat is XSS: if reflected cookies are not sanitized sufficiently, the attacker can embed malicious scripts in reflected cookies to launch XSS attacks through cookie injection.

Case-8: XSS via cookie injection on China Construction Bank, Amazon Cloud Drive, eBay and others. We found a number of websites do not validate reflected cookies sufficiently. Using cookie injection, we successfully mounted XSS against China Construction Bank, Amazon Cloud Drive, eBay and several other websites.

Case-9: Insufficient cookie validation on Bank of America. Among the XSS vulnerabilities we found, the one on the Bank of America website is fairly unique. We

[5] However, we note that many E-commerce sites, including Amazon, use mixed content, and thus are also vulnerable to attackers injecting scripts into the insecure domain that remain in the browser cache.

found that one cookie with path "/" on Bank of America's website could be exploited to inject XSS. At first, our naïve exploitation by overwriting the cookie with a XSS payload failed. The limitation was that the website performed a strict validation on the cookie at the login URL. Only if the cookie was absent would the website set a clean value on the cookie from the response of the login URL, then used it in subsequent requests without validation. Our naïve exploitation was prevented by the strict validation at the login URL.

We found a technique to bypass the limitation, so that the XSS payload can be buried into the victim's browser and triggered when next time the victim logs in by injecting multiple cookies. We injected two cookies. The first one had the same 3-tuple identifier as the legitimate one, but with an expired time to ensure the legitimate cookie was discarded and absent at the login URL. The second injected cookie contained the XSS payload and had a different cookie path "/myaccount" that matched with the first URL after login. Although the server set a clean cookie in the response of the login URL, the specific path of the second injected cookie not only avoid being overwritten by the clean cookie, but also shadowed the clean cookie in subsequent requests, triggering a successful XSS attack.

This case implies a possible misconception that performing a complete cookie validation on one "entry point" is sufficient. In fact, because of the asymmetry between cookie read and write operations, every different URL might bring different and unexpected cookie values no matter how server set cookies in previous responses. Developers must treat every request as a new entry point and carefully validate all associated cookies.

5.3.2 BREACH Attacks through Cookie Injection

In 2002, Kelsey observed that when combining encryption with some compression algorithms, the size of compressed data can be used as a side channel, potentially causing plaintext leakage under certain conditions [20]. Rizzo and Duong found a real-world case in 2012, named as CRIME attack, in which an active network attacker initiates encrypted HTTP requests from a victim browser with different URLs as partially-chosen plaintexts, then infer embedded secrets like session cookies by observing the sizes of the compressed and encrypted requests [29]. Rizzo and Duong also mentioned that a similar attack could also be mounted to infer secrets in encrypted HTTP responses. This was explored and demonstrated by Gluck *et al.*, named as the BREACH attack [12].

BREACH requires the attacker to be able to 1) inject a partially-chosen plaintext into the HTTP response of one webpage, and 2) measure the size of the compressed then encrypted response. An active network attacker satisfies the second condition. If a webpage contains a reflected cookie, the attacker can abuse it with cookie injection as the first vector to launch the BREACH attack to infer secrets in this webpage.

Case-10: BREACH attacks on phpMyAdmin and MediaWiki. We found phpMyAdmin, a popular open source web application for remote database management, reflects a cookie for language preference after sanitization in error page if its value is invalid. The BREACH attack using this cookie can reliably infer the CSRF token in the error page, enabling further CSRF attacks. Similarly, MediaWiki reflects a cookie into its login form, also allowing the BREACH attack to infer the CSRF token in the login page.

5.4 Summary

These exploitations show that cookie injection enables undesired and complicated interactions among cookie implementations, web applications, and various known threats. It is clear that our empirical assessment only touches a part of the whole problem space. Nevertheless, we believe these cases demonstrate that the security implications of cookie's lack of integrity are not well and widely understood by the community, and current cookie practices have widespread problems when cookie injection is taken into consideration.

Report and Response. We have reported all vulnerabilities to the affected websites. Some have acknowledged (*e.g.*, Amazon), and some (*e.g.*, Bank of America) have fixed the issues.

6 Possible Defenses

Some existing techniques can help mitigate this threat, including full HSTS, public suffix list, defensive cookie practices, and anomaly detection.

Full HSTS and Public Suffix List. We strongly recommend that websites deploy full HSTS to prevent cookie injection from active network attackers, as this provides complete protection once a site is pinned by a user visit. The community should also make the effort to raise the awareness of cookie injection attacks, and clarify the different levels of security provided by HTTPS, HSTS, and full HSTS. For websites that host shared domains, the best practice is to use separate domains and register them on the public suffix list. Efforts also should be made to increase the awareness of cookie injection from shared domains and the public suffix list.

Defensive Cookie Practices. For websites that cannot enable full HSTS, and have concerns about cookie injection from related domains, defensive cookie practices may mitigate certain cookie injection threats. For example, frequently invalidating session cookies could reduce the risk of sub-session hijacking. Instead of using cookies, Websites can also use new features in HTML5 like `localStorage` and `sessionStorage` to facilitate browser-side state management, which does not have cookie's integrity deficiencies, although these mechanisms are less convenient for cross-protocol and cross-domain state sharing.

Anomaly Detection. Websites should consider detecting same name cookies in the cookie header, as we discussed in the `accounts.google.com` case. This is reasonable because same name cookies should not be considered a legitimate use according to both the specification and the inconsistent implementations. This detection would protect non-Safari users from attacks using cookie shadowing.

6.1 Proposed Browser Enhancements

We propose several browser-side enhancements to mitigate cookie injection attacks. Our proposals do not require any server-side change, so they would benefit many legacy websites.

6.1.1 Mitigating Active Network Attackers

Currently, Chrome, Firefox and Safari, but not Internet Explorer, have deployed the HSTS support. We believe that if all major browsers could deploy it, websites with full HSTS would be capable of defending against active network attackers in most cases. However, deploying full HSTS needs all subdomains to support HTTPS with valid certificates. There are a number of practical hurdles for websites to satisfy such a strict requirement. For example, Google cannot enable full HSTS, because it is required to support non-HTTPS access for mandatory adult-content filtering at school and some other locations [13]. Kranch and Bonneau also reported the current incapability of Facebook and Twitter to deploy full HSTS [22]. Hence, we believe full HSTS is not likely to be adopted widely in the near future.

To protect a site which cannot deploy full HSTS, a browser must not allow an HTTP connection to replace or shadow secure cookies, effectively adding an HSTS-like pin for any secure cookie. We propose to modify the semantics of the existing cookie store by adding a "do not send" flag and changing the cookie store behavior with the following semantics. We believe these semantic change should provide protections while minimizing the disruption to existing sites:

1. A browser MUST NOT accept a cookie presented in an HTTP response with the `secure` flag set, nor overwrite an unexpired `secure` cookie, except the case in 5.
2. Cookies with the `secure` flag MUST be given higher priority over non-secure cookies.
3. A browser MUST only send the highest priority cookie for any cookie name.
4. In removing cookies due to a too-full cookie store, the browser MUST NOT remove a `secure` cookie when there are non-secure cookies that can be removed.
5. The browser MUST allow an HTTP connection to clear a `secure` cookie by setting an already-expired expiration date, but the browser MUST NOT remove the cookie from the store. Instead, the browser MUST set the "do not send" flag and maintain the original expiration date.
6. The browser MUST NOT send a cookie with the "do not send" flag, nor send any non-secure cookie with the same name.

The first rule prevents an active network attacker from injecting or replacing `secure` cookies. The second and third rules combined prevent an active network attacker from shadowing a secure cookie. The fourth rule prevents an attacker from flooding the cookie store to evict secure cookies. The fifth and sixth rules are subtle but necessary: mixed-content sites might have a "logout" button in HTTP which clears `secure` session cookies. We wish to enable this functionality without allowing attackers to remove and replace a `secure` cookie.

Taken together, our proposals should add HSTS-like pinning to secure cookies within the existing cookie store. If a cookie was set with `secure` flag, an active network attacker can only delete it, which largely mitigates cookie injection attacks [6].

Compatibility. We implemented the first three rules as a Chrome extension[7], and used the extension to manually examined the Alexa top 40 websites. We found one broken case: the signing out operation on `http://www.bing.com/` results in a request-reply with `http://login.live.com/logout.srf` which expires several `secure` session cookies under its SSO IdP domain `live.com`. Allowing HTTP to clear secure cookies should improve compatibility with such signing-out practice.

We also crawled the Alexa top 100,000 domains with both HTTP and HTTPS. In total, 48,039 domains responded with cookies. 152 (0.32%) domains returned `secure` cookies over HTTP; 570 (1.19%) domains responded with cookies that have same name yet different

[6] The non-conforming cookie name behaviors of PHP, ASP, and ASP.NET described in Section 4.1 still expose some possibilities for cookie shadowing. We suggest vendors to fix these incorrect implementations.

[7] `https://github.com/seccookie/ExtSecureCookie`

domains and/or paths. These numbers suggest `secure` cookies over HTTP (incompatible with the first rule) and same name cookies (related to the second and third rules) are rare in real-world websites. We manually examined 10 domains in each case with our extension and did not observe evidence of broken behaviors.

While the results from our compatibility testing are promising, we acknowledge they are preliminary. First, we may have missed subtle incompatibility issues in our manual testing. Second, some incompatibility behaviors may only occur with logged-in sessions and/or specific paths, which our testing may have failed to uncover. We hope our limited experiments will motivate browser vendors to conduct large-scale in-depth compatibility evaluation.

6.1.2 Mitigating Web attackers

A domain can set cookies with a more specific domain scope (*e.g.* host-only) to prevent cookie stealing by XSS from sibling domains. But this currently has no effect on cookie injection since injected cookies with shared domain scopes yet longer paths are effective for cookie shadowing, and longer paths are available in most cases if an adversary is in control of a related domain. Combining the second rule of the above proposals, we suggest:

7. When issuing a request, the browser MUST rank the cookie list by a) presence of the `secure` flag, and b) specificity of the domain scope.

Together with the third rule presented above, this should enable developers to prevent cookies from being overwritten or shadowed by using specific domain scope (together with the `secure` flag when using HTTPS). We have also implemented this policy in the same Chrome extension mentioned above.

7 Related Work

Comparison to Previous Work. We are aware of several research papers that are directly related to cookie's weak SOP and integrity problem [4, 30, 24, 23, 5, 2, 32], and some other papers that are comparable to our work [17, 22]. Among the directly related research, Barth's [2] and Zalewski's work [32] focused on explaining the cause of the cookie integrity problem. Most previous research only briefly touched cookie integrity as a relevant subproblem rather than main topic [4, 30, 24, 23]. Bortz *et al.* 's research is close to ours. Especially, they introduced the notion of a related domain attacker which we use throughout this paper. However, their work is limited to high-level discussion [5]. In summary, previous research has discussed the problem of cookie's lack of integrity, its root cause, and its security implications. However, prior understanding of the subtlety, prevalence, and severity of this problem in the real world is limited. We take a much closer look at the problem space, provide a number of new empirical assessments which we believe will help the community understand the problem more deeply and know the status quo better. Specifically, we conduct a detailed measurement of full HSTS adoption and reveal the threat to CDN customers. Prior to our work, Kranch and Bonneau recently studied full HSTS deployment practice but within a different context [22]. The cookie related problems revealed in our assessment of browser and server libraries are largely unknown, except a few fragmented knowledge from Lundeen's [23] and Lundeen *et al.* 's work [24]. The attack cases we present also supplement previous discussion on potential exploitations in both breadth and depth. Our close-up study also leads us to find promising cookie isolation enhancements that only require browser-side adoption. In contrast, the previous proposed defenses need both browser- and server-side changes [4, 5].

Broadly, our work can be viewed as an in-depth case study of inconsistent access control policies in web. Jackson and Barth's [17] and Singh *et al.* 's work [30] explored this general problem, and each provided various instances. One example illustrated by Jackson and Barth is the ability of JavaScript to read all cookies with matching domain scopes regardless of their paths [17]. This behavior has now been noted explicitly in the current specification [2].

Security Related Cookie Measurement. Zhou and Evans studied the rare deployment of the `HTTPOnly` cookies at the time [33]. They believed that the requirement of both client and server changes played an important hurdle in its adoption. Kranch and Bonneau found many websites deploy HSTS yet do not marked their cookies with the `secure` flag, which are vulnerable to cookie theft in certain conditions [22]. These two measurements were concerned with cookie's confidentiality, while our work looks at the other property, *i.e.* cookie's integrity. Singh *et al.* measured the real-world usages of `secure` cookies (0.07%, 62 out of 89,222 sites) over HTTP and same name cookies (they called duplicate cookies) (5.48%, 4,893 out of 89,222 sites) [30]. Our assessment obtains similar results.

8 Conclusions

Cookies lack integrity. Although long known in community lore, the community has under-appreciated the implications. We have attempted to systematically evaluate the implications of cookie integrity, including evaluating weaknesses and evaluation artifacts in both browser and server libraries, building real-world attacks against major sites including Google and Bank of America, including subtle account-hijack attacks and XSS attacks buried in injected cookies, and developing an alternate browser cookie policy that mitigates the threat from network-level attackers. We expect our work to raise the awareness of the problem, and to provide a context for further discussion among researchers, developers and vendors.

Acknowledgements

We would like to thank our shepherd Hovav Shacham, and the anonymous reviewers for their insightful comments. We are grateful to Vern Paxson, Frank Li and David Fifield for valuable discussion, and Jianjun Chen for help of some exploitations. We also thank Chris Evans, Joel Weinberger, Chris Palmer, and Nick Sullivan for valuable feedback. This work is partially supported by the National Natural Science Foundation of China (Grant No. 61472215), Tsinghua National Laboratory for Information Science and Technology (TNList) Academic Exchange Foundation, and the Natinoal Science Foundation (CNS-1213157 and CNS-1237265).

References

[1] Edit This Cookie. `http://www.editthiscookie.com/`. [accessed Feb-2015].

[2] Barth, A. HTTP State Management Mechanism. *IETF RFC 6265* (2011).

[3] Barth, A. The Web Origin Concept. *IETF RFC 6454* (2011).

[4] Barth, A., Jackson, C., and Mitchell, J. C. Robust Defenses for Cross-Site Request Forgery. In *Proceedings of the 15th CCS* (2008), ACM, pp. 75–88.

[5] Bortz, A., Barth, A., and Czeskis, A. Origin Cookies: Session Integrity for Web Applications. *Web 2.0 Security and Privacy (W2SP)* (2011).

[6] Chen, S., Mao, Z., Wang, Y.-M., and Zhang, M. Pretty-Bad-Proxy: An Overlooked Adversary in Browsers' HTTPS Deployments. In *Proceedings of the 30th IEEE S&P (Oakland)* (2009), IEEE, pp. 347–359.

[7] Evans, C. Cookie Forcing. `http://scarybeastsecurity.blogspot.com/2008/11/cookie-forcing.html`, 2008. [accessed Feb-2015].

[8] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. Hypertext Transfer Protocol–HTTP/1.1. *IETF RFC 2616* (1999).

[9] Fielding, R., and Reschke, J. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. *IETF RFC 7230* (2014).

[10] Foundation, O. OpenID Authentication 2.0 - Final. `http://openid.net/specs/openid-authentication-2_0.html`. [accessed Feb-2015].

[11] GitHub. Yummy Cookies across Domains. `https://github.com/blog/1466-yummy-cookies-across-domains`, 2013. [accessed Feb-2015].

[12] Gluck, Y., Harris, N., and Prado, A. BREACH: Reviving the CRIME Attack. `http://breachattack.com/resources/BREACH%20-%20SSL,%20gone%20in%2030%20seconds.pdf`, 2013. [accessed Feb-2015].

[13] GoogleSupport. Block Adult Content at Your School. `https://support.google.com/websearch/answer/186669?hl=en`. [accessed Feb-2015].

[14] Hardt, D. The OAuth 2.0 Authorization Framework. *IETF RFC 6749* (2012).

[15] Hodges, J., Jackson, C., and Barth, A. Http Strict Transport Security (HSTS). *IETF RFC 6797* (2012).

[16] IEBlog. Project Spartan and the Windows 10 January Preview Build. `http://blogs.msdn.com/b/ie/archive/2015/01/22/project-spartan-and-the-windows-10-january-preview-build.aspx`. [accessed Feb-2015].

[17] Jackson, C., and Barth, A. Beware of Finer-Grained Origins. *Proceedings of 2th W2SP* (2008).

[18] Jackson, C., and Barth, A. ForceHTTPS: Protecting High-Security Web Sites from Network Attacks. In *Proceedings of the 17th WWW* (2008), ACM, pp. 525–534.

[19] Johnston, P., and Moore, R. Multiple Browser Cookie Injection Vulnerabilities. `http://www.westpoint.ltd.uk/advisories/wp-04-0001.txt`, 2004. [accessed Feb-2015].

[20] Kelsey, J. Compression and Information Leakage of Plaintext. In *Fast Software Encryption* (2002), Springer, pp. 263–276.

[21] Kolšek, M. Session Fixation Vulnerability in Web-based Applications. `http://www.acros.si/papers/session_fixation.pdf`, 2002. [accessed Feb-2015].

[22] Kranch, M., and Bonneau, J. Upgrading HTTPS in mid-air: An empirical study of strict transport security and key pinning. In *Proceedings of the 22th NDSS* (2015).

[23] Lundeen, R. The Deputies Are still Confused. *Blackhat EU* (2013).

[24] Lundeen, R., Ou, J., and Rhodes, T. New Ways Im Going to Hack Your Web App. *Blackhat AD* (2011).

[25] Mozzila. Public Suffix List. `https://publicsuffix.org/`. [accessed Feb-2015].

[26] Nginx. Module ngx_http_core_module. `http://nginx.org/en/docs/http/ngx_http_core_module.html#large_client_header_buffers`. [accessed Jun-2015].

[27] Paxson, V. Bro: A System for Detecting Network Intruders in Real-Time. *Computer networks 31*, 23 (1999), 2435–2463.

[28] Projects, T. C. HTTP Strict Transport Security. `http://www.chromium.org/hsts`. [accessed Feb-2015].

[29] Rizzo, J., and Duong, T. The CRIME Attack. In *EKOparty Security Conference* (2012), vol. 2012.

[30] Singh, K., Moshchuk, A., Wang, H. J., and Lee, W. On the Incoherencies in Web Browser Access Control Policies. In *Proceedings of the 31th IEEE S&P (Oakland)* (2010), IEEE, pp. 463–478.

[31] Wang, R., Zhou, Y., Chen, S., Qadeer, S., Evans, D., and Gurevich, Y. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization. In *USENIX Security* (2013), pp. 399–314.

[32] Zalewski, M. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2012.

[33] Zhou, Y., and Evans, D. Why Arent HTTP-only Cookies More Widely Deployed. *Proceedings of 4th W2SP 2* (2010).

The Unexpected Dangers of Dynamic JavaScript

Sebastian Lekies
Ruhr-University Bochum
paper@sebastian-lekies.de

Ben Stock
FAU Erlangen-Nuremberg
ben.stock@fau.de

Martin Wentzel
SAP SE
martin.wentzel@sap.com

Martin Johns
SAP SE
martin.johns@sap.com

Abstract

Modern Web sites frequently generate JavaScript on-the-fly via server-side scripting, incorporating personalized user data in the process. In general, cross-domain access to such sensitive resources is prevented by the Same-Origin Policy. The inclusion of remote scripts via the HTML script tag, however, is exempt from this policy. This exemption allows an adversary to import and execute dynamically generated scripts while a user visits an attacker-controlled Web site. By observing the execution behavior and the side effects the inclusion of the dynamic script causes, the attacker is able to leak private user data leading to severe consequences ranging from privacy violations up to full compromise of user accounts.

Although this issues has been known for several years under the term Cross-Site Script Inclusion, it has not been analyzed in-depth on the Web. Therefore, to systematically investigate the issue, we conduct a study on its prevalence in a set of 150 top-ranked domains. We observe that a third of the surveyed sites utilize dynamic JavaScript. After evaluating the effectiveness of the deployed countermeasures, we show that more than 80% of the sites are susceptible to attacks via remote script inclusion. Given the results of our study, we provide a secure and functionally equivalent alternative to the use of dynamic scripts.

1 Introduction

Since its beginning in the early nineties, the Web evolved from a mechanism to publish and link static documents into a sophisticated platform for distributed Web applications. This rapid transformation was driven by two technical cornerstones:

1. Server-side generation of code: For one, on the server side, static HTML content was quickly replaced by scripting to dynamically compose the Web server's HTTP responses and the contained HTML/JavaScript on the fly. In turn, this enabled the transformation of the Web's initial document-centric nature into the versatile platform that we know today.

2. Browser-driven Web front-ends: Furthermore, the Web browser has proven to be a highly capable container for server-provided user interfaces and application logic. Thanks to the flexible nature of the underlying HTML model and the power of client-side scripting via JavaScript, the server can push arbitrary user interfaces to the browser that rival their counterparts of desktop application. In addition, and unlike monolithic desktop applications, however, browser-based UIs enable easy incorporation of content from multiple parties using HTML's inherent hypertext capabilities.

Based on this foundation, the recent years have shown an ongoing shift from Web applications that host the majority of their application logic on the server side towards rich client-side applications, which use JavaScript to realize a significant portion of their functionality within the user's browser.

With the increase of the functionality implemented on the client side, the necessity for the JavaScript code to gain access to additional user data rises natrually. In this paper, we explore a specific technique that is frequently used to pull such data from the server to the client-side: Dynamic JavaScript generation.

Similar to HTML, which is often generated dynamically, JavaScript may also be composed on the fly through server-side code. In this composition process, user-specific data is often included in the resulting script code, e.g., within the value of a variable. After delivering the script to the browser, this data is immediately available to the client-side logic for further processing and presentation. This practice is potentially dangerous as the inclusion of script files is exempt from the Same-Origin Policy [23]. Therefore, an attacker-controlled Web page is able to import such a dynamically generated script and observe the side effects of the execution, since all included scripts share the global object of the embedding web document. Thus, if the script contains user-specific data, this data might be accessible to other attacker-controlled JavaScript. Although this attack, dubbed *Cross-Site Script Inclusion* (XSSI), has been mentioned within the literature [31], the prevalence of flaws which allow for this attack vector has not been studied on real-world Web sites.

In this paper we therefore present the first, systematic analysis of this vulnerability class and provide empirical evidence on its severeness. First, we outline the general attack patterns and vectors that can be used to conduct such an attack. Furthermore, we present the results of an empirical study on several high-profile domains, showing how these domains incorporate dynamic scripts into their applications. Thereby, we find evidence that many of these scripts are not or only inadequately protected against XSSI attacks. We demonstrate the sever consequences of these data leaks by reporting on real-world exploitation scenarios ranging from de-anonymization, to targeted phishing attacks up to complete compromise of a victim's account.

To summarize, we make the following contributions:

- We elaborate on different ways an attacker is capable of leaking sensitive data via dynamically generated scripts, enabled by the object scoping and dynamic nature of JavaScript.

- We report on the results of an empirical study on several high-ranked domains to investigate the prevalence of dynamic scripts.

- Using the data collected during our empirical study, we show that many dynamic scripts are not properly protected against XSSI attacks. To demonstrate the severity of the outlined vulnerabilities, we present different exploitation scenarios ranging from de-anonymization to complete hijacking of a victim's account.

- Based on the observed purposes of the dynamic scripts encountered in our study, we discuss secure ways of utilizing such data without the use of dynamically generated scripts.

The remainder of the paper is structured as follows: In Section 2, we explain the technical foundations needed for rest of the paper. Section 3 then covers the general attack patterns and techniques to exploit cross-domain data leakage vulnerabilities. In Section 4, we report on the results of our empirical study and analyze the underlying purposes of dynamic scripts. Furthermore, in Section 5, we provide a scheme that is functionally equivalent, but is not prone to the attacks described in this paper. Section 6 covers related work, Section 7 gives an outlook and Section 8 concludes the paper.

2 Technical Background

In this section, we cover the technical background relevant for this work.

2.1 The Same-Origin Policy

The *Same-Origin Policy* (SOP) is the principal security policy in Web browsers. The SOP strongly separates mutually distrusting Web content within the Web browser through origin-based compartmentalization [23]. More precisely, the SOP allows a given JavaScript access only to resources that have the same *origin*. The origin is defined as the triple consisting of scheme, host, and port of the involved resources. Thus, for instance, a script executed under the origin of `attacker.org` is not able to access a user's personal information rendered under `webmail.com`.

While JavaScript execution is subject to the SOP, the same does not hold true for cross-domain inclusion of Web content using HTML tags. Following the initial hypertext vision of the WWW HTML-tags, such as `image`, may reference resources from foreign origins and include them into the current Web document.

Using this mechanism, the HTML `script` tag can point to external script resources, using the tag's `src` attribute. When the browser encounters such a tag, it issues a request to the foreign domain to retrieve the referenced script. Important to note in this instance is the fact that the request also carries authentication credentials in the form of cookies which the browser might have stored for the remote host. When the response arrives, the script code inherits the origin of the *including* document and is executed in the context of the hosting page. This mechanism is used widely in the Web, for instance to consume third party JavaScript services, such as traffic analysis or advertisement reselling [24].

2.2 JavaScript Language Features

In the following, we cover the most important JavaScript concepts necessary for the rest of the paper.

Scoping In JavaScript, a scope is "a lexical environment in which a function object is executed" [6]. From a developer's point of view, a scope is the region in which an identifier is defined. While C++ or Java make use of block scoping, JavaScript utilizes so-called function scoping. This means that the JavaScript engine creates a new scope for each new function it encounters. As a consequence, an identifier that is locally defined in such a function is associated with the corresponding scope. Only code that is defined within the same function is thus able to access such a variable residing in the *local scope*, whereas global variables are associated with the *global scope*.

Listing 1 shows an example for local and global variables. A local variable in JavaScript can be created by utilizing the var keyword. All variables defined outside of a function are associated with the global scope, whereas code within a function can define variables in the global scope by either omitting the var keyword or explicitly assigning to window.varname.

Listing 1 Example for global and local variables

```
// A global variable
var globalVariable1 = 5;

function globalFunction(){
  // A local variable
  var localVariable = 2;

  // Another global variable
  globalVariable2 = 3;

  // Yet another global variable
  window.globalVariable3 = 4;
}
```

The Prototype Chain As opposed to classical programming languages such as C++ or Java, JavaScript is a prototype-based language. This means that JavaScript's inheritance is not based on classes but directly on other objects, whereas "each object has a link to another object called its prototype" [21]. On creation of an object, it either automatically inherits from Object.prototype or if a prototype object is explicitly provided, the prototype property will point to this object. On access to an object's property, the JavaScript runtime checks whether the current object contains a so-called *own property* with the corresponding name.

If no such property exists, the object's prototype is queried for the same property and if lookup fails again, the process is recursively repeated for the object's prototypes. Hence, objects in JavaScript form a so-called *prototype chain*. Listing 2 gives a commented example for this behavior.

Listing 2 The prototype chain

```
var object1 = {a: 1};
// object1 ---> Object.prototype ---> null

var object2 = Object.create(object1);
// object2 ---> object1
//         ---> Object.prototype ---> null
console.log(object2.a); // 1 (inherited)
```

3 Cross-Domain Data Leakages

In this section, we show how an adversary can utilize an external JavaScript file, which is dynamically generated at runtime, to leak security sensitive data. After first covering the different types of these dynamic scripts, we elaborate on the attacker model and then demonstrate different attack vectors that can be leveraged to leak sensitive data from such a script.

3.1 Dynamic Scripts

As discussed in Section 2.1, Web pages can utilize script-tags to import further JavaScript resources. For the remainder of this paper, we define the term *dynamic script* to describe such a JavaScript resource in case it is generated by the Web server on the fly via server-side code.

As opposed to static scripts, the contents of dynamic scripts may vary depending on factors such as input parameters or session state. In the context of this paper, the latter type is of special interest: If a dynamic JavaScript is generated within a user's authenticated Web session, the contents of this script may contain privacy or security sensitive data that is bound to the user's session data. Thus, an execution of the script can potentially lead to side effects which leak information about this data.

3.2 Attack Method

HTML script tags are not subject to the Same-Origin Policy (see Section 2.1). Hence, script resources can be embedded into cross-domain Web pages. Although such cross-domain Web pages cannot access the source code of the script directly, this inclusion process causes the browser to load and execute the script code in the context of the cross-domain Web page, allowing the importing page to observe the script's behavior. If a dynamic script exposes side effects dependent on sensitive data in the script code, the execution of such a script may leak the secret data.

Figure 1 depicts an example attack. A user is authenticated to his mail provider at webmail.com, thus his browser automatically attaches the corresponding session cookies to all requests targeting webmail.com, which utilizes session-state dependent dynamic scripts. Thus, whenever a user is logged in, the script at webmail.com/script.js creates a global variable containing the current user's email address. In the same browser, the user now navigates to an attacker-controlled Web site at attacker.org. The attacker includes the dynamic script in his own Web page and subsequently, the browser requests the script with attached authentication cookies. Although the script originates

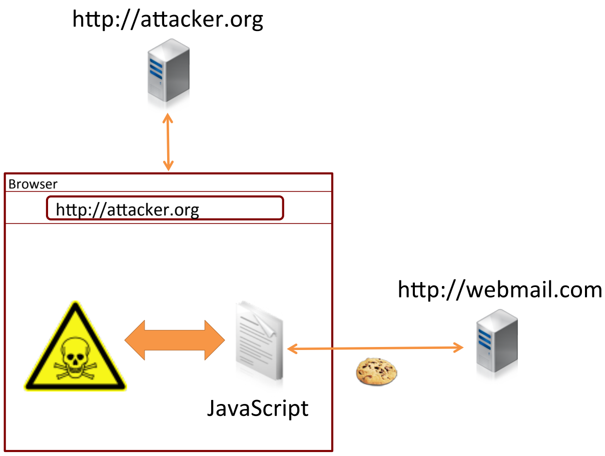


Figure 1: Attacker Model

from `webmail.com`, it is now executed in the context of `attacker.org`, creating the global variable with the user's email in the corresponding context. The global variable is now accessible to any other script executed by `attacker.org`. Hence, the attacker can simply access the value of this global variable, effectively leaking the user's email address.

3.3 Attack Vectors

As previously explained, an attacker is able to leak sensitive user data by including a script from a different domain and observing the results of the execution. In this section we outline different situations in which sensitive data is accessible to an attacker after the included script has been executed.

3.3.1 Global Variables

As noted in the previous section, global variables created by a dynamic script can be accessed by any other script executed on the same Web document. Hence, whenever sensitive user data is assigned to such a global variable inside a script, an attacker can gain access to the corresponding data. In order to do so, he simply includes the script and waits for the global variable to be created. As soon as the value assignment has occurred, the attacker's code can read the sensitive data and leak it back to his backend.

3.3.2 Redefinition of Global APIs

Due to JavaScript's dynamic nature, (almost) any function can be overwritten by an attacker, including a number of globally available APIs. If a dynamic script passes a security-sensitive value to such a function, the attacker may overwrite it beforehand and hence retrieve the secret value. Listing 3 demonstrates how an attacker can, for example, change the behavior of the global function `JSON.stringify`. In order to conduct an attack, the attacker overrides the function first and then includes a dynamic script which passes a sensitive data value to the function. When the user visits the attacker's Web site, his browser retrieves and executes the dynamic script. Rather than invoking the native `JSON.stringify` function, the contained code invokes the attacker-controlled function. In this case, instead of serializing the object, the function sends the user's data back to the attacker's server.

Listing 3 Passing a variable to a global function

```
// Attacker's script overwriting a global function
JSON.stringify = function(data){
  sendToAttackerBackend(data);
}

-------------------------------------------------------
//Within the dynamic script
function myFunction() {
  var myVar = { secret: "secret value"};

  // Calling a predefined global function
  return JSON.stringify(myVar);
}
```

3.3.3 Prototype Tampering

As outlined in the previous section, variables are available in the scope in which they were defined unless the `var` keyword is omitted. Listing 4 shows an example of code making use of this paradigm. The function allocates an array with three secret values using the `var` keyword and therefore, as it seems, protects the array from access by outside code. As discussed in Section 2.2, JavaScript is a prototype-based language. Hence, when requesting a property of an object, the JavaScript interpreter walks up the prototype chain until it finds a matching property. In our example shown in Listing 4, the function `slice` is called on the array named `arr`. By default, an array object does not provide the slice function itself. Therefore, the call is made to the function in the array's prototype, which points to the object `Array.prototype`. In a scenario where the script is included without any malicious intent, the programmer may assume that the call will eventually trigger invocation of the `slice` method for arrays.

This behavior may, however, be changed by an attacker. Listing 5 depicts a small snippet of code that is provided by the attacker. Similar to what we discussed earlier with respect to overwriting global functions, the snippet overwrites the `slice` method in the array's pro-

Listing 4 Variable protected in a closure

```
(function(){
  var arr = ["secret1","secret2","secret3"];
  // intents to slice out first entry
  var x = arr.slice(1);
  ...
})();
```

totype. Since by default all arrays in JavaScript share the same prototype, the call to `slice` in Listing 4 is passed to the attacker-provided function. Since the function is called on the `arr` object, the attacker can use the `this` keyword to gain a reference to the object. Therefore, rather than exhibiting the intended behavior of slicing out a part of the array, the attacker's code now sends the otherwise properly protected information back to the attacker. This attack works for any object that has a globally accessible prototype, i.e., it is feasible on any built-in objects such as Strings or Functions.

Listing 5 Leaking data via the this keyword

```
Array.prototype.slice = function(){
  //leaks ["secret1","secret2","secret3"]
  sendToAttackerBackend(this);
};
```

3.4 Distinction towards CSRF

On first view, the described attack method is related to Cross-site Request Forgery (CSRF) [1], as it follows a similar attack pattern.

In fact, leaking sensitive information via cross-domain script includes belongs to a larger class of Web attacks which function via creating authenticated requests in the context of an authenticated Web user (including CSRF, ClickJacking [12] and reflected Cross-site Scripting [2]).

However, the goal and consequences of the attack differ significantly from other attack variants: CSRF is an attack in which an attacker generates requests to cause state-changing actions in the name of the user. Thereby the attacker is by no means able to read content from a response to a CSRF request. To prevent CSRF developers are advised to conduct state-changing actions only via HTTP POST requests and to protect all these post requests with CSRF tokens.

As opposed to this, dynamic scripts are neither designed to conduct state-changing actions on the server-side nor are these scripts ever fetched via POST requests. Those scripts are stateless and are fetched via GET requests through script tags and, hence, are not classified as a critical endpoint in the context of CSRF, i.e., not contained in the application's CSRF protection surface.

4 Empirical Study

In this section we report on the results of an empirical study designed to gain insights into the prevalence and exploitability of data leakages due to the use of dynamic script generation on the server side. We first discuss the methodology of our study and report on the general prevalence of dynamically generated JavaScript files in the wild. Based on the gathered data, we analyze the underlying purposes of these scripts, discuss the types of security-sensitive data contained in the scripts and highlight who these can be leaked, allowing us specific exploits against a number of sites. We end the section with a discussion of situations in which we could not exploit a dynamic script due to the use of adequate protection measures.

4.1 Methodology

In the following, we cover our research questions, explain our detection methodology and describe our data set.

4.1.1 Research Questions

This study provides an in-depth analysis of dynamic script includes. Before diving into the security aspects of these scripts, we aim at collecting data on this technique in general. Hence, we are first interested in the general prevalence of dynamically generate scripts. More specifically, the goal is to find out how common dynamic script generation is in today's Web and how often these dynamic scripts are dependent on a user's session state. The study sheds light on the purpose of these scripts and the contained data. Finally, we investigate the security aspects by investigating the exploitability and discussing potential countermeasures.

4.1.2 Detecting State-dependent Scripts

As a basis for our empirical study, we needed a means to easily detect state-dependent dynamic scripts. Therefore, we implemented a Chrome browser extension that fulfills two separate tasks:

1. **Collecting scripts:** The first step towards analyzing Web pages for dynamic scripts is the collection of all external script resources included by the Web page. For this purpose, we created a browser extension that collects all included scripts of a page by using a so-called *Mutation Observer* [22]. As soon as a new script node is found, it is immediately passed on to our analysis module.

2. **Detecting dynamic code generation based on authentication credentials:** Whenever the analysis is

invoked, our extension requests the script file twice: once with authentication credentials attached to the request, and once without authentication credentials. After the responses have been received, the extension compares both and if they differ, stores them in a central database for later analysis.

In order to allow for valid credentials to be sent along with the request, a necessary prerequisite are valid session cookies. To obtain these, the user needs to manually log in to the application under investigation beforehand.

The final step in this phase is the manual analysis of the gathered data to precisely determine which scripts have a dynamic nature depending on the user's session state rather than randomness (such as banner rotation scripts).

4.1.3 Data Set

Unlike general vulnerabilities, the detection of potential data leakages through dynamic JavaScript generation requires an active user account (or a similar stateful relationship) at the tested site, so that the scripts are generated in the context of an authenticated Web session.

Since this requires initial manual registration and account set up on sites we want to test, the size and the nature of our data set is limited. We therefore chose the 150 highest ranking (according to Alexa) Web pages matching the following criteria:

1. Account registration and login is freely available for anyone. This excludes, services that have only paid subscription models or require country-dependent prerequisites (such as a mobile phone number).

2. Available in either German, English or a Web site which can be translated using Google Translate. If this is not given, the set up of meaningful user accounts was not feasible.

3. Not a duplicate or localized variant of an already investigated site (e.g. google.com vs. google.co.in)

After manually registering accounts on these sites, we investigated the site employing the methodology and techniques previously explained, thoroughly interacting with the complete functionality of the sites by adding, processing and viewing plausible data within the different Web applications.

4.2 Prevalence of Dynamic Scripts

The first goal of our study was to count the number of Web sites that make use of dynamic script generation. In the course of this study, using our aforementioned methodology, we gathered a total of 9,059 script files spread across 334 domains and their subdomains. Although our data set only consists of 150 different domains, we gathered scripts from such a large number of domains due to the fact that the investigated Web sites include third-party frames pointing to, e.g., advertisement providers. In a first step, we therefore filtered out scripts from all sites not directly related to the domains under investigation.

Out of these, we found that over half of the sites—81 out of the 150 analyzed domains—utilized some form of dynamic script generation. In a subsequent manual examination step we removed dynamic scripts which only exposed changes in apparently random token values (see below for details), resulting in 209 unique scripts on 49 domains, that were dependent on a user's session state. In relation to our initial data set of 150 domains, this shows that the usage of state-dependent dynamic scripts is widespread, namely one third of the investigated domains.

4.3 Purposes of Dynamic Scripts

We analyzed the applications to ascertain the underlying purpose motivating the utilization of the dynamic scripts. In doing so, we found three categories of use cases as well as a few purposes which could not be categorized. Since these were only single use cases specific to one application, we do not outline these any further but instead put them in the *Others* category. The results of our categorization are depicted in Table 1, showing the total amount of domains per category as well as the highest Alexa rank.

The most commonly applied use case was **retrieval of user-specific data**, such as the name, email address or preferences for the logged-in user. This information was used both to greet users on the start page as well as to retrieve user-provided settings and profile data on the corresponding edit pages. We observed that a number of Web applications utilized modal dialogs to present the profile data forms to the user, whereas the HTML code of said form was embedded into the document already and all currently stored values were retrieved by including a dynamic script.

The second category of scripts we found was **service bootstrapping**, i.e., setting up variables necessary for a rich client-side application to work. One example of such a bootstrapping process was observed in a popular free-mail service's file storage system in which the UI was implemented completely in JavaScript. When initially loading the page, the dynamic script we found provided a secret token which was later used by the application to interact with the server using XMLHttpRequests.

Category	# domains	Highest rank
Retrieval of user-specific data	16	7
Service bootstrapping	15	5
Cross-service data sharing	5	8
Others	13	1

Table 1: Amounts and highest Alexa rank of domains with respect to their use case

The third widely witnessed use case was **cross-service data sharing**, which was often applied to allow for single sign-on solutions across multiple services of the same provider or for tracking of users on different domains through a single tracking service. The latter was evidenced by the same script being included across a multitude of domains from different service providers.

4.4 Types of Security Sensitive Data

In a next step, we conducted a manual analysis of the scripts' data that changed its value, depending on the authentication state of the script request. Within our data, we identified four categories of potentially security-critical data:

- **Login state**: The first type of data that could be extracted from dynamic scripts was a user's login state to a certain application. We found that this happened either explicitly, i.e., assign a variable differently if a user is logged in – or implicitly, e.g. in cases where a script did not contain any code if a user was not logged in.

- **Unique identifiers**: The second category we discovered was the leakage of data that uniquely identified the user. Among these values are customer or user IDs as well as email addresses with which a user was registered to a specific application.

- **Personal data**: In this category we classified all those pieces of data which do not necessarily uniquely identify a user, but provide additional information on him, such as his real name, his location or his date of birth.

- **Tokens & Session IDs**: The last category we encountered were tokens and session identifiers for an authenticated user. These tokens potentially provide an attacker with the necessary information to interact with the application in the name of the user.

Table 2 depicts our study's results with respect to the occurrences of each category. Please note, that a given domain may carry more than one script containing security sensitive information and that a given script may fit into more than one of the four categories.

Data	domains	exploitable	highest rank
Login state	49	40	1
Unique Identifiers	34	28	5
Personal data	15	11	11
Tokens & Session IDs	7	4	107

Table 2: Sensitive data contained in dynamic scripts

The following sections give a more detailed insight into these numbers. The final column shows the highest rank of any domain on which we could successfully extract the corresponding data, i.e., on which we could bypass encountered protection mechanisms.

4.5 Exploitation

In the following, we discuss several attacks which leverage the leakage of sensitive user information. After outlining potential attack scenarios, we discuss several concrete examples of attacks we successfully conducted against our own test accounts.

4.5.1 Utilizing Login Oracles

In the previous section, we discussed that 49 domains had scripts which returned somewhat different content if the cookies for the logged in user were removed. In our notion, we call these scripts *login oracles* since they provide an attacker with either explicit or implicit information on whether a user is currently logged into an account on a given website or not. However, out of these domains, nine domains had scripts with unguessable tokens in the URL, therefore these cannot be utilized as login oracles unless the tokens are known, leaving 40 domains with login oracles.

The most prominent script we found to show such a behavior is hosted by Google and is part of the API for Google Plus. This script, which has a seemingly static address, shows differences in three different variables, namely `isLoggedIn`, `isPlusUser` and `useFirstPartyAuthV2` and hence enables an attacker to ascertain a user's login status with Google.

The information obtained from the oracles can be utilized to provide additional bits to fingerprinting approaches [7]. It may however also be used by an attacker to perform a service-specific phishing attack against his victim. Oftentimes, spam emails try to phish user credentials from banks or services the receiving user does not even have an account on. If, however, the attacker knows with certainty that the user currently visiting his website is logged in to, e.g., google.com, he can display a phishing form specifically aimed at users of Google. This attack can also be improved if additional information about the user is known – we will discuss this attack later in this section.

4.5.2 Tracking Users

Out of the 40 domains which provided a login oracle, 28 also provided some pieces of data which uniquely identify a user. Among these features, the most common identifier was the email address used to register for the corresponding service, followed by some form of user ID (such as login name or customer ID). These features can be used to track users even across device platforms, given that they log in to a service leaking this information. The highest-rated service leaking this kind of unique identifier was a top-ranked Chinese search engine. Following that, we found that a highly-frequented page which features a calendar function also contained a script leaking the email address of the currently logged in user. Since the owning company also owns other domains which all use a single sign-on, logging in to any of these sites also enabled the attack.

4.5.3 Personalized Social Engineering

In many applications, we found that email addresses were being leaked to an attacker. This information can be leveraged to construct highly-personalized phishing attacks against users. As Downs et al. [5] discovered, users tend to react on phishing emails in more of the cases if they have a standing business relationship with the sending entity, i.e. have an account on a given site, or the email appears to be for them personally.

Hence, gathering information on sites a user has an account on as well as retrieving additional information such as his name can aid an attacker in a personalized attack. An attacker may choose to abuse this in two ways – first and foremost, trying to send phishing mails to users based on the services they have accounts. However, by learning the email address and hence email provider of the user, an attacker may also try to phish the user's mail account. In our study, we found that 14 different domains leak email addresses and out of these, ten domains also revealed (at least) the first name of the logged in user.

In addition, two domains leaked the date of birth and one script, hosted on a Chinese Web site, even contained the (verified) mobile phone number of the victim. We believe that, especially considering the discoveries by Downs et al., all this information can be leveraged towards creating highly-personalized phishing attacks.

Another form of personalized social engineering attacks enabled by our findings is targeted advertisement. We found that two online shopping platforms utilize a dynamic script which provides the application with the user's wish list. This information can be leveraged by an attacker to either provide targeted advertisements aimed at profiting (e.g. linking to the products on Amazon, using the attacker's affiliate ID) or to sell fake products matching the user's wishes.

Application-Specific Attacks Alongside the theoretical attack scenarios we discussed so far, we found multiple applications with issues related to the analyzed leaking scripts as well as several domains with CSRF flaws. In the following, we discuss these attacks briefly.

Extracting Calendar Entries: One of the most prominent Web sites we could successfully exploit was a mail service which offers a multitude of additional functionality such as management of contacts and a calendar. The latter is implemented mostly in JavaScript and retrieves the necessary bootstrap information when the calendar is loaded. This script, in the form a function call to a custom JavaScript API, provides the application with all of the user's calendars as well as the corresponding entries. This script was not protected against inclusion by third-party hosts and hence, leaks this sensitive information to an attacker. Alongside the calendar's and entries, the script also leaks the e-mail address of the victim, therefore allowing the attacker to associate the appointments to their owner.

Reading Email Senders and Subjects: When logging in to the portal for a big Chinese Web service provider, we found that the main page shows the last five emails for the currently logged in user. Our browser extension determined that this information was provided by an external script, solely using cookies to authenticate the user. The script contained the username, amount of unread emails and additionally the senders and subjects as well as the received dates for the last five emails of the victim. An abbreviated excerpt is shown in Listing 6. Although this attack does not allow for an actual extraction of the content of an email, at the very least contacts and topics of current discussions of the victim are leaked which we believe to be a major privacy issue.

Listing 6 Excerpt of the script leaking mail information

```
var mailinfo = {
  "email": "user@domain.com",
  ...,
  "maillist": [{
    "mid": "0253FE71.....001",
     "mailfrom":"First Last <firstlast@gmail.com>",
    "subject":"Top secret insider information",
    "ctime": "2014-05-02 21:11:46"}]
  ..}
```

Session Hijacking Vulnerabilities: During the course of our study, we found that two German file storage services contained session hijacking vulnerabilitie. Both these services are implemented as a JavaScript application, which utilizes XMLHttpRequest to retrieve directory listings and manage files in the storage. To avoid unauthorized access to the system, both applications require a session key to be present within a cookie as well as in an additional HTTP header. When first visiting the file storage service, the application loads an external script called `userdata.js` which contains the two necessary secrets to access the service: the username and the aforementioned session key. We found that this script is not properly protected against cross-domain data leakage, allowing an attacker to leak the secret information. With this information at hand, we were able to list and access any file in the victim's file storage. Furthermore, it enabled us to invoke arbitrary actions in the name of the user such as creating new files or deleting existing ones.

One minor drawback in this attack is the need for the attacker to know the victim's username in advance, since the dynamic script requires a GET parameter with the username. Regardless, we believe that by either targeted phishing emails or retrieving the email address through another service (as discussed earlier) this attack is still quite feasible.

Circumventing CSRF Protection: One way of preventing cross-domain attacks is the use of CSRF tokens, namely secrets that are either part of the URL (as a GET parameter) or need to be posted in a form and can then be verified by the server. Although CSRF tokens are a well-understood means of preventing these attacks and provide adequate security, the proper implementation is a key factor. In our analysis, we found that two domains contained scripts which leaked just these critical tokens.

The first one was present on a new domain, which required the knowledge of two secrets in order to change profile data of the user – a 25 byte long token as well as the numerical user ID. While browsing the Web site, our extension detected a state-dependent dynamic script that exactly contained these two values. As a consequence, we were able to leak this data and use it to send a properly authenticated profile change request to the corresponding API. As a consequence, we were able to arbitrarily change a user's profile data. Interestingly, one field that was only visible to the user himself contained a stored XSS vulnerability. Hence, we were able to send a Cross-Site Scripting payload within this field to exploit the, otherwise unexploitable, XSS flaw.

Apart from the obvious issues an XSS attack could cause, for a user logged in via the Facebook Social Login, we could retrieve the Facebook API access token and hence interact with the Facebook API in the name of the user, accessing profile information and even make posts in the name of the user.

Similar to the first finding, we found an issue on the highly-ranked domain of a weather service. The application provides an API for changing a user's profile as well as the password, whereas the old password does not need to be entered to set a new one. Nevertheless, the API requires knowledge of the email address of the currently logged in user, thereby employing at least a variant of a CSRF token. Similar to the previously outlined flaw, we found a script that provides information on the user – among which also the email address is contained. Hence, we could successfully automate the attack by first retrieving the necessary token (email) from the leaking script and subsequently sending a password change request to the API. Afterwards, we sent both the email address (which is also used as the login name) and the new password back to our servers, essentially taking over the user's account in a fully automated manner.

4.5.4 Notification of Vulnerable Sites

In order to allow affected pages to fix the vulnerabilities before they can be exploited, we notified the security teams of all domains for which we could successfully craft exploits. To allow for a better understanding of the general vulnerability as well as the specifics of each domain, we created a Web site detailing the problem associated with cross-domain includes of JavaScript and the attack pattern. In addition, we created proof-of-concept exploits for each flaw and shared this information, augmented by a description of the problem and its impact, e.g., the potential to hijack a user's session, with the domains owners.

As of this writing, we received only three replies stating that the flaw was either being dealt with or had been fixed already. However, none of the affected sites agreed to be mentioned in the paper, therefore we opted to anonymize all the vulnerable services we discovered.

4.5.5 Summary of Our Findings

In total, we found that out of the 49 domains which are dependent on the user's login state, 40 lack adequate protection and can therefore be used to deduce if a user is logged into a certain application. On 28 of these domains, dynamic scripts allowed for unique identification of the current user through various means like customer IDs or email addresses.

Additionally and partly overlapping with the aforementioned scripts, we found that personal data (such as the name or location) was contained in scripts on 13 domains. Last but not least, we encountered four domains which allow for extraction of tokens that could in turn be used to control the target application in the name of the victimized user. An overview of these results is depicted in Table 2.

4.6 Non-exploitable Situations

As shown in Table 2, we were not able to leak data from all of the dynamic scripts we found. In general, we identified two different reasons for this: Either the URL of the script was not guessable by an attacker or the Web site utilized referrer checking to avoid the inclusion of resources by third parties. While these mechanisms protected some Web sites from being exploitable, we believe that the corresponding countermeasures were not placed intentionally against the described attack, but were rather in place because of the used application framework (Referrer checking) or because of the application's design (unguessable URLs). In this section, we briefly discuss and analyze these situations.

4.6.1 Unguessable URLs

A prerequisite for the attack described in this paper is that an attacker is able to include a certain script file into his page during a user's visit. For this, the attacker needs to know the exact URL under which a certain dynamic script is available.

Some of the scripts we found required a session ID or another unguessable token to be present in a GET parameter of the URL. As the attacker is in general not able to obtain such a session ID, the script cannot be included by the attacker and hence sensitive data cannot be leaked.

4.6.2 Referrer Checking

Another technique that prevented us from exploiting a script leakage vulnerability was referrer checking. When a browser generates an HTTP request for an embedded script, it adds the Referer header containing the URL of the embedding site. Many Web pages tend to misuse this header as a security feature [31]. By checking the domain of the referrer, a Web site is in theory able to ascertain the origin of the page requesting a resource.

In 2006, however, Johns showed that referrer checking has several pitfalls [17]. As the Referer header was never intended to serve as a security feature, it should not be used as a reliable source of information. So, for example, many proxies and middle boxes remove the Referer header due to privacy concerns. Furthermore, several situations exist in which a browser does not attach a Referer header to a request and as discussed by Kotowicz, an attacker can intentionally remove the header from requests [19].

As a consequence, servers should not rely on the presence of the Referer header. Hence, if a server receives a request for a dynamic script that does not provide a Referer header, it needs to decide whether to allow the request or whether to block it. If the request is allowed, the attacker may force the removal of the referrer as discussed before. On the other hand, if the server blocks the request (strict referrer checking), it might break the application for users behind privacy-aware proxies.

We found several domains that implemented referrer checking. However, of seven pages that conducted such a check, only two conducted strict referrer checking. As a consequence, the other five Web sites were still exploitable by intentionally removing the Referer header. Listing 7 shows the attack we utilized aiming at stripping the Referer header. In this example, we use a data URI assigned to an iframe to embed the leaking script.

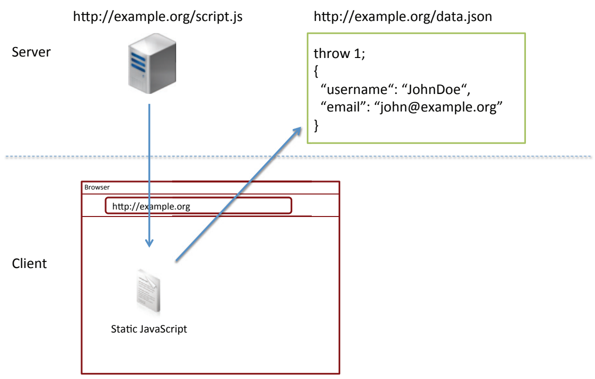
Listing 7 Using a data URL within a frame to send a request without a Referer header

```
var url = "data:text/html,"
    + "<script src=’"
    + "http://example.org/dynamic_script.js"
    + "’></script>"

    + "<script>"
    +  "function leakData(){ ... }; "
    +  "leakData();"
    + "</script>";

// create a new iframe
var frame = document.createElement(’iframe’);
// assign the previously created data url
frame.src = url;
body.appendChild(frame);
```

5 Protection Approach

In our study, we observed a surprisingly high number of popular Web sites utilizing the dangerous pattern of using external, dynamically-generated scripts to provide user-specific data to an application. It seems that developers are not aware of the severe consequences this practice has. In order to improve this situation, we provide a secure and functionally-equivalent solution. The main problem of dynamically generated script includes is the incorporation of sensitive user data into files that are not completely protected by the Same-Origin Policy. We discourage this practice and advise developers to strictly separate JavaScript code from sensitive user data.

Figure 2 depicts our design proposal. In this proposal script code is never generated on the fly, but always pulled from a static file. Sensitive and dynamic data values should be kept in a separate file, which cannot be interpreted by the browser as JavaScript. When the static JavaScript gets executed, it sends an XMLHttpRequest to the file containing the data. By default access



```
throw 1;
{
 “username“: “JohnDoe“,
 “email”: “john@example.org”
}
```

Figure 2: Protection scheme

to the response of an XMLHttpRequest is governed by the Same-Origin Policy. Hence, third-party sites cannot request the file within the user's authentication context. As a consequence, attackers cannot access the data contained within this file. By using Cross-Origin Resource Sharing (CORS) [28], Web developers are able to selectively grant access to the file to any third party service that requires access.

While the attacker is able to include and execute the static JavaScript file within his page, the corresponding code will be executed in the origin of the attacker's Web site. Hence, when the script code requests the data file, which resides in the origin of the legitimate Web site, the two origins do not match and hence the Same-Origin Policy protects the file's content from being accessed by the attacker. If, however, the legitimate site requests the data files, the two origins match and thus access is granted.

As the data file does not contain valid JavaScript code, it cannot be included and executed by the attacker via the HTML `script` tag. To completely avoid this risk, Web developers can either include a so-called unparseable cruft to the beginning of file which causes a compile time failure or add valid JavaScript that effectively stops execution during run time, such as an uncatchable exception (cp. Figure 2) [31].

6 Related Work

Conceptually closest to the attacks presented in Section 4.5 is *JSON Hijacking*, an exploitation technique initially presented by Grossman in 2006 [9]. In his attack he utilized a cross-domain script include pointing to a JSON-array resource, which originally was intended as an end-point for an XMLHttpRequest. Via using a non-standard redefinition of JavaScript's object constructor, he was able obtain the content of the user's GMail address book. Grossman reported the issue to Google, where the term Cross-Site Script Inclusion (XSSI) was coined by Christoph Kern. Kern later mentioned the term publicly for the first time in his book from 2007 [18]. Several other authors later on picked up this term to refer to slight variations of the attack [27, 31].

At the same time Chess et al. [3] picked up Grossman's technique, slightly generalized it and coined the term *JavaScript Hijacking*. Unlike the vulnerabilities in this paper, these attacks do not target dynamic JavaScript resources. Instead they use `script`-tags in combination with a non-standard JavaScript quirk (that has been removed from all major browsers in the meantime) to leak data that is encoded in the JSON-array format.

Furthermore, in 2013, Grossman [11] discussed the idea of utilizing resources which are only accessible by users that are logged in to determine the logon status of a user. He also proposed to employ click-jacking attacks on the user to force him to like the attacker's Facebook or Google+ site. In doing so and in comparing the latest addition to his followers, an attacker could thereby deduce the identity of the user currently visiting his website. The idea of determining a user logon status was picked up by Evans [8], who demonstrated a login oracle on `myspace.com` by including a Cascading Style Sheet file from the service which changed certain properties based on whether the user was logged in or not.

In 2015, Takeshi Terada presented another variation of the attack that he called Identifier-based XSSI [27]. Terada used script tags to reference CSV files from third-party domains. A CSV file usually consists of a comma separated list of alphanumeric words. Under certain circumstances this list also represents a syntactically correct list of JavaScript variable declarations. Hence, by referencing such a file the JavaScript engine will create a set of global variables named like the values in the CSV file. By enumerating all globally accessible variables, Terada was able leak the contents of the file.

Other related work has focused on CSS-based history leakage [13, 10, 14]. Analogously to login state leakage, retrieval of a user's history allows an attacker to deduce that a victim has an account on a given site, hence enabling him to start target phishing attacks similar to the ones we outlined in Section 4.5.3.

Another means of utilizing history leakage was discussed in 2010 by Wondracek et al. [30], who proposed a scheme capable of de-anonymizing users based on their group membership in OSNs. To do so, they utilized the stolen history of a user to determine the group sites the user had previously visited. Comparing these to a list of the members of the corresponding groups allowed the authors to determine the user's identity. Recently, for a poster, Jia et al. [16] discussed the notion of utilizing timing side-channels on the browser cache to ascertain a user's geo location.

In 2012, Nikiforakis et al [24] conducted a large-scale analysis of remote JavaScript, focusing mainly on the potential security issues from including third-party code. For W2SP 2011, two groups [20, 15] conducted an analysis of cross-domain policies for Flash, aiming specifically at determining those domains which allow access from any domain. Since Flash attaches the cookies for the *target* domain to said requests, they discussed attack scenarios in which a malicious Flash applet is used to retrieve proprietary information.

In addition to these attacks, Paul Stone demonstrated another means of stealing sensitive information across origin boundaries. To do so, he leveraged a timing side channel, allowing him to leak a framed document pixel by pixel [26].

7 Outlook

The goal of this paper was to conduct an initial study into the usage and potential pitfalls of dynamic scripts in real world applications. Our data set of 150 highly ranked domains gives a good glimpse into the problems caused by such scripts. Nevertheless, we believe that a large-scale study could provide additional key insights into the severity of the issue. To enable such a study, an important problem to solve is the automation of the analysis—starting from fully automated account registration and ranging to meaningful interaction with the application. Therefore, implementing such a generic, yet intelligent crawler and investigating how well it can imitate user interaction is a challenging task we leave for future work. Along with such a broader study, enhancements have to be made to cope with the increased amount of data. As an example, our Chrome extension could use advanced comparisons based on syntactical and semantical differences of the JavaScript code rather than based on content. Since our data set was limited by the fact that our analysis required manual interaction with the investigated applications, the need to automate the secondary analysis steps, i.e., examination of the differences and verification of a vulnerability, did not arise.

Recently, the W3C has proposed a new security mechanism called *Content Security Policy* (CSP), which is a "declarative policy that lets authors of a web application inform the client from where the application expects to load resources" [25]. In its default setting, CSP forbids the usage of inline scripts and hence, programmers are compelled to put the code into external scripts. During our study we noticed that many of these inline scripts are also generated dynamically and incorporate sensitive user data. If all these current inline scripts are naively transformed into dynamic, external script resources, it is highly likely that the attack surface of this paper's attacks will grow considerably.

For instance, Doupé et al. [4] developed a tool called *deDacota* which automatically rewrites applications to adhere to the CSP paradigms by moving all inline script code to external scripts. As our work has shown, these external scripts – if not protected properly – may be included by any third-party application and hence might leak secret data. Therefore, we believe that it is imperative that measures are taken to ensure the secure, yet flexible client-side access to sensitive data and that the changing application landscape caused by CSP adoption is closely monitored. As discussed by Weissbacher et al., however, CSP is not yet widely deployed and significantly lags behind other security measures [29].

Furthermore, in this paper, we exclusively focused on dynamic JavaScript that is pulled into the browser via `script`-tags. This is not necessarily the only method, how server generated script content is communicated. An alternative to `script` tags is to transport the code via XMLHttpRequests bodies, which are subsequently passed to the `eval()` API. In future work, we plan to investigate such XMLHttpRequests endpoints in respect to their susceptibility to attack variants related to this paper's topic.

Finally, as related work has indicated, internal application information, such as the login state of a user, may also be leaked via images or style sheets. In this case, the observed effects of a cross-domain element inclusion manifest themselves through side effects on the DOM level, as opposed to a footprint in the global script object. Hence, a systematical further analysis on other classes of server-side content generation that might enable related attacks would be a coherent extension of our work.

8 Summary & Conclusion

In this paper, we conducted a study into the prevalence of a class of vulnerabilities dubbed Cross-Site Script Inclusion. Whenever a script is generated on the fly and incorporates user-specific data in the process, an attacker is able to include the script to observe its execution behavior. By doing so, the attacker can potentially extract the user-specific data to learn information which he otherwise wouldn't be able to know.

To investigate this class of security vulnerabilities, we developed a browser extension capable of detecting such scripts. Utilizing this extension, we conducted an empirical study of 150 domains in the Alexa Top 500, aimed at gaining insights into prevalence and purpose of these scripts as well as security issues related to the contained sensitive information.

Our analysis showed that out of these 150 domains, 49 domains utilize server-side JavaScript generation. On 40 domains we were able to leak user-specific data leading to attacks such as deanonymizing up to full account hijacking. Our practical experiments show that even high-profile sites are vulnerable to this kind of attacks.

After having demonstrated the severe impact these flaws can incur, we proposed a secure alternative using well-known security concepts, namely the Same-Origin Policy and Cross-Origin Resource Sharing, to thwart the identified security issues.

Acknowledgements

The authors would like to thank the anonymous reviewers for their valued feedback. More over, we want to thank our shepherd Joseph Bonneau for the support in getting our paper ready for publication. This work was in parts supported by the EU Project STREWS (FP7-318097). The support is gratefully acknowledged.

References

[1] Barth, A., Jackson, C., and Mitchell, J. C. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security* (2008), ACM, pp. 75–88.

[2] CERT. Advisory ca-2000-02 malicious html tags embedded in client web requests, February 2000.

[3] Chess, B., O'Neil, Y. T., and West, J. JavaScript Hijacking. [whitepaper], Fortify Software, `http://www.fortifysoftware.com/servlet/downloads/public/JavaScript_Hijacking.pdf`, March 2007.

[4] Doupé, A., Cui, W., Jakubowski, M. H., Peinado, M., Kruegel, C., and Vigna, G. dedacota: toward preventing server-side xss via automatic code and data separation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 1205–1216.

[5] Downs, J. S., Holbrook, M. B., and Cranor, L. F. Decision strategies and susceptibility to phishing. In *Proceedings of the second symposium on Usable privacy and security* (2006), ACM, pp. 79–90.

[6] ECMAScript, E., Association, E. C. M., et al. Ecmascript language specification, 2011.

[7] Electronic Frontier Foundation. Panopticlick – how unique – and trackable – is your browser? online, `https://panopticlick.eff.org/about.php`, last accessed 2014/05/10.

[8] Evans, C. Cross-domain leaks of site logins. online, `http://bit.ly/1lz1HPl`, last accessed 2014/05/10.

[9] Grossman, J. Advanced Web Attack Techniques using GMail. [online], `http://jeremiahgrossman.blogspot.de/2006/01/advanced-web-attack-techniques-using.html`, January 2006.

[10] Grossman, J. I know where you've been. [online], `http://jeremiahgrossman.blogspot.com/2006/08/i-know-where-youve-been.html`, August 2006.

[11] Grossman, J. The web won't be safe or secure until we break it. *Communications of the ACM 56*, 1 (January 2013), 68–72.

[12] Hansen, R., and Grossman, J. Clickjacking. *Sec Theory, Internet Security* (2008).

[13] Jackson, C., Bortz, A., Boneh, D., and Mitchell, J. C. Protecting Browser State from Web Privacy Attacks. In *Proceedings of the 15th ACM World Wide Web Conference (WWW 2006)* (2006).

[14] Jakobsson, M., and Stamm, S. Invasive Browser Sniffing and Countermeasures. In *Proceedings of The 15th annual World Wide Web Conference (WWW2006)* (2006).

[15] Jang, D., Venkataraman, A., Sawka, G. M., and Shacham, H. Analyzing the cross-domain policies of flash applications. In *Proceedings of the 5th Workshop on Web* (2011), vol. 2.

[16] Jia, Y., Dongy, X., Liang, Z., and Saxena, P. I know where you've been: Geo-inference attacks via the browser cache. IEEE Security&Privacy 2014, `http://www.ieee-security.org/TC/SP2014/posters/JIAYA.pdf`, last accessed 2014/05/17.

[17] Johns, M., and Winter, J. Requestrodeo: Client side protection against session riding. *Proceedings of the OWASP Europe 2006 Conference* (2006).

[18] Kern, C., Kesavan, A., and Daswani, N. *Foundations of security: what every programmer needs to know*. Apress, 2007.

[19] Kotowicz, K. Stripping the referrer for fun and profit. online, `http://blog.kotowicz.net/2011/10/stripping-referrer-for-fun-and-profit.html`, last accessed 2014/05/10.

[20] Lekies, S., Johns, M., and Tighzert, W. The state of the cross-domain nation. In *Proceedings of the 5th Workshop on Web* (2011), vol. 2.

[21] Mozilla. Inheritance and the prototype chain. online, `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Inheritance_and_the_prototype_chain`, last accessed 2014/05/10.

[22] Mozilla. Mutationobserver. online, `https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver`, last accessed 2014/05/10.

[23] Mozilla Developer Network, and Ruderman, J. Same-origin policy. online, `https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy`.

[24] Nikiforakis, N., Invernizzi, L., Kapravelos, A., Acker, S. V., Joosen, W., Kruegel, C., Piessens, F., and Vigna, G. You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions. In *19th ACM Conference on Computer and Communications Security (CCS 2012)* (2012).

[25] Sterne, B., and Barth, A. Content security policy 1.0. online, `http://www.w3.org/TR/2012/CR-CSP-20121115/`. last accessed 2014/05/10.

[26] Stone, P. Pixel perfect timing attacks with html5.

[27] Terada, T. Identifier based xssi attacks, 2015.

[28] Van Kesteren, A., et al. Cross-origin resource sharing. *W3C Working Draft WD-cors-20100727* (2010).

[29] Weissbacher, M., Lauinger, T., and Robertson, W. Why is csp failing? trends and challenges in csp adoption. In *Research in Attacks, Intrusions and Defenses*. Springer, 2014, pp. 212–233.

[30] Wondracek, G., Holz, T., Kirda, E., and Kruegel, C. A practical attack to de-anonymize social network users. In *Security and Privacy (SP), 2010 IEEE Symposium on* (2010), IEEE, pp. 223–238.

[31] Zalewski, M. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2012.

ZigZag: Automatically Hardening Web Applications Against Client-side Validation Vulnerabilities

Michael Weissbacher
Northeastern University
mw@ccs.neu.edu

William Robertson
Northeastern University
wkr@ccs.neu.edu

Engin Kirda
Northeastern University
ek@ccs.neu.edu

Christopher Kruegel
UC Santa Barbara
chris@cs.ucsb.edu

Giovanni Vigna
UC Santa Barbara
vigna@cs.ucsb.edu

Abstract

Modern web applications are increasingly moving program code to the client in the form of JavaScript. With the growing adoption of HTML5 APIs such as `postMessage`, client-side validation (CSV) vulnerabilities are consequently becoming increasingly important to address as well. However, while detecting and preventing attacks against web applications is a well-studied topic on the server, considerably less work has been performed for the client. Exacerbating this issue is the problem that defenses against CSVs must, in the general case, fundamentally exist in the browser, rendering current server-side defenses inadequate.

In this paper, we present ZigZag, a system for hardening JavaScript-based web applications against client-side validation attacks. ZigZag transparently instruments client-side code to perform dynamic invariant detection on security-sensitive code, generating models that describe how – and with whom – client-side components interact. ZigZag is capable of handling templated JavaScript, avoiding full re-instrumentation when JavaScript programs are structurally similar. Learned invariants are then enforced through a subsequent instrumentation step. Our evaluation demonstrates that ZigZag is capable of automatically hardening client-side code against both known and previously-unknown vulnerabilities. Finally, we show that ZigZag introduces acceptable overhead in many cases, and is compatible with popular websites drawn from the Alexa Top 20 without developer or user intervention.

1 Introduction

Most of the over 2 billion Internet users [1] regularly access the World Wide Web, performing a wide variety of tasks that range from searching for information to the purchase of goods and online banking transactions. Unfortunately, the popularity of web-based services and the fact that the web is used for business transactions has also attracted a large number of malicious actors. These actors compromise both web servers and end-user machines to steal sensitive information, to violate user privacy by spying on browsing habits and accessing confidential data, or simply to turn them into "zombie" hosts as part of a botnet.

As a consequence, significant effort has been invested to either produce more secure web applications, or to defend existing web applications against attacks. Examples of these approaches include applying static and dynamic program analyses to discover vulnerabilities or prove the absence of vulnerabilities in programs [2, 3, 4, 5], language-based approaches to render the introduction of certain classes of vulnerabilities impossible [6, 7, 8], sandboxing of potentially vulnerable code, and signature- and anomaly-based schemes to detect attacks against legacy programs.

However, despite the large amount of research on preventing attacks against web applications, vulnerabilities persist. This is due to a combination of factors, including the difficulty of training developers to make use of more secure development frameworks or sandboxes, as well as the continuing evolution of the web platform itself.

In particular, advances in browser JavaScript engines and the adoption of HTML5 APIs has led to an explosion of highly complex web applications where the majority of application code has been pushed to the client. Client-side JavaScript components from different origins often co-exist within the same browser, and make use of HTML5 APIs such as `postMessage` to interact with each other in highly dynamic ways.

`postMessage` enables applications to communicate with each other purely within the browser, and are not subject to the classical same origin policy (SOP) that defines how code from mutually untrusted principals are separated. While SOP automatically prevents client-side code from distinct origins from interfering with each others' code and data, code that makes use of `postMessage` is expected to define and enforce their own security policy. While this provides much greater flexibility to application developers, it also opens the door for vulnerabilities to be introduced into web applications due to insufficient origin checks or other programming mistakes.

postMessage is but one potential vector for the more general problem of insufficient client-side validation (CSV) vulnerabilities. These vulnerabilities can be exploited by input from untrusted sources – e.g., the cross-window communication interface, referrer data, and others. An important property of these vulnerabilities is that attacks cannot be detected on the server side, and therefore any framework for defending against them at runtime must execute within the browser. Also, in contrast to other popular web attack classes such as Cross-Site Scripting (XSS), CSVs represent application logic flaws that are closely tied to the intended behavior of the application and, consequently, can be difficult to identify and defend against in a generic, automated fashion.

In this paper, we propose ZigZag, a system for hardening JavaScript-based web applications against client-side validation attacks. ZigZag transparently instruments client-side code to perform dynamic invariant detection over live browser executions. From this, it derives models of the normal behavior of client-side code that capture essential properties of how – and with whom – client-side web application components interact, as well as properties related to control flows and data values within the browser. Using these models, ZigZag can then automatically detect deviations from these models that are highly correlated with client-side validation attacks.

We describe an implementation of ZigZag as a proxy, and demonstrate that it can effectively defend against vulnerabilities found in the wild against real web applications without modifications to the browser or application itself aside from automated instrumentation. In addition, we show that ZigZag is efficient, and can be deployed in realistic environments without a significant impact on the user experience.

In summary, this paper makes the following contributions:

- We present a novel in-browser anomaly detection system based on dynamic invariant detection that defends clients against previously unknown client-side validation attacks.
- We present a new technique we term *invariant patching* for extending dynamic invariant detection to server-side JavaScript templates, a very common technique for lightweight parameterization of client-side code.
- We extensively evaluate both the performance and security benefits of ZigZag, and show that it can be effectively deployed in several real scenarios, including as a transparent proxy or through direct application integration by developers.

The rest of the paper is organized as follows. In Section 2, we motivate the need for defending against client-side validation vulnerabilities through the introduction of a running example and define our threat model. In Section 3, we present the high-level design of ZigZag. Sections 4 and 5 describe the details of ZigZag's invariant detection and enforcement. We then evaluate a prototype implementation of ZigZag in Section 6. Finally, Sections 7 and 8 discuss related work and conclude the paper.

2 Motivation and Threat Model

To contextualize ZigZag and motivate the problem of client-slide validation vulnerabilities, we consider a hypothetical webmail service. This application is composed of code and resources belonging both to the application itself as well as advertisements from multiple origins. Since these origins are distinct, the same origin policy applies, and code from each of these origins cannot interfere with the others. This type of origin-based separation is typical for modern web applications.

However, in this example, the webmail component communicates with the advertising network via postMessage to request targeted ads given a profile it has generated for its users. The ad network can respond that it has successfully placed ads, or else request further information in the case that a suitable ad could not be found. Figure 1 shows one side of this communication channel, where the advertising component both registers an onMessage event listener to receive messages from the webmail component, as well as sends responses using the postMessage method. In this case, because the ad network does not verify the origin of the messages it receives, it is vulnerable to a client-side validation attack [9].

To tamper with the ad network, an attacker must be able to invoke postMessage in the same context. This can be achieved by exploiting XSS vulnerabilities from user content, framing the webmail service, or exploiting a logic vulnerability. Hence, the attacker has to send an email to a victim user that contains XSS code, or lure the victim to a site that will frame the webmail service.

Despite the fact that the ad network component is vulnerable, ZigZag prevents successful exploitation of the vulnerability. With ZigZag, the webmail service is used through a transparent proxy that instruments the JavaScript code, augmenting each component with monitoring code. The webmail service then runs in a training phase where execution traces of the JavaScript programs are collected. Collected data points include function pa-

```
// Handle a received message
var receiveMessage = function(e) {
    // Missing check on e.origin!
}

var sendMessage = function(e) {
    // Send data to window ‘w’
    w.postMessage(data, ’*’);
}

// Register for messages
window.addEventListener(”message”, receiveMessage, false);
```

Figure 1: Insecure usage of the `postMessage` API in a hypothetical webmail client-side component.

rameters, caller/callee pairs, and return values. Once enough execution traces have been collected, ZigZag uses invariant detection to establish a model of normal behavior. Next, the original program is extended with enforcement code that detects deviations from the baseline established during training. Execution is compared against this baseline, and violations are treated as attacks.

In this example, ZigZag would recognize that messages received by the ad network must originate from the webmail component’s origin, and would terminate execution if a message is received from another origin – for instance, from the user content origin. Due to the nature of CSV vulnerabilities, this attack would go unnoticed for server-side invariant detection systems such as Swaddler [10] as they focus on more traditional web attacks against server-side code. These attacks can either happen on the client alone, where such systems have no visibility, or when server interaction is triggered through exploitation of a CSV vulnerability. In addition, these requests are indistinguishable from benign user interaction. We stress that this protection requires no changes to the browser or application on the server, and is therefore transparent to both developers and users alike.

We expand upon this example service with more vulnerabilities and learned invariants in following sections.

2.1 Threat model

The threat model we assume for this work is as follows. ZigZag aims to defend benign-but-buggy JavaScript applications against attacks targeting client-side validation vulnerabilities, where CSV vulnerabilities represent bugs in JavaScript programs that allow for unauthorized actions via untrusted input.

The attacker can provide input to JavaScript programs through cross-window communication (e.g., `postMessage`), or window/frame cross-domain properties. This can be performed by operating in an otherwise isolated JavaScript context within the same browser. However, the attacker cannot run arbitrary code in a ZigZag-protected context without first bypassing ZigZag, an eventuality we aim to prevent. In particular, we presume the presence of complementary defenses against XSS-based code injection attacks such as Content Security Policy (CSP) [11] or rigorous template auto-sanitization. Therefore, we assume that attackers cannot directly tamper with ZigZag invariant learning and enforcement by, for instance, overwriting these functions in the JavaScript context without first evading the system.

Because ZigZag depends on a training set to learn dynamic invariants, we assume that the training data is trusted and, in particular, attack-free. This is a general limitation of anomaly-based detection schemes, though one that also has partial solutions [12].

3 System Overview

ZigZag is an in-browser anomaly detection system that defends against client-side validation (CSV) vulnerabilities in JavaScript applications. ZigZag operates by interposing between web servers and browsers in order to transparently instrument JavaScript programs. This instrumentation process proceeds in two phases.

Learning phase. First, ZigZag rewrites programs with monitoring code to collect execution traces of client-side code. These traces are fed to a dynamic invariant detector that extracts likely invariants, or models. The invariants that ZigZag extracts are learned over data such as function parameters, variable types, and function caller and callee pairs.

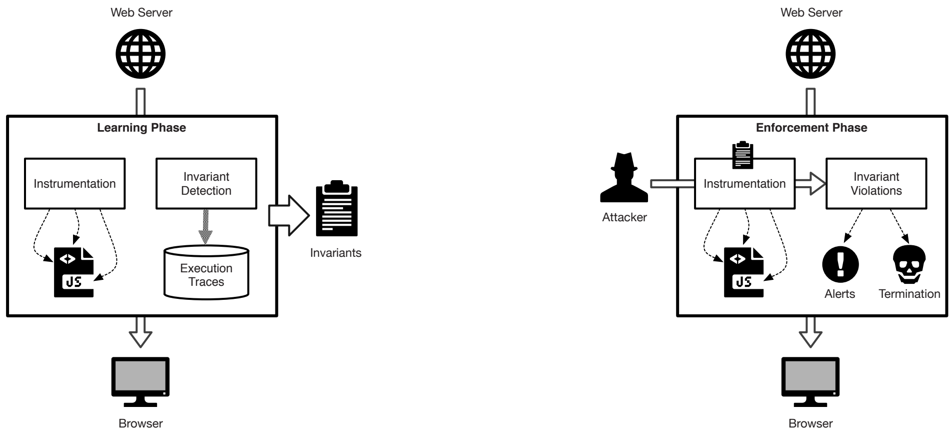
Enforcement phase. In the second phase, the invariants that were learned in the initial phase are used to harden the client-side components of the application. The hardened version of the web application preserves the semantics of the original, but also incorporates runtime checks to enforce that execution does not deviate from what was observed during the initial learning phase. If a deviation is detected, the system assumes that an attack has occurred and execution is either aborted or the violation is reported to the user.

An overview of this system architecture is shown in Figure 2. We note that instrumentation for both the learning phase and enforcement phase is performed *once*, and subsequent accesses of an already instrumented program re-use a cached version of that program.

In the following sections, we describe in detail each phase of ZigZag’s approach to defending against client-side validation vulnerabilities in web applications.

4 Invariant Detection

In this section, we focus on describing the invariants ZigZag learns, why we selected these invariants to enforce, and how we extract these invariants from client-side code.



(a) Learning phase. A JavaScript program is instrumented in order to collect execution traces. Invariant detection is then performed on the trace collection in order to produce a set of likely invariants.

(b) Enforcement phase. Given a JavaScript program and the invariants previously learned, instrumentation is again used to enforce those invariants.

Figure 2: ZigZag overview. Instrumentation is used in both the learning and enforcement phases to produce and enforce likely invariants, respectively. Note that instrumentation is only performed *once* in each case; subsequent loads use a cached instrumented version of the program.

Data Type	Invariants
All	Types
Numbers	Equality, inequality, oneOf
String	Length, equality, oneOf, isJSON, isPrintable, isEmail, isURL, isNumber
Boolean	Equality
Objects	All of the above for object properties
Functions	Calling function, return value

Table 1: Invariants supported by ZigZag.

4.1 Invariant Detection

Dynamic program invariants are statistically-likely assertions established by observing multiple program executions. We capture program state at checkpoints and compare subsets of these states for each individual checkpoint (we define checkpoints in further detail in Section 4.2). The underlying assumption is that invariants should hold not only for the observed executions, but also for future ones.

However, there is no guarantee that invariants will also hold in the future. Therefore, ZigZag only uses invariants which should hold with a high probability. These invariants are later used to decide whether a program execution is to be considered anomalous. By capturing state dynamically, ZigZag has insight into user behavior that purely static systems lack.

ZigZag uses program execution traces to generate Daikon [13] dtrace files. These dtrace files are then generalized into likely invariants with a modified version of Daikon we have developed. Daikon is capable of generating both univariate and multivariate invariants. Univariate invariants describe properties of a single variable; examples of this include the length of a string, the percentage of printable characters in a string, and the parity of a number. Multivariate models, on the other hand, describe relations between two or more variables, for example $x == y$, $x + 5 == y$, or $x < y$.

ZigZag analyzes multivariate relationships within function parameters, return values, and invoking functions. In addition, we extended the invariants provided by Daikon with additional ones, including checks on whether a string is a valid JSON object, URL, or email address.

For example, when used on a website with `postMessage`, ZigZag could learn that the origin attribute of the `onMessage` event is both printable and a URL, or equal to a string. Depending on the number of different origins, the system could also learn the legitimate set of sending origins

$$\text{v0.origin} \in \{o_1, \ldots, o_n\}.$$

As another example, since JavaScript is a dynamically typed language, it has no type annotations in function signatures. This language feature can lead to runtime errors or be exploited by an attacker. By learning likely type invariants over function parameters and return values that are checked during the enforcement phase, ZigZag can (partially) retrofit types into JavaScript programs. For example, this can become security-relevant when developers use numeric values for input, and do not consider other values during input sanitization. We describe an example of parameter injection, and how the attack is thwarted by ZigZag in Section 6.1.

One pitfall of anomaly detection is undertraining. To reduce the impact on our system, we check function coverage before issuing invariants for enforcement. We only allow for enforcement of a particular function after execution traces from four or more training sessions were collected, which was sufficient for the examples we considered. This threshold, however, is configurable and can easily be increased if greater variability is observed during invariant learning.

The full set of invariants supported by ZigZag is shown in Table 1.

4.2 Program Instrumentation

Trace collection and enforcement code is inserted at program points we refer to as *checkpoints*. The finest supported granularity is to insert checkpoints for every statement. However, while this is possible, statement granularity introduces unacceptable overhead with little benefit. The CSV vulnerabilities we have observed in the wild can be detected with a coarser and more efficient level of granularity. Since events such as receiving cross-window communication are handled by functions, function entry and exit points are natural program points to analyze input and return data. Consequently, for our prototype we opted to insert checkpoints at function prologues and epilogues.

During instrumentation, ZigZag performs a lightweight static analysis on the program's abstract syntax tree (AST) to prune the set of checkpoints that must be injected. Functions which contain `eval` sinks, XHR requests, access to the `document` object, and other potentially harmful operations are labeled as important. Only these functions are used in data collection and enforcement mode. As a consequence, large programs that only have few potentially harmful operations will have significantly less overhead as compared to instrumenting the entire program, while at the same time preserving the security of the overall approach. Aside from increased performance, whitelisting functions that are known not to be security-relevant also leads to a reduced risk of false positives.

Each function labeled as important during the static analysis phase is instrumented with pre- and post-function body hooks called `calltrace` and `exittrace`. The original return statement is inlined in the `exittrace` function call and returned by it. These functions access the instrumented function's parameters through the standard `arguments` variable, and either records a program state for invariant detection (learning phase) or checks for an invariant violation (enforcement phase).

```
function x(a, b) {
    // function body
    ...
    return a+b;
}
```

(a) Function body before instrumentation

```
function x(a, b) {
    var callcounter = __calltrace(functionid,
                                  codeid,
                                  sessionid);
    // function body
    ...
    return __exittrace(functionid,
                       callcounter,
                       subexitid,
                       codeid,
                       sessionid,
                       a+b);
}
```

(b) Function body after instrumentation

Figure 3: Function instrumentation example.

ZigZag uses a number of identifiers to label program states at checkpoints. `functionid` uniquely identifies functions within a program, `codeid` labels distinct JavaScript programs, and `sessionid` labels program executions. The variables `functionid` and `codeid` are hardcoded during program instrumentation, while `sessionid` is generated for each request.

The `callcounter` variable is used instead to connect call chains. Every invocation of `calltrace` increments and returns a global `callcounter` to provide a unique identifier such that `calltrace` and `exittrace` invocations can be matched. This is necessary since JavaScript is re-entrant, and therefore multiple threads of execution can invoke a function and yield before returning, potentially resulting in out-of-order pre- and post-function hook invocations.

ZigZag can not only instrument the code initially loaded by a site, but also code dynamically downloaded during execution. JavaScript programs can potentially modify themselves at runtime, since a program can generate code for its own execution. We address this by wrapping `eval` invocations, script tag insertion, and writes to the DOM. Our wrapper sends the new program code to the proxy and calls the original function with the instrumented program. This technique has been shown to be effective in prior work [14].

In our prototype implementation, each of these calls incurs a roundtrip to the server, where such code is treated the same way as non-`eval` code. As a possible optimization, the instrumented version of previously observed data passed to `eval` could be inlined with the enclosing (instrumented) program, removing the need for subsequent separate roundtrips. Furthermore, we often observed `eval` to be used for JSON deserialization. If such a use case is detected, instrumentation could be bypassed entirely. However, we did not find it necessary to implement these features in our research prototype.

The `calltrace` and `exittrace` functions reside in the same scope since they must be callable from all functions. An example of uninstrumented and instrumented code is shown in Figures 3a and 3b, respectively.

5 Invariant Enforcement

Given a set of invariants collected during the learning phase, ZigZag then instruments JavaScript programs to enforce these invariants. Since templated JavaScript is a prevalent technique on the modern web for lightweight parameterization of client-side code, we then present a technique for adapting invariants to handle this case. Finally, we discuss possible deployment scenarios and limitations of the system.

Daikon supports invariant output for several languages, including C++, Java, and Perl. However, it does not support JavaScript by default. Groeneveld et al. implemented extensions to Daikon to support invariant analysis using Daikon [15]. However, we found that their implementation was not capable of generating JavaScript for all of the invariants ZigZag must support, and therefore we wrote our own implementation.

In our implementation, the `calltrace` and `exittrace` functions perform a call to an enforcement function generated for each function labeled important during the static analysis step. `calltrace` examines the function input state, while `exittrace` examines the return value of the original function. These functions are generated automatically by ZigZag for each important function. Based on the invoking program point, assertions corresponding to learned invariants are executed. Should an assertion be violated, a course of action is taken depending on the system configuration. Options include terminating execution by navigating away from the current site, or alternatively reporting to the user that a violation occurred and continuing execution. Figure 4 shows a possible instance of the `calltrace` function, abbreviated for clarity.

5.1 Program Generalization

Modern web applications often make use of lightweight templates on the server, and sometimes within the browser as well. These templates usually take the form of a program snippet or function that largely retains the same structure with respect to the AST, but during instantiation placeholders in the template are substituted with concrete data – for instance, a timestamp or user identifier. This is often done for performance, or to reduce code duplication on the server. As an example, consider the templated version of the webmail example shown in Figure 5.

```
__calltrace = function(functionid, codeid, sessionid) {
    // Enforcement
    var v0 = arguments.callee.caller.caller.arguments[0];
    var v1 = ...

    if ( functionid === 0 ) {
        __assert(typeof(v0) === ’number’ && v0 > 5);
        __assert(typeof(v1) === ’string’ && v1 === ”x”);
        ...
    } else if ( functionid === 1 ) {
        ...
    }
    ...
    return __incCallCounter();
}
```

Figure 4: Example of invariant enforcement over a function’s input state.

```
// Server-side JavaScript template
var state = {
    user: {{username}},
    session: {{sessionid}}
};

// Client-side JavaScript code after template instantiation
var state = {
    user: ”UserX”,
    session: 0
};
```

Figure 5: Example of a JavaScript template.

Due to the cost of instrumentation and the prevalence of this technique, this mix of code and data poses a fundamental problem for ZigZag since a templated program causes – in the worst case – instrumentation on every resource load. Additionally, each template instantiation would represent a singleton training set, leading to artificial undertraining. Therefore, it was necessary to develop a technique for both recognizing when templated JavaScript is present and, in that case, to generalize invariants from a previously instrumented template instantiation to keep ZigZag tractable for real applications.

ZigZag handles this issue by using efficient structural comparisons to identify cases where templated code is in use, and then performing *invariant patching* to account for the differences between template instantiations in a cached instrumented version of the program.

Structural comparison. ZigZag defines two programs as structurally similar and, therefore, candidates for generalization if they differ only in values assigned to either primitive variables such as strings or integers, or as members of an array or object. Objects play a special role as in template instantiation properties can be omitted or ordered non-deterministically. As a result ASTs are not equal in all cases, only similar. Determining whether this

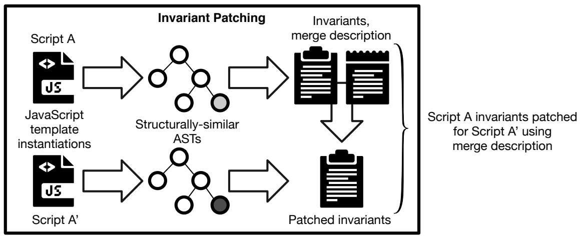

Invariant Patching
Script A
Invariants, merge description
JavaScript template instantiations
Structurally-similar ASTs
Script A invariants patched for Script A' using merge description
Script A'
Patched invariants

Figure 6: Invariant patching overview. If ZigZag detects that two JavaScript programs are structurally isomorphic aside from constant assignments, a merge description is generated that allows for efficient patching of previously-generated invariants. This scheme allows ZigZag to avoid re-instrumentation of templated JavaScript on each load.

is the case could be performed by pairwise AST equality that ignores constant values in assignments and normalizes objects. However, this straightforward approach does not scale when a large number of programs have been instrumented.

Therefore, we devised a string equality-based technique. From an AST, ZigZag extracts a string-based summary that encodes a normalized AST that ignores constant assignments. In particular, normalization strips all constant assignments of primitive data types encountered in the program. Also, assignments to object properties that have primitive data types are removed. Objects, however, cannot be removed completely as they can contain functions which are important for program structure. Removing primitive types is important as many websites generate programs that depend on the user state – e.g., setting `{logged_in: 1}` or omitting that property depending on whether a user is logged in or not. Removing the assignment allows ZigZag to correctly handle cases such as these.

Furthermore, normalization orders any remaining object properties such as functions or enclosed objects, in order to avoid comparison issues due to non-deterministic property orderings. Finally, the structural summary is the hash of the reduced, normalized program.

As an optimization, if the AST contains no function definitions, ZigZag skips instrumentation and serves the original program. This check is performed as part of structural summary generation, and is possible since ZigZag performs function-level instrumentation.

Code that is not enclosed by a function will not be considered. Such code cannot be addressed through event handlers and is not accessible through `postMessage`. However, calls to `eval` would invoke a wrapped function, which is instrumented and included in enforcement rules.

Fast program merging. The first observed program is handled as every other JavaScript program because ZigZag cannot tell from one observation whether a program represents a template instantiation. However, once ZigZag has observed two structurally similar programs, it transparently generates a *merge description* and *invariant patches* for the second and future instances.

The merge description represents an abstract version of the observed template instantiation that can be patched into a functional equivalent of new instantiations. To generate a merge description, ZigZag traverses the full AST of structurally similar programs pairwise to extract differences between the instantiations. Matching AST nodes are preserved as-is, while differences are replaced with placeholders for later substitution. Next, ZigZag compiles the merge description with our modified version of the Closure compiler [16] to add instrumentation code and optimize.

The merge description is then used every time the templated resource is subsequently accessed. The ASTs of the current and original template instantiations are compared to extract the current constant assignments, and the merge description is then patched with these values for both the program body as well as any invariants to be enforced. By doing so, we bypass repeated, possibly expensive, compilations of the code.

5.2 Deployment Models

We note that several scenarios for ZigZag deployment are possible. First, application developers or providers could perform instrumentation on-site, protecting all users of the application against CSV vulnerabilities. Since no prior knowledge is necessary in order to apply ZigZag to an application, this approach is feasible even for third parties. And, in this case there is no overhead incurred due to re-instrumentation on each resource load.

On the other hand, it is also possible to deploy ZigZag as a proxy. In this scenario, network administrators could transparently protect their users by rewriting all web applications at the network gateway. Or, individual users could tunnel their web traffic through a personal proxy, while sharing generated invariants within a trusted crowd.

5.3 Limitations

ZigZag's goal is to defend against attackers that desire to achieve code execution within an origin, or act on behalf of the victim. The system was not designed to be stealthy or protect its own integrity if an attacker manages to gain JavaScript code execution in the same origin. If attackers were able to perform arbitrary JavaScript commands, any kind of in-program defense would be futile without support from the browser.

Therefore, we presume (as discussed in Section 2.1) the presence of complementary measures to defend against XSS-based code injection. Examples of such techniques that could be applied today include Content Security Policy (CSP), or any of the number of template auto-sanitization frameworks that prevent code injection in web applications [17, 18, 6].

Another important limitation to keep in mind is that anomaly detection relies on a benign training set of sufficient size to represent the range of runtime behaviors that could occur. If the training set contains attacks, the resulting invariants might be prone to false negatives. We believe that access to, or the ability to generate, benign training data is a reasonable assumption in most cases. For instance, traces could be generated from end-to-end tests used during application development, or might be collected during early beta testing using a population of well-behaving users. However, in absence of absolute ground truth, solutions to sanitize training data exist. For instance, Cretu et al. present an approach that can sanitize polluted training data sets [12].

If the training set is too small, false positives could occur. To limit the impact of undertraining, we only generate invariants for functions if we have more than four sessions, which we found to be sufficient for the test cases we evaluated. We note that the training threshold is configurable, however, and can easily be increased if greater variability is observed at invariant checkpoints. Undertraining, however, is not a limitation specific to ZigZag, but rather a limitation of anomaly detection in general.

With respect to templated JavaScript, while ZigZag can detect templates of previously observed programs by generalizing, entirely new program code can not be enforced without previous training.

In cases where multiple users share programs instrumented by ZigZag, users might have legitimate privacy concerns with respect to sensitive data leaking into invariants generated for enforcement. This can be addressed in large part by avoiding use of the oneOf invariant, or by heuristically detecting whether an invariant applies to data that originates from password fields or other sensitive input and selectively disabling the oneOf invariant. Alternatively, oneOf invariants could be hashed to avoid leaking user data in the enforcement code.

6 Evaluation

To evaluate ZigZag, we implemented a prototype of the approach using the proxy deployment scenario. We wrote Squid [19] ICAP modules to interpose on HTTP(S) traffic, and modified the Google Closure compiler [16] to instrument JavaScript code.

```
// Dispatches received messages to appropriate function
if (e.data.action == ’markasread’) {
    markEmailAsRead(e.data);
}

// Communication with the server to mark emails as read
function markEmailAsRead(data) {
    var xhr = new XMLHttpRequest();
    xhr.open(’POST’, serverurl, true);
    xhr.send(’markasread=’ + data.markemail);
}

// Communication with the ad network iframe
function sendAds(e) {
    adWindow.postMessage({
        ’topic’: ’ads’,
        ’action’: ’showads’,
        ’content’: ’{JSON␣string}’
    }, ”*”);
}
```

Figure 7: Vulnerable webmail component.

```
// Receive JSON object from webmail component
function showAds(data) {
    var received = eval(’(’ + data.content + ’)’);
    // Work with JSON object...
}
```

Figure 8: Vulnerable ad network component.

Our evaluation first investigates the security benefits that ZigZag can be expected to provide to potentially vulnerable JavaScript-based web applications. Second, we evaluate ZigZag's suitability for real-world deployment by measuring its performance overhead over microbenchmarks and real applications.

6.1 Synthetic Applications

Webmail service. We evaluated ZigZag on the hypothetical webmail system first introduced in Section 2. This application is composed of three components, each isolated in iframes with different origins that contain multiple vulnerabilities. These iframes communicate with each other using `postMessage` on `window.top.frames`.

We simulate a situation in which an attacker is able to control one of the iframes, and wants to inject malicious code into the other origins or steal personal information. The source code snippets are described in Figures 7 and 8.

From the source code listings, it is evident that the webmail component is vulnerable to parameter injection through the `markemail` property. For instance, injecting the value `1&deleteuser=1` could allow an attacker to delete a victim's profile. Also, the ad network uses an `eval` construct for JSON deserialization. While highly discouraged, this technique is still commonly used in the wild and can be trivially exploited by sending code instead of a JSON object.

We first used the vulnerable application through the ZigZag proxy in a learning phase consisting of 30 sessions over the course of half an hour. From this, ZigZag extracted statistically likely invariants from the resulting execution traces. ZigZag then entered the enforcement phase. Using the site in a benign fashion, we verified that no invariants were violated in normal usage.

For the webmail component, and specifically the function handling the XMLHttpRequest, ZigZag generated the following invariants.

1. The function is only called by one parent function
2. `v0.topic === 'control'`
3. `v0.action === 'markasread'`
4. `typeof(v0.markemail) === 'number' && v0.markemail >= 0`
5. `typeof(v0.topic) === typeof(v0.action) && v0.topic < v0.action`

For the ad network, ZigZag generated the following invariants.

1. The function is only called by one parent function
2. `v0.topic === 'ads'`
3. `v0.action === 'showads'`
4. `v0.content is JSON`
5. `v0.content` is printable
6. `typeof(v0.topic) === typeof(v0.action) && v0.topic < v0.action`
7. `typeof(v0.topic) === typeof(v0.content) && v0.topic < v0.content`
8. `typeof(v0.action) === typeof(v0.content) && v0.action < v0.content`

Next, we attempted to exploit the webmail component by injecting malicious parameters into the `markemail` property. This attack generated an invariant violation since the injected parameter was not a number greater than or equal to zero.

Finally, we attempted to exploit the vulnerable ad network component by sending JavaScript code instead of a JSON object to the `eval` sink. However, this also generated an invariant violation, since ZigZag learned that `data.content` should always be a JSON object – i.e., it should not contain executable code.

URL fragments. Before `postMessage` became a standard for cross-origin communication in the browser, URL fragments were used as a workaround. The URL fragment portion of a URL starts after a hash sign. A distinct difference between URL fragments and the rest of the URL is that changes to the fragment will not trigger a reload of the document. Furthermore, while SOP generally denies iframes of different origin mutual access to resources, the document location can nevertheless be accessed. The combination of these two properties allows for a channel of communication between iframes of different origins.

We evaluated ZigZag on a demo program that communicates via URL fragments. The program expects as input an email address and uses it without proper sanitization in `document.write`. Another iframe could send unexpected data to be written to the DOM. The code is described in Figure 9.

```
function getFragment ( ) {
    return window.location.hash.substring(1);
}

function fetchEmailAddress() {
    var email = getFragment();
    document.write("Welcome␣" + email);
    // ...
}
```

Figure 9: Vulnerable fragment handling.

After the training phase, we generated the following invariants for the `getFragment` function.

1. The function is only called by one parent function
2. The return value is an email address
3. The return value is printable

6.2 Real-World Case Studies

In our next experiment, we tested ZigZag on four real-world applications that contained different types of vulnerabilities. These vulnerabilities are a combination of previously documented bugs as well as newly discovered vulnerabilities.[1]

These applications are representative of different, previously-identified classes of CSV vulnerabilities. In particular, Son et al. [9] examined the prevalence of CSV vulnerabilities in the Alexa Top 10K websites, found 84 examples, and classified them. The aim of this experiment is to demonstrate that the invariants ZigZag generates can prevent exploitation of these known classes of vulnerabilities.

For each of the following case studies, we first trained ZigZag by manually browsing the application with one user for five minutes, starting with a fresh browser state four times. Next, we switched ZigZag to the enforcement phase and attempted to exploit the applications. We consider the test successful if the attacks are detected with no false alarms. In each case, we list the relevant invariants responsible for attack prevention.

Janrain. A code snippet used by `janrain.com` for user management is vulnerable to a CSV attack. The application checks the format of the string, but does not check the origin of messages. Therefore, by iframing the site, an attacker can execute arbitrary code if the message has a specific format, such as `capture:x;alert(3):`. This is due to the fact that the function that acts as a message receiver will, under certain conditions, call a handler that evaluates part of the untrusted message string

[1] For each vulnerability we discovered, we notified the respective website owners.

as code. Both functions were identified as important by ZigZag's lightweight static analysis. We note that this vulnerability was previously reported in the literature [9]. As of writing, ten out of the 13 listed sites remain vulnerable, including `wholefoodsmarket.com` and `ladygaga.com`.

For the event handler, ZigZag generated the following invariants.

1. The function is only invoked from the global scope or as an event handler
2. `typeof(v0) === 'object' && v0.origin === 'https://dpsg.janraincapture.com'`
3. `v0.data === 's1' || v0.data === 's2'`[2]
4. `v0.data` is printable

For the function that is called by the event handler, ZigZag generated the following invariants.

1. The function is only called by the receiver function
2. `v0 === 's1' || v0 === 's2'`[3]

The attack is thwarted by restricting the receiver origin, only allowing two types of messages to be received, and furthermore restricting control-flow to the dangerous sink.

playforex.ru. This application contains an incorrect origin check that only tests whether the message origin *contains* the expected origin (using `indexOf`), not whether the origin equals or is a subdomain of the allowed origin. Therefore, any origin containing the string "playforex.ru" such as "playforex.ru.attacker.com" would be able to iframe the site and evaluate arbitrary code in that context. We reported the bug and it was promptly fixed. However, this is not an isolated case. Related work [9] has shown that such a flawed origin check was used by 71 hosts in the top 10,000 websites.

ZigZag generated the following relevant invariants.

1. The function is only invoked from the global scope or as an event handler
2. `typeof(v0) === 'object'`
   `&& v0.origin === 'http://playforex.ru'`
3. `v0.data === "$('#right_buttons').hide();"`
   `|| v0.data === 'calculator()'`

ZigZag detected that the `onMessage` event handler only receives two types of messages, which manipulate the UI to hide buttons or show a calculator. By only accepting these two types of messages, arbitrary execution can be prevented.

Yves Rocher. This application does not perform an origin check on received messages, and all received code is executed in an `eval` sink. The bug has been reported to the website owners. 43 out of the top 10,000 websites had previously been shown to be exploitable with the same technique. ZigZag generated the following relevant invariant.

1. `v0.origin === 'http://static.ak.facebook.com' || v0.origin === 'https://s-static.ak.facebook.com'`

From our manual analysis, this program snippet is only intended to communicate with Facebook, and therefore the learned invariant above is correct in the sense that it prevents exploitation while preserving intended functionality.

adition.com. This application is part of a European ad network. It used a `new Function` statement to parse untrusted JSON data, which is highly discouraged as it is equivalent to an `eval`. In addition, no origin check is performed. This vulnerability allows attackers that are able to send messages in the context of the site to replace ads without having full JavaScript execution.

ZigZag learned that only valid JSON data is received by the function, which would prevent the attack based on the content of received messages. This is different than the Yves Rocher example, as data could be transferred from different origins while still securing the site. The bug was reported and fixed.

Summary. These are four attacks against CSV vulnerabilities representative of the wider population. `postMessage` receivers are used on 2,245 hosts out of the top 10,000 websites. Such code is often included through third-party libraries that can be changed without the knowledge of website owners.

6.3 Performance Overhead

Instrumentation via a proxy incurs performance overhead in terms of latency in displaying the website in the browser. We quantify this overhead in a series of experiments to evaluate the time required for instrumentation, the worst-case runtime overhead due to instrumentation, and the increase in page load latency for real web applications incurred by the entire system.

Instrumentation overhead. We tested the instrumentation time of standalone files to measure ZigZag's impact on load times. As samples, we selected a range of popular JavaScript programs and libraries: Mozilla pdf.js, an in-browser pdf renderer; jQuery, a popular client-side scripting library; and, d3.js, a library for data visualization. Where available, we used compressed, production versions of the libraries. As Mozilla pdf.js is not minified by default, we applied the yui compressor for simple minification before instrumenting.

[2] s1 and s2 were long strings, which we omitted for brevity.

[3] s1 and s2 were long strings, which we omitted for brevity.

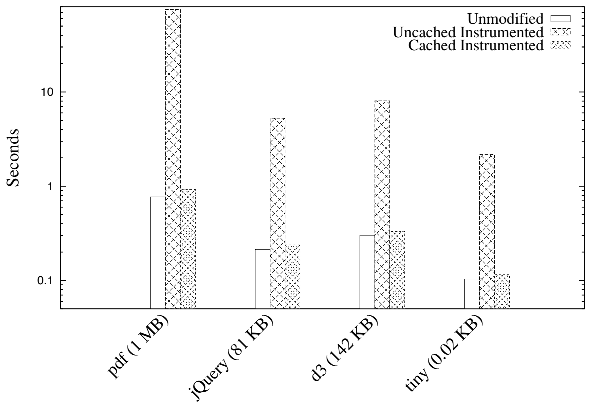


Figure 10: Instrumentation overhead for individual files. While the initial instrumentation can take a significant amount of time for large files, subsequent instrumentations have close to no overhead.

The worker file is at 1.5 MB uncompressed and represents an atypically large file. Additionally, we instrumented a simple function that returns the value of `document.cookie`. We performed 10 runs for cold and warm testing each. For cold runs, the database was reset after every run.

Figure 10 shows that while the initial instrumentation can be time-consuming for larger files, subsequent calls will incur low overhead.

Microbenchmark. To measure small-scale runtime enforcement overhead, we created a microbenchmark consisting of a repeated `postMessage` invocation where one iframe (A) sends a message to another iframe (B), and B responds to A. Specifically, A sends a message object containing a property `process` set to the constant 20. B calculates the Fibonacci number for `process`, and responds with another object that contains the result.

We trained ZigZag on this simple program and then enabled enforcement mode. Next, we ran the program in both uninstrumented and instrumented forms. The subject of measurement was the elapsed time between sending a message from A to B and reception of the response from B to A. We used the high resolution timer API `window.performance.now` to measure the round trip time, and ran the test 100 times each. The results of this benchmark are shown in Table 2.

ZigZag learned and enforced the following invariants for the receiving side.

1. The function is only invoked from the global scope or as an event handler
2. `typeof(v0) === 'object' &&`
   `v0.origin === 'http://example.com'`
3. `v0.data.process === 20`
4. `typeof(v0) === typeof(v0.data)`
5. `typeof(v0.timeStamp) === typeof(v0.data.process) && v0.timeStamp > v0.data.process`

	Uninstrumented	Instrumented
Average Runtime	3.11 ms	3.77 ms
Standard Deviation	1.80	0.54
Confidence (0.05)	0.11	0.35

Table 2: Microbenchmark overhead.

For the message receiver that calculates the response, ZigZag learned and enforced the following invariants.

1. The function is only invoked from the global scope or as an event handler
2. `typeof(v0) === 'object'`
   `&& v0.origin === 'http://example.com'`
3. `typeof(v0.data.process) === 'number'`
   `&& v0.data.process === 20`
4. `typeof(v0.timestamp) === typeof(v0.data.process)`

Finally, for the receiver of the response, ZigZag learned and enforced the following invariants.

1. The function is only invoked from the global scope or as an event handler
2. `typeof(v0) === 'object' &&`
   `v0.origin === 'http://example.com'`
3. `v0.data.response === 6765`
4. `typeof(v0) === typeof(v0.data)`
5. `typeof(v0.timeStamp) === typeof(v0.data.response) && v0.timeStamp > v0.data.response`

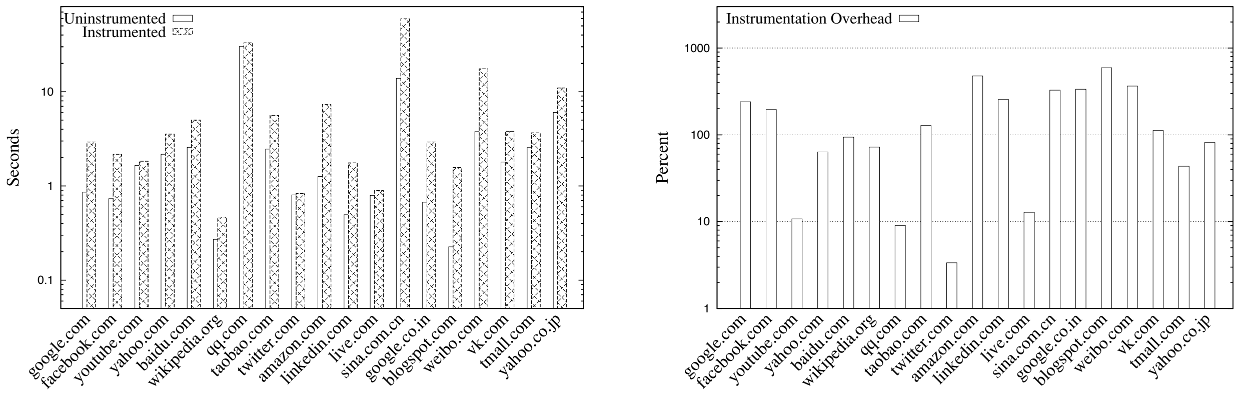
The above invariants represent a tight bound on the allowable data types and values sent across between each origin.

End-to-end benchmark. To quantify ZigZag's impact on the end-to-end user experience, we measured page load times on the Alexa Top 20. First, we manually inspected the usability of the sites and established a training set for enforcement mode. To do so, we browsed the target websites for half an hour each.

We used Chrome to load the site and measure the elapsed time from the initial request to the `window.load` event, when the DOM completed loading (including all sub-frames).[4] The browser was unmodified, with only one extension to display page load time.

Uninstrumented sites are loaded through the same HTTP(S) proxy ZigZag resides on, but the program text

[4] We note, however, that websites can become usable before that event fires.



(a) Absolute load times for uninstrumented and instrumented programs.

(b) Overhead due to instrumentation.

Figure 11: End-to-end performance benchmark on the Alexa 20 most popular websites (excluding hao123.com as it is incompatible with our prototype). A site is considered to be done loading content when the `window.load` event is fired, indicating that the entire contents of the DOM has finished loading.

is not modified. Instrumented programs are loaded from a ZigZag cache that has been previously filled with instrumented code and merge descriptions. However, we do not cache original web content, which is freshly loaded every time.

The performance overhead in absolute and relative terms is depicted in Figure 11. We excluded `hao123.com` from the measurement as it was incompatible with our prototype.[5] On average, load times took 4.8 seconds, representing an overhead of 180.16%, with median values of 2.01 seconds and an overhead of 112.10%. We found server-side templated JavaScript to be popular with the top-ranked websites. In particular, `amazon.com` served 15 such templates, and only 6 out of 19 serve no such templates.

`sina.com.cn` is an obvious outlier, with an absolute average overhead of 45 seconds. With 115 inlined JavaScript snippets and 112 referenced JavaScript files, this is also the strongest user of inline script. Furthermore, we noticed that the site fires the `DOMContentLoaded` event in less than 6 seconds. Hence, the website appears to become usable quickly even though not all sub-resources have finished loading.

In percentages, the highest overhead of 593.36% is introduced for `blogspot.com`, which forwards to Google. This site has the shortest uninstrumented loading time (0.226 seconds) in our data set, hence an absolute overhead will have the strongest implications on relative overhead. That is, in relative numbers, it seems higher than the actual impact on end-users.

We note that we measure the load event, which means that all elements (including ads) have been loaded. Websites typically become usable before that event is fired. Our research prototype could be further optimized to reduce the impact of our technique for performance-critical web applications, for example by porting our ICAP Python code, including parsing libraries, to an ECAP C module. However, generally speaking we believe that trading off some performance for improved security would be acceptable for high assurance web applications and security-conscious users.

6.4 Program Generalization

As discussed in Section 3, ZigZag supports structural similarity matching and invariant patching for templated JavaScript to avoid singleton training sets and excessive instrumentation when templated code is used. We measured the prevalence of templated JavaScript in the Alexa Top 50, and found 185 instances of such code. In addition, the median value per site was three. Without generalization and invariant patching, ZigZag would not have generated useful invariants and, furthermore, would perform significantly worse due to unnecessary re-instrumentation on template instantiations.

6.5 Compatibility

To check that ZigZag is compatible with real web applications, we ran ZigZag on several complex, benign JavaScript applications. Since ZigZag relies on user in-

[5] We discovered, as others have before, that `hao123.com` does not interact well with Squid. We attempted to work around the problem by adjusting Squid's configuration as suggested by Internet forum posts, but this did not succeed. Due to time constraints, we did not expend further effort in dealing with this particular site.

teraction and the functionality of a complex web application is not easily quantifiable, we added manual quantitative testing to augment automated tests. The testers were familiar with the websites before using the instrumented version, and we performed live instrumentation using the proxy-based prototype.

For YouTube and Vimeo, the testers browsed the sites and watched multiple videos, including pausing, resuming, and restarting at different positions. Facebook was tested by scrolling through several timelines and using the chat functionality in a group setting. The testers also posted to a timeline and deleted posts. For Google Docs, the testers created and edited a document, closed it, and re-opened it. For d3.js, the testers opened several of the example visualizations and verified that they ran correctly. Finally, the testers sent and received emails with Gmail and live.com.

In all cases, no enforcement violations were detected when running the instrumented version of these web applications.

7 Related Work

In this section, we discuss ZigZag in the context of related work.

Client-side validation vulnerabilities. CSV vulnerabilities were first highlighted by Saxena et al. [3]. In their work, the authors propose FLAX, a framework for CSV vulnerability discovery that combines dynamic taint analysis and fuzzing into taint-enhanced blackbox fuzzing. The system operates in two steps. JavaScript programs are first translated into a simplified intermediate language called JASIL. Then, the JavaScript application under test is executed to dynamically identify all data flows from untrusted sources to critical sinks such as cookie writes, `eval`, or `XMLHttpRequest` invocations. This flow information is processed into small executable programs called acceptor slices. These programs accept the same inputs as the original program but are reduced in size. Second, the acceptor slices are fuzzed using an input-aware technique to find inputs to the original program that can be used to exploit a bug. A program is considered to be vulnerable when a data flow from an untrusted source to a critical sink can be established.

Later, the same authors improved FLAX by replacing the dynamic taint analysis component with a dynamic symbolic execution framework [4]. Again, the goal of the static analysis is to find unchecked data flows from inputs to critical sinks. This method provides no completeness and can hence miss vulnerabilities.

The main difference between ZigZag and FLAX is that FLAX focuses on detecting vulnerabilities in applications, while ZigZag is intended to defend unknown vulnerabilities against attacks.

DOM-based XSS. Cross-site scripting (XSS) is often classified as either stored, reflected, or DOM-based XSS [20]. In this last type of XSS, attacks can be performed entirely on the client-side such that no malicious data is ever sent to the server. Programs become vulnerable to such attacks through unsafe handling of DOM properties that are not controlled by the server; examples include URL fragments or the referrer.

As a defense, browser manufacturers employ client-side filtering, where the state-of-the-art is represented by the Chrome XSS Auditor. However, the auditor has shortcomings in regards to DOM-based XSS. Stock et al. [21] have demonstrated filter evasion with a 73% success rate and proposed a filter with runtime taint tracking.

DexterJS [22] rewrites insecure string interpolation in JavaScript programs into safe equivalents to prevent DOM-based XSS. The system executes programs with dynamic taint analysis to identify vulnerable program points and verifies them by generating exploits. DexterJS then infers benign DOM templates to create patches that can mitigate such exploits.

JavaScript code instrumentation. Proxy-based instrumentation frameworks have been proposed before [23, 14]. JavaScript can be considered as self-modifying code since a running program can generate input code for its own execution. This renders complete instrumentation prior to execution impossible since writes to code cannot be covered. Hence, programs must be instrumented before execution and all subsequent writes to program code must be processed by separate instrumentation steps.

Anomaly detection. Anomaly detection has found wide application in security research. For instance, Daikon [13] is a system that can infer likely invariants. The system applies machine learning to make observations at runtime. Daikon supports multiple programming languages, but can also be used over arbitrary data as CSV files. In ZigZag, we extended Daikon with new invariants specific to JavaScript applications for runtime enforcement.

DIDUCE [24] is a tool that instruments Java bytecode and builds hypotheses during execution. When violations to these hypotheses occur, they can either be relaxed or raise an alert. The program can be used to help in tracking down bugs in programs semi-automatically.

ClearView [25] uses a modified version of DAIKON to create patches for high-availability binaries based on learned invariants. The focus of the system is to detect and prevent memory corruption through changing the program code at runtime. However, the embedded monitors do not extend to detecting errors in program logic.

Attacks on the workflow of PHP applications have been addressed by Swaddler [10]. Not all attacks on systems produce requests or, more generally, external behavior that can be detected as anomalous. These attacks can be detected by instrumenting the execution environment and generating models that are representative of benign runs. Swaddler can be operated in three modes: training, detection, and prevention. To model program execution, profiles for each basic block are generated, using univariate and multivariate models. During training, probability values are assigned to each profile by storing the most anomalous score for benign data, a level of "normality" is established. In detection and prevention mode, an anomaly score is calculated based on the probability of the execution data being normal using a preset threshold. Violations are assumed to be attacks. The results suggest that anomaly detection on internal application state allows a finer level of attack detection than exclusively analyzing external behavior.

While Swaddler focuses on the server component of web applications, ZigZag characterizes client-side behavior. ZigZag can protect against cross-domain attacks within browsers that Swaddler has no visibility into. Swaddler invokes detection for every basic block, while we use a dynamic level of granularity based on the types of sinks in the program, resulting in a dramatic reduction in enforcement overhead.

Client-side policy enforcement. ICESHIELD [26] is a policy enforcement tool for rules based on manual analysis. By adding JavaScript code before all other content, ICESHIELD is invoked by the browser before other code is executed. Through ECMAScript 5 features, DOM properties are frozen to maintain the integrity of the detection code. ICESHIELD protects users from drive-by downloads and exploit websites. In contrast, ZigZag performs online invariant detection and prevents previously unknown attacks.

ConScript [27] allows developers to create fine-grained security policies that specify the actions a script is allowed to perform and what data it is allowed to access or modify. Conscript can generate rules from static analysis performed on the server as well as by inspecting dynamic behavior on the client. However, it requires modifications to the JavaScript engine, which ZigZag aims to avoid.

The dynamic nature of JavaScript renders a purely static approach infeasible. Chugh et al. propose a staged approach [28] where they perform an initial analysis of the program given a list of disallowed flow policies, and then add residual policy enforcement code to program points that dynamically load code. The analysis of dynamically loaded code can be performed at runtime. These policies can enforce integrity and confidentiality properties, where policies are a list of tuples of disallowed flows (from, to).

Content Security Policy (CSP) [29, 11] is a framework for restricting JavaScript execution directly in the browser. CSP can be effective at preventing significant classes of code injection in web applications if applied correctly (e.g., without the use of `unsafe-inline` and `unsafe-eval`) and if appropriate rules are enforced. However, CSP does not defend against general CSV attacks, and therefore we view it and other systems with similar goals as complementary to ZigZag. In particular, CSP could be highly useful to prevent code injection and thereby protect the integrity of ZigZag in the browser.

Web standards. Although Barth et al. [30] made the HTML5 postMessage API more secure, analysis of websites suggests that it is nevertheless used in an insecure manner. Authentication weaknesses of popular websites have been discussed by Son et al. [9]. They showed that 84 of the top 10,000 websites were vulnerable to CSV attacks, and moreover these sites often employ broken origin authentication or no authentication at all. Their proposed defenses rely on modifying either the websites or the browser.

In ZigZag, we aim for a fine-grained, automated, annotation-free approach that dynamically secures applications against unknown CSV attacks in an unmodified browser.

8 Conclusion

Most websites rely on JavaScript to improve the user experience on the web. With new HTML5 communication primitives such as `postMessage`, inter-application communication in the browser is possible. However, these new APIs are not subject to the same origin policy and, through software bugs such as broken or missing input validation, applications can be vulnerable to attacks against these client-side validation (CSV) vulnerabilities. As these attacks occur on the client, server-side security measures are ineffective in detecting and preventing them.

In this paper, we present ZigZag, an approach to automatically defend benign-but-buggy JavaScript applications against CSV attacks. Our method leverages dynamic analysis and anomaly detection techniques to learn and enforce statistically-likely, security-relevant invariants. Based on these invariants, ZigZag generates assertions that are enforced at runtime. ZigZag's design inherently protects against unknown vulnerabilities as it enforces learned, benign behavior. Runtime enforcement is carried out only on the client-side code, and does not require modifications to the browser.

ZigZag can be deployed by either the website operator or a third party. Website owners can secure their JavaScript applications by replacing their programs with a version hardened by ZigZag, thereby protecting all users of the application. Third parties, on the other hand, can deploy ZigZag using a proxy that automatically hardens any website visited using it. This usage model of ZigZag protects all users of the proxy, regardless of the web application.

We evaluated ZigZag using a number of real-world web applications, including complex examples such as online word processors and video portals. Our evaluation shows that ZigZag can successfully instrument complex applications and prevent attacks while not impairing the functionality of the tested web applications. Furthermore, it does not incur an unreasonable performance overhead and, thus, is suitable for real-world usage.

Acknowledgements

This work was supported by the Office of Naval Research (ONR) under grant N00014-12-1-0165, the Army Research Office (ARO) under grant W911NF-09-1-0553, the Department of Homeland Security (DHS) under grant 2009-ST-061-CI0001, the National Science Foundation (NSF) under grant CNS-1408632, and SBA Research. We would like to thank the anonymous reviewers for their helpful comments. Finally, we would like to thank the Marshall Plan Foundation for partially supporting this work.

References

[1] Internet World Stats, "Usage and Population Statistics," http://www.internetworldstats.com/stats.htm, 2013.

[2] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper)," in *IEEE Symposium on Security and Privacy (Oakland)*, 2006.

[3] P. Saxena, S. Hanna, P. Poosankam, and D. Song, "FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications," in *ISOC Network and Distributed System Security Symposium (NDSS)*, 2010.

[4] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A Symbolic Execution Framework for JavaScript," in *IEEE Symposium on Security and Privacy (Oakland)*, 2010.

[5] D. Crockford, "JSLint: The JavaScript Code Quality Tool," April 2011, http://www.jslint.com/.

[6] M. Samuel, P. Saxena, and D. Song, "Context-sensitive Auto-sanitization in Web Templating Languages using Type Qualifiers," in *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[7] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay, "Safe Active Content in Sanitized JavaScript," Google, Inc., Tech. Rep., 2008.

[8] S. Maffeis and A. Taly, "Language-based Isolation of Untrusted JavaScript," in *IEEE Computer Security Foundations Symposium*, 2009.

[9] S. Son and V. Shmatikov, "The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites," in *ISOC Network and Distributed System Security Symposium (NDSS)*, 2013.

[10] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna, "Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications," in *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2007.

[11] "Content Security Policy 1.1," 2013. [Online]. Available: https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html

[12] G. F. Cretu, A. Stavrou, M. E. Locasto, S. J. Stolfo, and A. D. Keromytis, "Casting out Demons: Sanitizing Training Data for Anomaly Sensors," in *IEEE Symposium on Security and Privacy (Oakland)*, 2008.

[13] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon System for Dynamic Detection of Likely Invariants," *Science of Computer Programming*, 2007.

[14] H. Kikuchi, D. Yu, A. Chander, H. Inamura, and I. Serikov, "JavaScript Instrumentation in Practice," in *Asian Symposium on Programming Languages and Systems (APLAS)*, 2008.

[15] F. Groeneveld, A. Mesbah, and A. van Deursen, "Automatic Invariant Detection in Dynamic Web Applications," Delft University of Technology, Tech. Rep., 2010.

[16] "Closure Compiler," 2013. [Online]. Available: https://developers.google.com/closure/compiler

[17] "ctemplate - Powerful but simple template language for C++," 2013. [Online]. Available: https://code.google.com/p/ctemplate/

[18] "Handlebars.js: Minimal Templating on Steroids," 2007. [Online]. Available: http://handlebarsjs.com/

[19] "Squid Internet Object Cache," http://www.squid-cache.org, 2005.

[20] A. Klein, "DOM Based Cross Site Scripting or XSS of the Third Kind," *Web Application Security Consortium, Articles*, 2005.

[21] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns, "Precise Client-side Protection against DOM-based Cross-Site Scripting," *USENIX Security Symposium*, 2014.

[22] I. Parameshwaran, E. Budianto, S. Shinde, H. Dang, A. Sadhu, and P. Saxena, "Auto-Patching DOM-based XSS At Scale," *Foundations of Software Engineering (FSE)*, 2015.

[23] D. Yu, A. Chander, N. Islam, and I. Serikov, "JavaScript Instrumentation for Browser Security," in *Principles of Programming Languages (POPL)*, 2007.

[24] S. Hangal and M. S. Lam, "Tracking Down Software Bugs Using Automatic Anomaly Detection," in *International Conference on Software Engineering (ICSE)*, 2002.

[25] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan *et al.*, "Automatically Patching Errors in Deployed Software," in *ACM Symposium on Operating Systems Principles (SIGOPS)*, 2009.

[26] M. Heiderich, T. Frosch, and T. Holz, "ICESHIELD: Detection and Mitigation of Malicious Websites with a Frozen DOM," in *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2011.

[27] L. A. Meyerovich and B. Livshits, "Conscript: Specifying and Enforcing Fine-grained Security Policies for JavaScript in the Browser," in *IEEE Symposium on Security and Privacy (Oakland)*, 2010.

[28] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner, "Staged Information Flow for JavaScript," in *ACM Sigplan Notices*, 2009.

[29] S. Stamm, B. Sterne, and G. Markham, "Reining in the Web with Content Security Policy," in *International Conference on World Wide Web (WWW)*, 2010.

[30] A. Barth, C. Jackson, and J. C. Mitchell, "Securing Frame Communication in Browsers," *Communications of the ACM*, 2009.

Anatomization and Protection of Mobile Apps' Location Privacy Threats

Kassem Fawaz, Huan Feng, and Kang G. Shin
The University of Michigan
{*kmfawaz, huanfeng, kgshin*}*@umich.edu*

Abstract

Mobile users are becoming increasingly aware of the privacy threats resulting from apps' access of their location. Few of the solutions proposed thus far to mitigate these threats have been deployed as they require either app or platform modifications. Mobile operating systems (OSes) also provide users with location access controls. In this paper, we analyze the efficacy of these controls in combating the location-privacy threats. For this analysis, we conducted the first location measurement campaign of its kind, analyzing more than 1000 free apps from Google Play and collecting detailed usage of location by more than 400 location-aware apps and 70 Advertisement and Analytics (A&A) libraries from more than 100 participants over a period ranging from 1 week to 1 year. Surprisingly, 70% of the apps and the A&A libraries pose considerable profiling threats even when they sporadically access the user's location. Existing OS controls are found ineffective and inefficient in mitigating these threats, thus calling for a finer-grained location access control. To meet this need, we propose `LP-Doctor`, a light-weight user-level tool that allows Android users to effectively utilize the OS's location access controls while maintaining the required app's functionality as our user-study (with 227 participants) shows.

1 Introduction

Mobile users are increasingly aware of the privacy threats caused by apps' access of their location [12, 42]. According to recent studies [14, 17, 42], users are also taking measures against these threats ranging from changing the way they run apps to disabling location services all together on their mobile devices. How to mitigate location-privacy threats has also been researched for some time. Researchers have proposed and even implemented location-privacy protection mechanisms (LPPMs) for mobile devices [2, 6, 12, 20, 30]. However, few of them have been deployed as they require app or system-level modifications, both of which are unappealing/unrealistic to the ordinary users.

Faced with location-privacy threats, users are left only with whatever controls the apps and OSes provide. Some, but not all, apps allow the users to control their location access. OSes have been improving on this front. iOS includes a new permission to authorize location access in the background, or when the app is not actively used. Also, iOS, Windows OS, and Blackberry (Android to follow suit) utilize per-app location-access permissions. The user authorizes location access at the very first time an app accesses his location and has the option to change this decision for every subsequent app invocation. We want to answer two important questions related to this: *(i) are these controls effective in protecting the user's location privacy and (ii) if not, how can they be improved at the user level without modifying any app or the underlying OS?*

To answer these questions, we must understand the location-privacy threats posed by mobile apps. This consists of understanding the apps' location-access patterns and their usage patterns. For this, we instrumented and analyzed the top 1165 downloaded free apps (that require location-access permissions) from Google Play to study their location-access patterns. We also studied the behavior of Advertisement and Analytics (A&A) libraries, such as Flurry, embedded in the apps that might access location. We analyzed only those apps/libraries that access location through Android's official location APIs. While some apps/libraries might circumvent the OS in accessing location, it is an orthogonal problem to that addressed in this paper.

We then analyzed the users' app-usage patterns by utilizing three independent datasets. First, we collected and analyzed app-tagged location traces through a 10-month data collection campaign (Jan. 2013—Nov. 2013) for 24 Android smartphone users. Second, we recruited 95 Android users through PhoneLab [31], a smartphone measurement testbed at New York State University at Buffalo, for 4 months. Finally, we utilized the dataset from Livelab at Rice University [34] that contains app-usage and location traces for 34 iPhone users for over a year.

Utimately, we were able to evaluate the privacy threats posed by 425 apps and 77 third-party libraries. 70% of the apps are found to have the potential of posing profiling threats that have not yet been adequately studied or addressed before [15, 16, 25, 41]. Moreover, the A&A libraries pose significant profiling threats on more than 80% of the users as they aggregate location information from multiple apps. Most of the users are unaware of these threats as they can't keep track of exposure of their location information. The issue becomes more problematic in the case of A&A libraries where users are oblivious to which apps these libraries are packed in and whether they are receiving location updates.

Given the nature of the threats, we studied the effectiveness of the existing OS controls. We found that these controls are capable of thwarting only a fraction of the underlying privacy threats, especially tracking threats. As for profiling, the user only has the options of either blocking or allowing location access. These two options come at either of the two extremes of the privacy–utility spectrum: the user either enjoys full privacy with no utility, or full utility with no privacy. As for A&A libraries, location accesses from a majority of the apps must be blocked to thwart the location-privacy threats caused by these libraries.

The main problem arises from the user's inability to exercise fine-grained control on when an app should receive a location update. The interface provided by existing controls makes it hard for the user to enforce location-access control on a per visited place/session basis. Even if the user can dynamically change the control of location access, he cannot estimate the privacy threats at runtime. The location-privacy threat is a function of the current location along with previously released locations. This makes it difficult to estimate the threat for apps and even harder for A&A libraries.

To fill this gap, we propose `LP-Doctor`, a user-level app, to protect the location privacy of smartphone users, which offers three salient features. First, `LP-Doctor` evaluates the privacy threat that the app might pose before launching it. If launching the app from the current location poses a threat, then it acts to protect the user's privacy. It also warns the user of the potential threat in a non-intrusive manner. Second, `LP-Doctor` is a *user-level* app and does not require any modification to the underlying OS or other apps. It acts as a control knob for the underlying OS tools. Third, `LP-Doctor` lets the user control, for each app, the privacy–utility tradeoff by adjusting the protection level while running the app.

We implemented `LP-Doctor` as an Android app that can be downloaded from Google Play. The privacy protection that `LP-Doctor` provides comes at a minimal performance overhead. We recruited 227 participants through Amazon Mechanical Turk and asked them to download and use `LP-Doctor` from Google Play. The overwhelming majority of the participants reported little effect on the quality of service and user experience. More than 77% of the participants indicated that they would install `LP-Doctor` to protect their location privacy.

In summary, we make the following main contributions:

- The first location data collection campaign of its kind to measure, analyze, and model location-privacy threats from the apps' perspectives (Sections 3–6);
- Evaluation of the effectiveness of OS's location privacy controls by anatomizing the location-privacy threats posed by the apps (Sections 7–8);
- Design, implementation and evaluation of a novel user-level app, `LP-Doctor`, based on our analysis to fill the gaps in existing controls and improve their effectiveness (Section 9).

2 Related Work

App-Based Studies: To the best of our knowledge, this is the first attempt to quantify and model location privacy from the apps' perspective. Researchers already concluded that many mobile apps and A&A libraries leak location information about the users to the cloud [5, 23, 38]. These efforts are complimentary to ours; we study the quantity and quality of location information that the apps and libraries locally gather while assuming that they may leak this information outside the device.

Analysis of Location Privacy: Influenced by existing location datasets (vehicular traces, cellular traces, etc.), most of the existing studies view location privacy in smartphones as if there were only one app *continuously* accessing a user's location [7, 11, 25, 26, 29, 33, 41]. Researchers also proposed mechanisms [28, 29, 32] (their effectiveness analyzed by Shokri *et al.* [36]) to protect against the resulting tracking-privacy threats. Such mechanisms have shown to be ineffective in thwarting the profiling threats [41] which are more prevalent as we will show later.

Researchers started considering sporadic location-access patterns as a source of location-privacy threat that calls for a different treatment than the continuous case [4]. Still, existing studies focus mostly on the tracking threat [3, 35]. The only exception to this is the work by Freudiger *et al.* [15]. They assessed the erosion of the user's privacy from sporadic location accesses as the portion of the PoIs identified after downsampling the continuous location trace. In this paper, we propose a formal metric to model the profiling threats. Also, we show that an app's location-access behavior can't be modeled as simply downsampling the user's mobility.

Location-Privacy Protection Proposals: Several solutions have been proposed to protect mobile users' location privacy. MockDroid [6] allows for blocking apps' location access to protect the user's location privacy. LP-Guardian [12] is another system aiming at protecting the user's location privacy by incorporating a myriad of mechanisms. Both systems require platform modifications, hindering their deployment. Other mechanisms, such as Caché [2] and the work by Micinski *et al.* [30], provide apps with coarsened locations but require modifications to the apps. Koi [20] proposed a location privacy enhancing system that utilizes a cloud service, but requires developers to use a different API to access location. Apps on Google Play such as PlaceMask and Fake GPS Location Spoofer rely on the user to manually feed apps with fake locations, which reduce their usability.

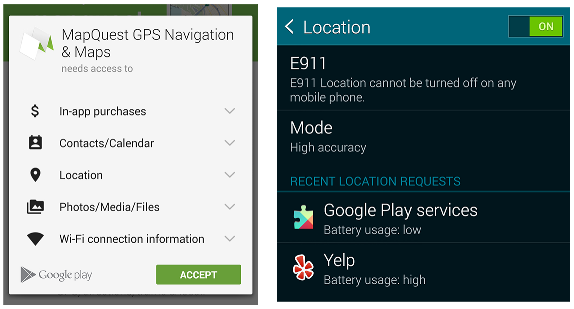
Finally, researchers have proposed improved permission models for Android [1, 24]. In their models, the users are aware of how much the apps access their location and have the choice to enable/disable location access for each app (AppOps provided such functionality in Android 4.3). LP-Doctor improves on these in three ways. First, it provides a privacy model that maps each app's location access to a privacy metric. This model includes more information than just the number of location accesses by the app. Second, LP-Doctor makes some decisions on behalf of the users to avoid interrupting their tasks and to make privacy protection more usable. Third, LP-Doctor employs *per-session* location-access granularity which achieves a better privacy–utility tradeoff.

3 Background and Data Collection

To study the efficacy of location-access controls of different mobile OSes, we had to first analyze location-privacy threats from the apps' perspectives. This includes studying how different apps collect the user's location. We conduct a data collection campaign to achieve this using the Android platform. Our results, however, can be generalized to other mobile platforms like iOS.

3.1 Location-Access Controls

Each mobile platform provides users with a set of location-access controls to mitigate possible location-privacy threats. Android (prior to Android M) provides a one-time permission model that allows users to authorize location access. Once the user approves the permission list (Fig. 1–left) for the app, it is installed and the permissions can't be revoked. It also provides a global location knob (Fig. 1–right) to control location services. The user can't exercise per-app location-access control.

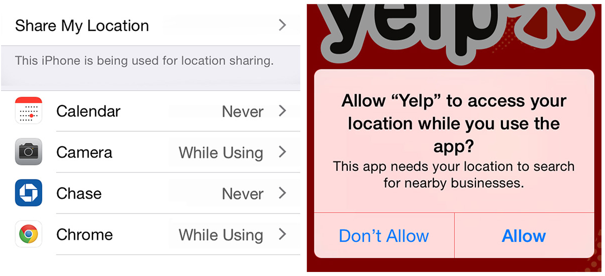

MapQuest GPS Navigation & Maps
needs access to

$ In-app purchases
Contacts/Calendar
Location
Photos/Media/Files
Wi-Fi connection information

Google play
ACCEPT

Location ON
E911
E911 Location cannot be turned off on any mobile phone.
Mode
High accuracy
RECENT LOCATION REQUESTS
Google Play services
Battery usage: low
Yelp
Battery usage: high

Figure 1: Android's permission list (left) and location settings (right).


Share My Location
This iPhone is being used for location sharing.

Calendar	Never
Camera	While Using
Chase	Never
Chrome	While Using

Allow “Yelp” to access your location while you use the app?
This app needs your location to search for nearby businesses.
Don't Allow
Allow

Figure 2: iOS's location settings (left) and prompts (right).

Other platforms, such as Blackberry OS and iOS, provide finer-grained location permissions. Each app has a settings menu (Fig. 2–left) that indicates the resources it is allowed to access, including location. The user can at any point of time revoke resource access by any app. The first time an app accesses location, the OS prompts the user to authorize location access for the app in the current and future sessions (Fig. 2–right). Also, Google, starting from Android M, will provide a similar permission model (an evolution of the previously deployed AppOps in Android 4.3) to control access of location and other resources. At present, iOS provides the users with an additional option to authorize location access in the background to prevent apps from tracking users.

In the rest of this paper, we study the following controls: (1) one-time location permissions, (2) authorization of location access in the background, and (3) finer-grained per-app permissions.

3.2 System Model

We study location-privacy threats through apps and A&A libraries that access the user's location. These apps and libraries then provide the service, and keep the location records indexed by a user ID, such as MAC address, Android ID, IMEI, etc.

We assume that the app/library communicates all of the user's location samples to the service provider.[1] This allows us to model the location-privacy threats caused by apps/libraries in the worst-case scenario. The app is the only means by which the service provider can collect the user's location updates. We don't consider cases where the service provider obtains the user's location via side channels other than the official API, e.g., an app reads the nearby access points and sends them to a localization service, such as skyhook.

We preclude system and browsing apps from our study for the following reasons. System apps are part of the OS that already has access to the user's location all the time. Hence, analyzing their privacy implications isn't very informative. As for the browsing apps, the location sink might be a visited website as well as the browser itself. We decided not to monitor the user's web history during the data collection for privacy concerns. Also, app-usage patterns differ from browsing patterns. The conclusions derived for the former don't necessarily translate to those for the latter.

3.3 App and A&A libraries Analysis

In February 2014, we downloaded the top 100 apps of each of Google Play's 27 app categories. We were left with 2588 unique apps, of which 1165 apps request location permissions. We then instrumented Android to intercept every location access invoked by both the app and the packed A&A libraries.

The main goal of this analysis was to unravel the situations in which an app accesses location and whether it feeds a packed A&A library. In Android, the app could be running in the foreground, cached in the background, or as a service. Using a real device, we ran every app in foreground, moved it to background, and checked if it forked a service, while recording its location requests.

Apps running in the foreground can access location spontaneously or in response to some UI event. So, we ran every app in two modes. In the first mode, the app runs for a predefined period of time and then closes, while in the second, we manually interact with each app to trigger the location-based functionality. Finally, we analyzed the functionality of every app and the required location granularity to achieve this functionality.

3.4 Data Collection

As will be evident in Section 4, the app-usage pattern is instrumental in determining the underlying location-privacy threats. We collected the app-usage data using an app that we developed and published on Google Play. Our study was deemed as not-requiring an IRB oversight by the IRB at our institution; all the data we collected is anonymous. Also, we clustered the participants' location on the device to extract their visited places. We define the "place" as a center location with a radius of 50m and a minimum visit time of 5 min. Then, we logged place IDs instead of absolute location samples to further protect the participants' privacy.

PhoneLab: PhoneLab [31] is a testbed, deployed at the NY State University at Buffalo, composed of 288 smartphone users. PhoneLab aims to free the researchers from recruiting participants by providing a diverse set of participants, which leads to stronger conclusions.

We recruited 95 participants to download and run our app for the period between February 2014 and June 2014. We collected detailed usage information for 625 apps, of which 218 had location permissions and were also part of the apps inspected in the app-analysis stage.

Our Institution: The second set consists of 24 participants whom we recruited through personal relations and class announcements. We launched this study from January 2013 till November 2013, with the participation period per user varying between 1 week and 10 months. From this set, we collected usage data of 256 location-aware apps.

We also collected location access patterns of some apps from a subset of the participants. We handed 11 participants Galaxy Nexus devices with an instrumented Android (4.1.2) that recorded app-tagged location accesses. We measured how frequently do ordinary users invoke location-based functionality of apps that don't spontaneously access location (e.g., Whatsapp).

LiveLab: Finally, we utilize the Livelab dataset [34] from Rice University. This dataset contains the app usage and mobility records for 34 iPhone users over the course of a year (2010). We post-processed this dataset to map app-usage records to the location where the apps were invoked. We only considered those apps that overlapped with our Android dataset (35 apps).

4 Location-Access Patterns

We address the location-access patterns by analyzing how different apps collect location information while running in foreground and background. The former represents the state where the user actively interacts with the app, while the latter represents the case where the app runs in the background either as cached by Android or as a persistent service.

As evident from Table 1, 74% of the apps solely access location when running in the foreground, while only 3% continuously access the user's location in the back-

[1] We refer to both the app developers and A&A agencies as the service provider.

Table 1: Location-access patterns for smartphone apps according to Android location permissions

	Fore. (%)	Cached (%)	Back. (%)	None (%)	Gran. Coarse (%)
Coarse	71	6	1	22	100
Fine	74	14	4	12	48
All	74	12	3	14	66

Table 2: Location-access patterns for A&A libraries

Total	No Location Access	App Feeds Location	Auto Location Access		
			Coarse	Fine	Both
77	22	17	3	2	33

ground. Around 70% of the apps accessing location in the foreground spontaneously perform such access preceding any user interaction. Examples of these apps include Angry Birds, Yelp, Airbnb, etc.

Android caches the app when the user exits it; depending on the app's behavior it might still access location; only 12% of the apps access the user's location when they are cached. Interestingly, for 14% of the apps, we didn't find any evidence that they access location in any state.

We also analyzed the location-based functionality of every app and the required location granularity to achieve such functionality. We focused on two location granularity levels: *fine* and *coarse*. A fine location sample is one with block-level granularity or higher, while coarse location is that with zipcode-level granularity or lower. We manually interacted with each app to assess the change in its functionality while feeding it locations with different granularity. We show the percentage of the apps that can accommodate coarse location without noticeable loss of app functionality in Table 1 under the column titled *Gran. Coarse*. One can notice that apps abuse the location permissions: 48% of the apps requesting fine location permissions can accommodate locations with coarser granularity without loss of functionality.

Finally, we analyzed the packed A&A libraries in these apps. We were able to identify 77 of such libraries packed in these apps. Table 2 shows basic statistics about these libraries. Most (more than 70%) libraries require location access where some are fed location from the apps (22%). The rest of the libraries automatically access location where 3 of them require coarse location permissions, 2 require fine permissions, and the rest don't specify a permission. Also, these libraries are included within more than one location-aware app giving them the ability to track the user's location beyond what a single app can do. For example, of 1165 analyzed apps, Google Ads is packed within 499 apps, Flurry within 325 apps, Medialets within 35 apps, etc.

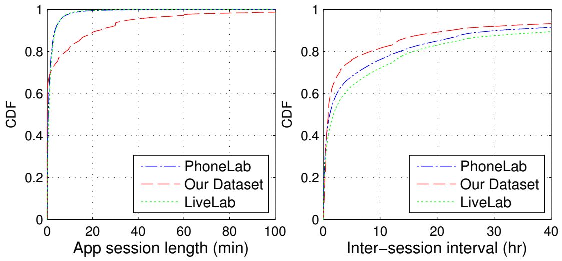


Figure 3: The distribution of app session lengths (left) and inter-session intervals (right) for the three datasets.

5 App-Usage Patterns

As apps mostly access users' location in the foreground, the app-usage patterns (the way that users invoke different apps) help determine how much location information each app collects. Apps are shown to sporadically sample the user's location based on two facts. First, an app session is equivalent to the place visited during the session. Second, apps' inter-session intervals follow a Pareto-law distribution.

For foreground apps, we define a session as a single app invocation—the period of time in which a user runs the app then exits it. The session lengths are not long enough to cover more than one place the user visits, where 80% of these app sessions are shorter than 10 minutes (the left plot of Fig. 3). We confirmed this from our PhoneLab dataset; 98% of the app sessions started and ended at the same place.

This allows for collapsing an app session into one location-access event. It doesn't matter what frequency the app polls the user's location with. As long as the app requests the user's location at least once, while it is running in the foreground, it will infer that the user visited that location. We thus ignore the location-access frequency of foreground apps, and instead focus on the app-usage patterns.

We define the inter-session time as the interval separating different invocations (sessions) of the same app by the same user. The right plot of Fig. 3 shows the distribution of the inter-session intervals for the three datasets. More than 50% of the app sessions were separated by at least one hour.

We also found that the inter-session intervals follow a Pareto-law distribution rather than a uniform distribution. This indicates that apps don't sample the user's location uniformly, indicating that existing models for apps' location access don't match their actual behavior.

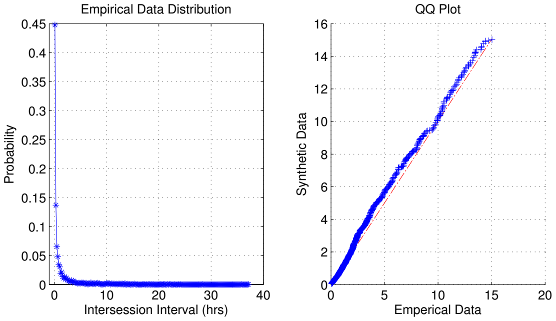


Figure 4: The distribution of the inter-session times for Facebook in Livelab dataset (left), and the QQ plot of this distribution versus a Pareto law distribution (right).

Fig. 4 shows the distribution of the inter-session intervals of a user running Facebook. It is evident that the distribution of the inter-session intervals decays linearly with respect to the increase of inter-session intervals. We observed a similar trend with all other apps. This suggests that the data decays according to a Pareto law (QQ plot in Fig. 4). We followed the guidelines outlined by Clauset *et al.* [10] to fit the data to the truncated Pareto distribution. Three parameters (L, H, and α) define the truncated Pareto law distribution:

$$p_X(x) \begin{cases} \frac{(-\alpha-1)L^{-\alpha-1}x^{\alpha}}{1-\left(\frac{L}{H}\right)^{-\alpha-1}} & \text{if } L \leq x \leq H \\ 0 & \text{otherwise.} \end{cases}$$

After fitting the data, more than 97% of the app-usage models are found to have α between -1 and -1.5. According to Vagna *et al.* [40], Pareto law fits different human activity models with α between -1 and -2.

6 Privacy Model

Here we model the privacy threats caused by mobile apps'/libraries' access of the user's location.

6.1 Preliminaries

Below we describe the models of user mobility, app-usage, and adversaries that we will use throughout the paper.

User Mobility Model: We assume there is a region (e.g., city) of interest which includes set of places that the user can visit. So, a domain of interest is represented by the set Pl of all the places available in that domain: $Pl = \{pl_1, pl_2, pl_3 \ldots\}$. Under this model, the user visits a set of places, $U_{Pl} \subseteq Pl$, as part of his daily life, spends time at pl_i running some apps and then moves to another place pl_j. We alternatively refer to these places as the user's *Points of Interest* (PoIs).

We associate every place pl_i with a visit probability of p_i, reflecting the portion of time the user spends at pl_i. The user's *mobility profiles* are defined as the set, U_{pl}, of places he visited and the probability, p_i, of visit to each place. The mobility profile is unique to each user since a different user visits a different set of places with a different probability distribution [41].

App-Usage Model: In Section 5, we showed that each app session is equivalent to an observation of the user's visit to a place. The app accumulates observations of the set of places that the user visits. The app will eventually observe that a user visited a certain place pl_i for c_{pl_i} times. So, we view the app as a random process that samples the user's entire location trace and outputs a histogram of places of dimension $|U_{Pl}|$. Each bin in the histogram is the number of times, c_{pl_i}, the app observes the user at that specific place. The total number of visits is represented as $N = \sum_{i=1}^{|U_{Pl}|} c_{pl_i}$.

The histogram represents the app's view of the user's mobility. Most apps don't continuously monitor user's mobility as they don't access location in the background. As such, they can't track users; the most these apps can get from a user is the histogram of the places he visited, which constitutes the source of location-privacy threats in this case.

Adversary Model: The adversary in our model is not necessarily a malicious entity seeking to steal the user's private information. It is rather a curious entity with possession of the user's location trace. The adversary will process and analyze these traces to infer more information about the user that allows for a more personalized service. This is referred to as *authentic apps* [39]. The objective of our analysis is to study the effect of the ordinary apps collecting location on the user's privacy.

Apps accessing location in the foreground can't track the user (Section 8). So, the adversary seeks to profile the user based on locations he visited. We use the term *profiling* to represent the process of inferring more information about the user through the collected location data. The profiling can take place at multiple levels, ranging from identifying preferences all the way to revealing the user's identity. Instead of modeling the adversary's profiling methods/attacks, we quantify the amount of information that location data provides the adversary with. The intuition behind our analysis of the profiling threat is that the more descriptive the app's histogram of the actual user's mobility pattern, the higher the threat is.

6.2 Privacy Metrics

Table 3 summarizes the set of metrics that we utilize to quantify the privacy threats that each app poses from its location access. The simplest metric is the fraction of the users' PoIs the app can identify [15]. We evaluate this

Table 3: The metrics used for evaluating the location privacy threats.

Metric	Description
PoI_{total}	Fraction of the user's PoIs
PoI_{part}	Fraction of the user's infrequently visited PoIs
$Prof_{cont}$	Distance between the user's histogram and mobility profile
$Prof_{bin}$	χ^2 test of the user's histogram fitting the mobility profile

metric by looking at the apps' actual collected location traces, rather than a downsampled location trace. We will henceforth refer to this metric as PoI_{total}.

We also consider a variant of the metric (referred to as PoI_{part}) as the portion of the sensitive PoIs that the apps might identify. We define the sensitive PoIs as those that have a very low probability of being visited. These PoIs will exhibit abnormalities in the user's behavior. Research results in psychology [19, 21] indicated that people regard deviant (abnormal) behavior as being more private and sensitive. Places that an individual might visit that are not part of his regular patterns might leak a lot of information and are thus more sensitive in nature.

The histogram, as we mentioned before, is a sample of the user's mobility pattern. The second aspect of the profiling is quantifying how descriptive of the user's mobility pattern (original distribution) the app's histogram (sample) is.

For the purpose of our analysis and the privacy-preserving tool we propose later, we need two types of metrics. The first is a *continuous* metric, $Prof_{cont}$, that quantifies the profiling threat as the distance between the original distribution (mobility profile) and the sample (app's histogram). The second is a *binary* metric, $Prof_{bin}$, that indicates whether a threat exists or not.

For $Prof_{cont}$, we use the KL-divergence [27] as a measure of the difference (in bits) between the histogram (H) and the user's mobility pattern. The K-L divergence is given by $D_{\mathrm{KL}}(H\|p) = \sum_{i=1}^{|U_{Pl}|} H(i) \ln \frac{H(i)}{p_i}$, where $H(i)$ is the probability of the user visiting place pl_i based on the histogram, while p_i is the probability of the user visiting that place based on his mobility profile. The lower (higher) the value of $Prof_{cont}$, the higher (lower) the threat will be since the distance between the histogram and mobility pattern will be smaller (larger).

$Prof_{cont}$ is not useful in identifying histograms that pose privacy threats. There is no intuitive way by which a threshold can separate values that pose threats and those not posing any threat. So, we need a criterion indicating whether or not a threat exists based on the app's histogram. We use Pearson's Chi-square goodness of fit test to meet this need. This test indicates if the observed sample differs from the original (theoretical) distribution. Specifically, it checks if the null hypothesis of the sample originating from an original distribution can be accepted or not.

The test statistic, in our context, is $\chi^2 = \sum_{i=1}^{|U_{Pl}|} \frac{(c_{pl_i} - E_i)^2}{E_i}$ where $E_i = N.p_i$ is the expected number of visits to the place pl_i. The statistic converges to a Chi-squared distribution with $|U_{Pl}| - 1$ degrees of freedom when the null hypothesis holds. The test yields a p-value which if smaller than the significance level (α) then the null hypothesis can be rejected ($Prof_{bin} = 0$—no threat), else $Prof_{bin} = 1$, where null hypothesis can't be rejected, indicating the existence of a threat. In Sections 7 and 8, we employ the widely-used value of 0.05 as the significance level.

A&A libraries: can aggregate location information from the different apps in which they are packed and allowed to access location. We can thus view the histogram pertaining to an A&A library as the aggregate of the histograms of the apps in which the library is packed. We evaluate the same metrics for the aggregated histogram.

For the case of PoI_{total} and PoI_{part} metric, the aggregate histogram will be representative of the threat posed by the libraries. As for $Prof_{cont}$ and $Prof_{bin}$, we consider the aggregate histogram as well as the individual apps' histograms. The threat per library is the highest of that of the aggregate and individual histograms. The privacy threat posed by the library is at least as bad as that of any app that packs it in.

7 Anatomy

We now present the major findings from our measurement campaign. We analyze the location trace of each app and user, and hence, every data point in the subsequent plots belongs to an app–user combination. We constructed each app's histogram by overlaying its location-access pattern on its usage data for every user.

Privacy Threat Distribution: Fig. 5 shows the distributions of PoI_{total}, PoI_{part}, and $Prof_{cont}$ for both the apps and A&A libraries. As to PoI_{total}, most of the apps can identify at least 10% of the user's PoIs; while for 20% of the app–user combinations, apps were able to identify most of the user's PoIs. Apps can't identify all of the user's PoIs for two reasons: (1) not all apps access the user's location every time, as highlighted in Section 4, and (2) users don't run their apps from every place they visit. On the other hand, A&A libraries can identify more of the user's PoIs, with most of the libraries identifying at least 20% of the user's PoIs. Moreover, as the middle plots of Fig. 5 indicate, around 30% of the apps were able to identify some of the user's sensitive (less frequently visited) PoIs. More importantly, A&A libraries were able to identify more of the user's sensitive PoIs,

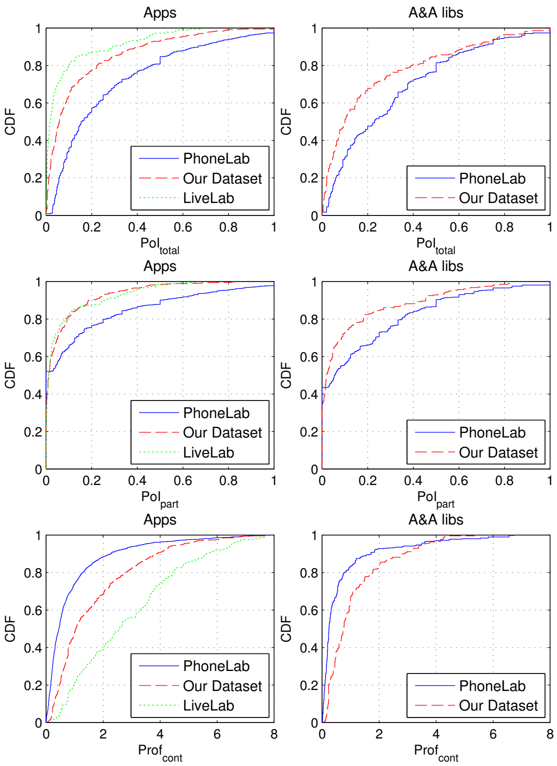


Figure 5: The distributions of PoI_{total} (top), PoI_{part} (middle), and $Prof_{cont}$ (bottom) for the apps (left) and A&A libraries (right) from our datasets.

indicating the level of privacy threats they pose.

The two bottom plots of Fig. 5 show the distributions of the profiling metric $Prof_{cont}$ for the foreground apps in the three datasets. The lower the value of the metric, the higher the privacy threat is. There are two takeaways from these two plots. First, apps do pose significant privacy threats; the distance between the apps' histogram and the user's mobility pattern is less than 1 bit in 40% of the app–user combinations for the three datasets. The second observation has to do with the threat posed by A&A libraries. It is clear from the comparison of the left and right plots that these libraries pose considerably higher threats. In more than 80% of user–library combinations, the distance between the observed histograms and the user's mobility profile is less than 1 bit.

Apps tend to even pose higher identification threats. As evident from Fig. 5, some apps can identify a relatively minor portion of the user's mobility which might not be sufficient to fully profile the user. Nevertheless, the portion of PoIs tend to be those users frequently visit (e.g., home and work) which may suffice to identify them [18, 25, 41]. This might not be a serious issue for those apps, such as Facebook, that can learn the user's home and work from other methods. Other apps

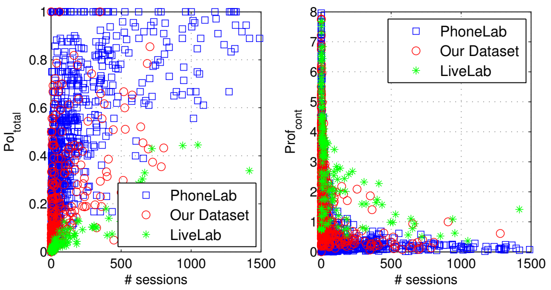


Figure 6: The distribution of PoI_{total} (left) and $Prof_{cont}$ (right) vs. the number of app sessions.

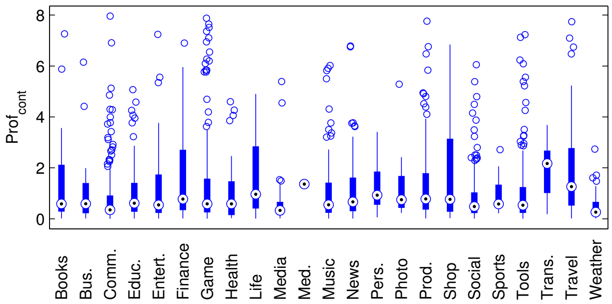


Figure 7: The distribution of $Prof_{cont}$ vs. app categories.

and libraries (e.g., Angry Birds), however, might infer the user's identity even when he anonymously uses them (without providing an identity or login information).

Fig. 5 also confirms our intuition in studying the location traces from the apps' perspective. If apps were to uniformly sample the user's mobility as has been assumed in literature, $Prof_{cont}$ should be mostly close to 0 (indicating no difference between the histogram and the mobility pattern), which is not the case.

Privacy Threats and App-Usage: We also evaluated the posed privacy threats vs. the app-usage rate as shown in Fig. 6. As evident from the plots, there is little correlation between the amount of posed threats and the app-usage rate. Apps that are used more frequently, do not necessarily pose higher threats, as user mobility, the app's location-access pattern, and the user's app-usage pattern affect the privacy threat.

With lower usage rates, both PoI_{total} and $Prof_{cont}$ vary significantly. Users with little diversity in their mobility pattern are likely to visit the same places more frequently. Even the same user could invoke apps differently; he uses some apps mostly at unfamiliar places (navigation apps), while using other apps more ubiquitously (gaming apps), thus enabling the apps to identify more of his PoIs.

Finally, we studied the distribution of the threat in relation to app categories. Fig. 7 shows that the threat level is fairly distributed across different app categories and

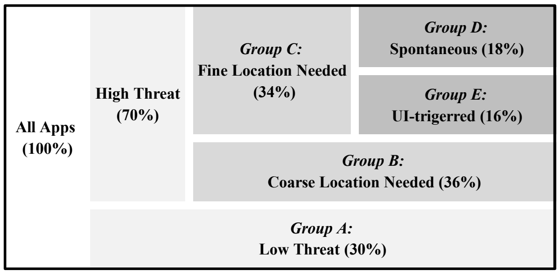

All Apps (100%)
High Threat (70%)
Group C: Fine Location Needed (34%)
Group D: Spontaneous (18%)
Group E: UI-trigerred (16%)
Group B: Coarse Location Needed (36%)
Group A: Low Threat (30%)

Figure 8: App categorization according to threat levels, location requirements, and location-access patterns.

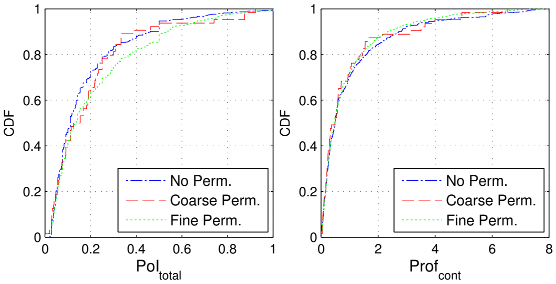


Figure 9: The distribution of PoI_{total} (left) and $Prof_{cont}$ (right) for PhoneLab apps with different permissions.

the same category. This confirms, again, that privacy threats result from multiple sources and are a function of both apps and users. Some app categories, however, pose lower threats on average. For example, transportation apps (including navigation apps) pose lower threats as users tend to use from unfamiliar places.

Threat Layout: Given the three datasets, we were able to analyze the profiling threats as posed by 425 location-aware apps (Fig. 8). For this part, we use $Prof_{bin}$ metric to decide which apps pose privacy threats and those which don't. As apps pose different threats depending on the users, we counted an app as posing a threat if it poses a privacy threat to at least one user. Only a minority of the apps (30%) pose negligible threats.

The rest of the apps pose a varying degree of profiling threat. We analyzed their functionality: 52% of such apps don't require location with high granularity to provide location-based functionality. For these apps, a zipcode- or city-level granularity would be more than enough (weather apps, games). This leaves us with 34% of the apps that require block-level or higher location granularity to provide usable functionality. These apps either spontaneously access location (18%) or in response to a UI event (16%).

8 OS Controls

Having presented an anatomy for the location-privacy threats posed by mobile apps, we are now ready to evaluate the effectiveness of existing OSes' location access controls in thwarting these threats.

Global Location Permissions: Android's location permissions attempt to serve two purposes: notification and control. They notify the user that the app he is about to install can access his location. Also, permissions aim to control the granularity by which apps access location. Apps with coarse-grained location permission can only access location with both low granularity and frequency.

Fig. 9 compares the profiling threats (PoI_{total} and $Prof_{cont}$) posed by apps with fine location permissions and those with coarse location permissions. It also plots the distribution of the privacy metrics for apps without location permissions assuming that they accessed location when running. While this might seems oblivious at a first glance, we aim to compare the location-based usage of apps with different location permissions. This allows us to study if the location permissions are effective as a notification mechanism so that users use apps from different places depending on the location permissions.

The apps with fine-grained location permissions exhibited very similar usage pattern to those apps without location access. The users ran the app from the same places regardless of whether they have location permissions or not. We conclude that this notification mechanism does little to alert users on potential privacy threats and has no effect on the app-usage behavior. Similar observations have also been made by others [17].

Almost a half of the apps (Table 1) that request fine-grained location permissions are found to be able to achieve the location-based functionality with coarser-granularity location. This suggests that apps abuse location permissions. If used appropriately, permissions can be effective in thwarting the threats resulting from apps' abuse of location access (∼40% of the apps — Group B — according to Fig. 8).

Background Location Access: Background location access is critical when it comes to tracking individuals. It enables comprehensive access to the user's mobility information including PoIs and frequent routes. Recently, iOS 8 introduced a new location permission that allows users to authorize location access in the background for apps on their devices.

This permission strikes a balance between privacy and QoS. We showed in Section 4 that apps rarely access location in the background. Thus, this option affects a very low portion of the user's apps, but is effective in terms of privacy protection, especially in thwarting tracking threats. We evaluated the tracking threat in terms of tracking time per day [12, 22] for the three datasets for foreground location access.

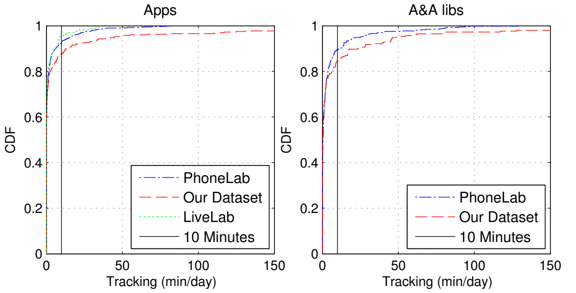


Figure 10: The distribution of the tracking threat posed by the foreground apps (left) and A&A libraries (right).

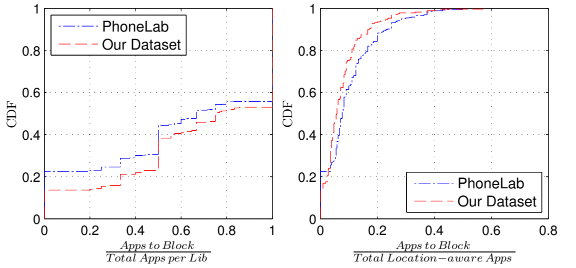


Figure 11: The fraction of the user's apps that must be blocked from accessing location to protect against privacy threats posed by A&A libraries.

Fig. 10 (left) shows that in 90% of the app–user combinations, blocking background location access will limit the location exposure to less than 10 minutes a day (from foreground location access). The third-party libraries tend to pose slightly higher tracking threats than apps (Fig. 10 – right).

Per-app Location Permissions: To improve over static permissions, iOS enables the user to allow/disallow location access on a per-app basis. The users gain two advantages from this model: (i) location access can be blocked for a subset, but not all, of the apps, and (ii) the apps retain some functionality even when the location access is blocked.

Even if the user trusts an app with location access, the app can still profile him through the places he visited (Groups D and E in Fig. 8). To combat these threats, the user has to either allow location access to fully exploit the app and lose privacy, or gain his privacy while losing the location-based app functionality. Currently, mobile platforms offer no middle ground to balance privacy and QoS requirements.

In Section 7, we showed that A&A libraries pose significant threats that users are completely unaware of as they access location from more than one app. The user can't identify which apps he must disallow to access location in order to mitigate threats from third-party libraries. Fig. 11 shows the portion of the user's apps that must be disbarred from accessing location to thwart threats from packed A&A libraries. It turns out (left plot of Fig. 11) that in order to protect the user from privacy threats posed by a single library, at least 50–70% of the apps carrying the library must be disbarred from accessing location. This amounts to blocking location for more than 10% of the apps installed on the device.

In conclusion, a static permission model suffers serious limitations, blocking location access in the background is effective in mitigating the tracking threat but not the profiling one, and per-app controls exhibit an unbalanced tradeoff between privacy and QoS. Also, they are ineffective against the threats caused by A&A libraries. Thus, a finer-grained location access control is required, allowing control for each app session depending on the context. Per-session location access control allows users to leverage better and more space in the privacy–QoS spectrum.

9 LP-Doctor

Users can't utilize the existing controls to achieve per-session location-access controls for two reasons. First, these controls are coarse-grained (providing only per-app controls at best). For finer-level controls, the user has to manually modify the location settings before launching each app, which is quite cumbersome and annoying. Second, even if the user can easily change these settings, making an informed decision is a different story. Therefore, we propose `LP-Doctor` that helps users utilize the existing OS controls to provide location-access control on a per-session basis.

9.1 Design

`LP-Doctor` trusts the underlying OS and its associated apps; it targets user-level apps accessing location while running in the foreground, as we found that most apps don't access location in the background. `LP-Doctor` focuses on the apps with fine location permissions as they could pose higher threats. `LP-Doctor` automatically coarsens location for apps requesting coarse location permissions to ensure a commensurate privacy-protection level.

The main operation of `LP-Doctor` consists of two parts. The first involves the user-transparent operations described below, while the second includes the interactions with the user described in Section 9.2.

We bundled `LP-Doctor` with CyanogenMod's app launcher.[2] It runs as a background service, intercepts app-launch events, decides on the appropriate actions, performs these actions, and then instructs the app to launch. Fig. 12 shows the high-level execution flow of

[2] Source code: `https://github.com/kmfawaz/LP-Doctor`.

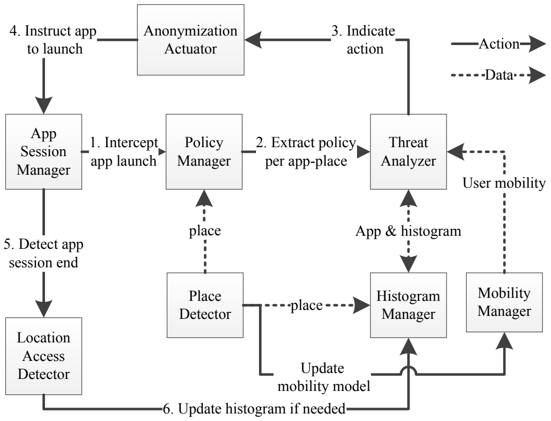


Figure 12: The execution flow of LP-Doctor when a location-aware app launches.

LP-Doctor. Next, we elaborate on LP-Doctor's components and their interactions.

App Session Manager: is responsible for monitoring app launch and exit events. LP-Doctor needs to intercept app-launch events to anonymize location.

Fortunately, Android (recently iOS as well) allow for developing custom app launchers. Users can download and install these launchers from the app store which will, in turn, be responsible for listening to the user's events and executing the apps. We instrumented CyanogenMod's app launcher (available as open source and under Apache 2 license) to intercept app launch events.

Particularly, before the app launcher instructs the app to execute, we stop the execution, save the state, and send an intent to LP-Doctor's background service (step 1 in Fig. 12). LP-Doctor takes a set of actions and sends an intent to the app launcher, signaling the app can launch (steps 2 and 3 in Fig. 12). The app launcher then restores the saved state and proceeds with execution of the app (step 4 in Fig. 12). In Section 9.4, we will report the additional delay incurred by this operation.

In the background, LP-Doctor frequently polls (once every 10s) the current foreground app to detect if the app is still running. For this purpose, it uses getRecentTasks on older versions of Android and AppUsageStats class for Android L. When an app is no longer running in the foreground, LP-Doctor executes a set of maintenance operations to be described later (steps 5 and 6 in Fig. 12).

Policy Manager: fetches the privacy policy for the currently visited place and the launched app as shown in Fig. 13.

At installation time, the user specifies a privacy policy to be applied for the app. We call this the *per-app policy* which specifies three possible actions: *block*, *allow*, and *protect*. If the per-app policy indicates privacy protection, LP-Doctor asks the user to specify a per-place policy for the app. The per-place policy indicates the policy that LP-Doctor must follow when the app launches from a particular place. The policy manager passes the app's policy and the current place to the threat analyzer.

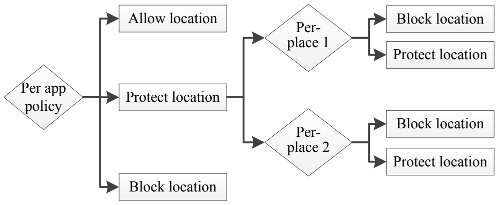


Figure 13: The policy hierarchy of LP-Doctor.

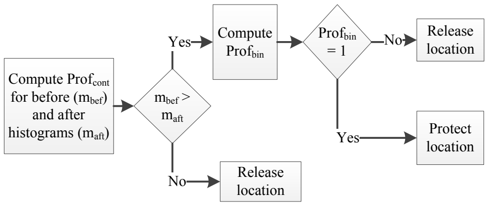


Figure 14: The threat analyzer's decision diagram.

Place Detector & Mobility Manager: The place detector monitors the user's actual location, and applies online clustering to extract the spatio-temporal clusters which represent places that the users visit. Whenever the user changes the place he is visiting, the place detector module instructs the mobility manager to update the mobility profile of the user as defined in Section 6.

Histogram Manager: maintains the histogram of the places visited as observed by each app. It stores the histograms in an SQLite table that contains the mapping of each app–place combination to a number of observations. The threat analyzer module consults the histogram manager to obtain two histograms whenever an app is about to launch. The first is the current histogram of the app (based on previous app events) which we refer to as the "before" histogram. While the second one is the potential histogram if the app were to access location from the currently visiting place; we call this histogram as the "after" one.

Threat Analyzer: decides on the course of action regarding apps associated with a *protect* policy. It basically performs the decision diagram depicted in Fig. 14 to decide whether to release the location or add noise.

The threat analyzer determines whether the "after" histogram leaks more information than the old one through computing $Prof_{cont}$ for each histogram. If $Prof_{cont}$ increases LP-Doctor decides to release the location to the app. On the other hand, if $Prof_{cont}$ decreases, LP-Doctor uses $Prof_{bin}$ to decide if the "after" histogram fits the user's mobility pattern and whether to release or anonymize location.

$Prof_{bin}$ depends on the significance level, α, as we specified in Section 6. In LP-Doctor, α is a function of the privacy level chosen by the user. LP-Doctor recognizes three privacy levels: low, medium, and high. Low privacy corresponds to $\alpha = 0.1$; medium privacy corresponds to $\alpha = 0.05$; and high privacy protection corresponds to the most conservative $\alpha = 0.01$.

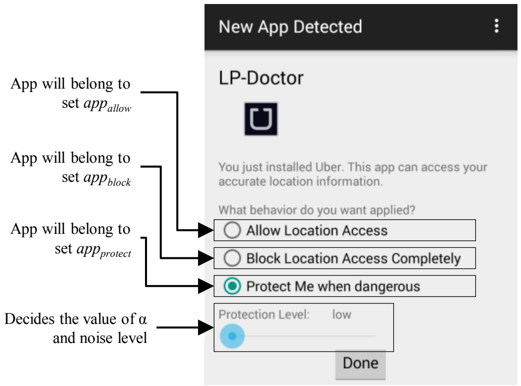
The procedure depicted in Fig. 14 won't hide places that the user seldom visits but are sensitive to him. The per-place policies allow the user to set a privacy policy for each visited place, effectively allowing him to control the places he wants revealed to the service providers. Also, LP-Doctor can be extended to support other privacy criteria that try to achieve optimal privacy by perturbing location data [9,37].

Anonymization Actuator: receives an action to perform from the threat analyzer. If the action is to protect the current location, the actuator computes a fake location by adding Laplacian noise [3] to ensure location indistinguishability. The privacy level determines the amount of noise to be added on top of the current location. One the other hand, if the action is to block, the actuator computes the fake location of $< 0,0 >$.

As specified by Andrés *et al.* [3], repetitive engagement of Laplacian noise mechanism at the same location leaks information about the location. To counter this threat, LP-Doctor computes the anonymized location once per location and protection-level combination, and saves it. When the user visits the same location again, LP-Doctor employs the same anonymized location that was previously computed to prevent LP-Doctor from recomputing a fake location for the same place.

After computing/fetching the fake location, the actuator module will engage the mock location provider. The mock location provider is an Android developer feature to modify the location provided to the app from Android. It requires no change in the OS or the app. The actuator then displays a non-intrusive notification to the user, and signals the session manager to start the app.

End-of-Session Maintenance: When the app finishes execution, the actuator disengages the mock location provider, if engaged. The location-access detector will then detect if the app accessed location to update the app's histogram accordingly. The location access detector performs a "dumpsys location" to exactly detect if the app accessed location or not while running. If it did access location, the location-access detector module updates the app's histogram (increment the number of visits from the current location). It is worth noting that LP-Doctor treats sessions of the same app within 1 min as the same app session.


New App Detected

LP-Doctor

You just installed Uber. This app can access your accurate location information.

What behavior do you want applied?

○ Allow Location Access

○ Block Location Access Completely

● Protect Me when dangerous

Protection Level: low

Done

App will belong to set app_{allow}
App will belong to set app_{block}
App will belong to set $app_{protect}$
Decides the value of α and noise level

Figure 15: The installation menu.

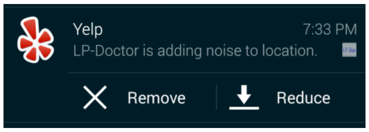
9.2 User Interactions

LP-Doctor interacts with the user to communicate privacy-protection status. It also enables him to populate the privacy profiles for different apps and places. As will be evident below, the main philosophy guiding LP-Doctor's design is to minimize the user interactions, especially intrusive ones. We satisfy two design principles proposed by Felt *et al.* [13] that should guide the design of a permission granting UI. The first principle is to conserve user attention by not issuing excessively repetitive prompts. The second is to avoid interrupting the user's primary tasks.

Bootstrapping Menu: The first communication instance with LP-Doctor takes place upon its installation. LP-Doctor will ask the user to set general configuration options. These options include (1) alerting the user when visiting a new location to set the per-place policies and (2) invoking protection for A&A libraries. The menu will also instruct the user to enable the mock location provider and grant the app "DUMP" permissions through ADB. This interaction takes place only once per LP-Doctor's lifetime.

Installation Menu: LP-Doctor prompts the user when a new (non-system and location-aware) app is installed. The menu enables the user to set the per-app profiles. Fig. 15 shows the displayed menu when an app ("uber" in this case) has finished installation. The user can choose one of three options which populates three app sets: app_{allow}, app_{block}, and $app_{protect}$.

Logically, this menu resembles the per-app location settings for iOS, except that it provides users with an additional option of privacy protection. The protection option acts as a middle-ground between completely allowing and blocking location access to the app. The user will interact with this menu; only once per app, and only for non-system apps that requests the fine location permission. Based on our PhoneLab dataset, we estimate


Yelp 7:33 PM

LP-Doctor is adding noise to location.

Figure 16: LP-Doctor's notification when adding noise.

that the user will be issued this menu on average for one app he installs per five installed apps on the device.

Per-Place Prompts: LP-Doctor relies on the user to decide its actions in different visited places, if he agrees to get prompted when visiting new places. Specifically, whenever the user visits a new place, LP-Doctor prompts him to decide on actions to perform when running apps that the user chose to protect. We call these *per-place policies* (Fig. 13).

The per-place policies apply for apps belonging to the set $app_{protect}$. The user has the option to specify whether to block location access completely, or apply protection. Applying protection will proceed to execute the operations of the threat analyzer as defined in Fig. 14. LP-Doctor allows the user to modify the policies for each app–place combination.

LP-Doctor issues this prompt only when the user launched an app of the set $app_{protect}$ from a new location. From our PhoneLab dataset, we estimate that such a prompt will be issued to the user at most once a week.

Notifications: As specified earlier, the threat actuator displays a non-intrusive notification (Fig. 16) to the user to inform him about the action being taken.

If the action is to allow location access (because the policy dictates so or there is no threat), LP-Doctor notifies the user that there is no action being taken. The user has the option to invoke privacy protection for the current app session. If the user instructs LP-Doctor to add noise for a single app over two consecutive sessions from the same place, LP-Doctor will create a per-place policy for the app and move it to the $app_{protect}$ set if it were part of app_{allow}.

On the other hand, if LP-Doctor decides to add noise to location or block it, it will notify the user of it (Fig. 16). The notification includes two actions that the user can make: remove or reduce noise. If the user overrides LP-Doctor's actions for two consecutive sessions of an app from the same place, LP-Doctor remembers the decision for future reuse.

LP-Doctor leverages the user's behavior to learn the protection level that achieves a favorable privacy–utility tradeoff. Since the mapping between the chosen privacy and noise levels is independent of the running app, the functionality of certain apps might be affected. LP-Doctor allows the user to fine-tune this noise level and then remembers his preference for future reuse.

Reducing the noise level will involve recomputing the fake location with a lower noise value (if no such location has been computed before). One could show that leak of information (from lowering noise level successively) will be capped by that corresponding to the fake location with the lowest noise level released to the service provider.

Using our own and PhoneLab's datasets, we estimate LP-Doctor's need to issue such non-intrusive notification (indicating protection taking place) for only 12% of the sessions on average for each app.

9.3 Limitations

The user-level nature of LP-Doctor introduces some limitations related to certain classes of apps. First, LP-Doctor, like other mechanisms, is inapplicable to apps that require accurate location access such as navigation apps for elongated period of times.

Second, LP-Doctor can't protect the user against apps utilizing unofficial location sources such as "WeChat." Such apps might scan for nearby WiFi access points and then use scan results to compute location. LP-Doctor can't anonymize location fed to such apps, though it can warn the user of the privacy threat incurred if the user is to invoke the location-based functionality. Also, it can offer the user the option to turn off the WiFi on the device to prevent accurate localization by the app when running.

Finally, LP-Doctor doesn't apply privacy protection to the apps continuously accessing location while running in the background. Constantly invoking the mock location provider affects the usability of apps that require fresh and accurate location when running. Fortunately, we found that the majority of the apps don't access location in the background (Section 4). Nevertheless, this still highlights the need for OS support to control apps' location access in the background (like the one that iOS currently provides).

9.4 Evaluation

We now evaluate and report LP-Doctor's overhead on performance, Quality of Service (QoS), and usability.

9.4.1 Performance

LP-Doctor performs a set of operations which delay the app launching. We evaluate this delay on two devices: Samsung Galaxy S4 running Android 4.2.2, and Samsung Galaxy S5 running Android 4.4.4. We recorded the delay in launching a set of apps while running LP-Doctor. We partitioned those apps into two sets. The first (set 1) includes the apps which LP-Doctor doesn't target, while the second (set 2) includes non-system apps that request fine location permissions.

Fig. 17 plots the delay distribution for both devices and for the two app sets. Clearly, apps that belong to the

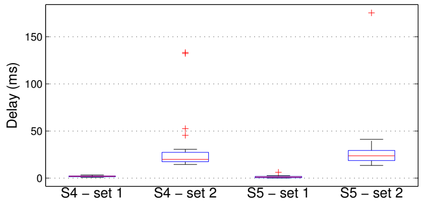


Figure 17: The app launch delay caused by LP-Doctor.

first set experience very minimal delay, varying between 1 and 3ms. The second set of apps experience longer delays without exceeding 50ms for both devices. We also tested LP-Doctor's impact on the battery by recording the battery depletion time when LP-Doctor was running in the background and when it was not. We found that LP-Doctor has less than 10% energy overhead (measured as the difference in battery depletion time). Besides, LP-Doctor runs the same logic as our PhoneLab survey app in the background which 95 users ran over 4 months and reported no performance or battery issues.

9.4.2 User Study

To evaluate the usability of LP-Doctor and its effect on QoS, we conducted a user study over Amazon Mechanical Turk. We designed two Human Intelligence Tasks (HITs), each evaluating a different representative testing scenario of LP-Doctor.

Apps that provide location-based services (LBSes) fall into several categories. On one dimension, an app can *pull* information to the user based on the current location, or it can *push* the user's current location to other users. On another dimension, the app can access the user's location *continuously* or *sporadically* to provide the LBS. One can then categorize apps as: *pull-sporadic* (e.g., weather, Yelp, etc.), *pull-continuous* (e.g., Google Now), *push-sporadic* (e.g., geo-tagging, Facebook check-in, etc.), or *push-continuous* (e.g., Google Latitude). As LP-Doctor isn't effective against apps continuously accessing the user's location (which are a minority to start with), we focus on studying the user's experience of LP-Doctor while using **Yelp**, as a representative example of *pull-sporadic* apps, and **Facebook**, as representative example of *push-sporadic* apps.

We recruited 120 participants for the Yelp HIT and another 122 for the Facebook HIT[3]; we had 227 unique participants in total. On average, each participant completed the HIT in 20min and was compensated $3 for his response. We didn't ask the users for any personal information and nor did LP-Doctor . We limited the study to Android users.

Of the participants: 28% were females vs. 72% males; 32% had high school education, 47% with BS degree or equivalent; and 37% are older than 30 years. Also, 52% of the participants reported that they have taken steps to mitigate privacy threats. Interestingly, 93% of the participants didn't have mock locations enabled on their devices indicating the participants are not tech-savvy.

We constructed the study with a set of connected tasks. In every task, the online form displays a set of instructions/questions that the participant user must follow/answer. After successfully completing the task, LP-Doctor displays a special code that the participant must input to proceed to the next task. In what follows, we describe the various tasks that we asked users to perform and how they responded.

Installing and configuring LP-Doctor: The participants' first task was to download LP-Doctor from Google Play and enable mock locations. We asked the users to rate how difficult it was to enable mock locations on the scale of 1 (easy) to 5 (difficult). 83% of the participants answered with a value of 1 or 2 implying that LP-Doctor is easy to install.

Installation menu: In their second task, the participants interacted with the installation menu (Fig. 15). The users had to install (re-install if already installed) either Yelp or Facebook. Just when either app completes installation, LP-Doctor presents the user with the menu to input the privacy options. The participants reported a positive experience with this menu; 83% reported it was easy to use (rated 1 or 2 on a scale of 1 (easy) to 5 (hard)); 86% said it was informative; 83% thought it provides them with more control than Android's permission; 79% answered it is useful (rated 1 or 2 on a scale of 1 (useful) to 5 (useless)); and 74% would like to have such menu appearing whenever they install a location-aware app (12% answered with not sure).

Impact on QoS: The survey version of LP-Doctor adds noise on top of the user's location regardless of his previous choice. This allowed us to test the impact of adding noise (Laplacian with 1000m radius) to the location accessed by either Yelp or Facebook. We didn't ask the participants to assess the effect of location anonymization on the QoS directly. Rather, we asked the Yelp respondents to report their satisfaction with the list of restaurants returned by the app. While we asked the Facebook respondents to indicate whether the list of places to check-in from is relevant to them. The participants in the first HIT indicated that Yelp ran normally (82%), the restaurant search results were relevant (73%), the user experience didn't change (76%), and Yelp need not access the user's accurate location (67%).

The Facebook HIT participants exhibited similar results: Facebook ran normally (80%), the list of places to check-in was relevant (60%), user experience didn't change (80%), and Facebook need not access the user's

[3] https://kabru.eecs.umich.edu/wordpress/wp-content/uploads/lp-doctor-survey-fb.pdf

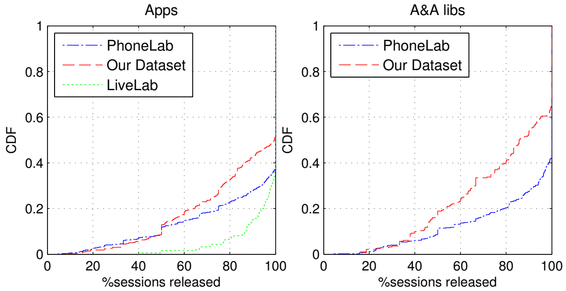


Figure 18: The distribution of percentage of sessions where apps maintain QoS for apps (left) and A&A libraries (right).

accurate location (80%).

Fig. 18 shows the percentage of sessions (for all app–user combinations) that won't experience any noise addition according to our datasets. It is obvious that the percentage of sessions with potential loss in QoS (when LP-Doctor adds noise) is minimal (less than 20%, a bit higher if the user opts for A&A libraries protection). Our user study shows that more than 70% of the users won't experience loss in QoS in these sessions. For those users who do face loss in QoS, LP-Doctor provides them with the option of adjusting the noise level at runtime through the notifications.

Notifications: In the final task, we asked the participants to test the noise reduction feature that allows for a personalized privacy–utility trade-off. After they reduced the noise level, they would invoke the location-based feature in both Yelp and Facebook and check if the results were improved. Indeed, most of the participants who reported loss in QoS reported the Yelp's search results (64%) and Facebook's check-in places (70%) improved after reducing the noise.

The participants also indicated the the noise reduction feature is easy to use (75%). 86% of the participants won't mind having this feature whenever they launch a location-aware app.

Post-study questions: As we couldn't control the per-place prompts given our study design, we asked the participants for their opinion about being prompted when visiting new places (per-place prompts). Only 54% answered they would prefer prompted, 37% answered negatively, and the rest answered "I am not sure." These responses are consistent with our design decision; the user has to approve per-place prompts when initially configuring LP-Doctor as they are not automatically enabled.

Also, 82% of the participants felt comfortable that Facebook (80%) and Yelp (85%) didn't access their accurate location. Finally, 77% of the participants answered "Yes" when asked about installing LP-Doctor or other tool to protect their location privacy. Only 11% answered "No" and the rest answered with "I am not sure." This result comes at an improvement over the 52% who initially said they took steps in the past to address location-privacy threat.

In summary, we conducted one of the few studies (e.g., [8]) that evaluate a location-privacy protection mechanism in the wild. We showed that location-privacy protection is feasible in practice where a balance between QoS, usability, and privacy could be achieved.

10 Conclusion

In this paper, we posed a question about the effectiveness of OS-based location-access controls and whether they can be improved. To answer this question, we conducted a location-collection campaign that considers location-privacy threats from the perspective of mobile apps. From this campaign, we observed, modeled, and categorized profiling as being the prominent privacy threat from location access for both apps and A&A libraries. We concluded that controlling location access per session is needed to balance between loss in QoS and privacy protection. As existing OS controls don't readily provide such functionality, we proposed LP-Doctor, a user-level tool that helps the user better utilize existing OS-based location-access controls. LP-Doctor is shown to be able to mitigate privacy threats from both apps and A&A libraries with little effect on usability and QoS. In future, we would like to test LP-Doctor in the wild and use it to explore the dynamics that affect users' decisions to install a location-privacy protection mechanism.

11 Acknowledgments

We would like to thank the anonymous reviewers and the shepherd, Reza Shokri, for constructive suggestions. The work reported in this paper was supported in part by the NSF under Grants 0905143 and 1114837, and the ARO under W811NF-12-1-0530.

References

[1] ALMUHIMEDI, H., SCHAUB, F., SADEH, N., ADJERID, I., ACQUISTI, A., GLUCK, J., CRANOR, L. F., AND AGARWAL, Y. Your location has been shared 5,398 times!: A field study on mobile app privacy nudging. In *Proceedings of CHI '15* (2015), pp. 787–796.

[2] AMINI, S., LINDQVIST, J., HONG, J., LIN, J., TOCH, E., AND SADEH, N. Caché: Caching location-enhanced content to improve user privacy. In *Proceedings of MobiSys '11* (New York, NY, USA, 2011), ACM, pp. 197–210.

[3] ANDRÉS, M. E., BORDENABE, N. E., CHATZIKOKOLAKIS, K., AND PALAMIDESSI, C. Geo-indistinguishability: Differential privacy for location-based systems. In *Proceedings of CCS '13*.

[4] Andrienko, G., Gkoulalas-Divanis, A., Gruteser, M., Kopp, C., Liebig, T., and Rechert, K. Report from dagstuhl: the liberation of mobile location data and its implications for privacy research. *SIGMOBILE Mob. Comput. Commun. Rev. 17*, 2 (July 2013), 7–18.

[5] Ashford, W. Free mobile apps a threat to privacy, study finds. http://www.computerweekly.com/news/2240169770/Free-mobile-apps-a-threat-to-privacy-study-finds, October 2012.

[6] Beresford, A. R., Rice, A., Skehin, N., and Sohan, R. Mockdroid: Trading privacy for application functionality on smartphones. In *Proceedings of HotMobile '11* (New York, NY, USA, 2011), ACM, pp. 49–54.

[7] Bettini, C., Wang, X., and Jajodia, S. Protecting privacy against location-based personal identification. *Secure Data Management* (2005), 185–199.

[8] Bilogrevic, I., Huguenin, K., Mihaila, S., Shokri, R., and Hubaux, J.-P. Predicting Users' Motivations behind Location Check-Ins and Utility Implications of Privacy Protection Mechanisms. In *NDSS'15* (2015).

[9] Bordenabe, N. E., Chatzikokolakis, K., and Palamidessi, C. Optimal geo-indistinguishable mechanisms for location privacy. In *Proceedings of CCS '14* (2014), pp. 251–262.

[10] Clauset, A., Shalizi, C. R., and Newman, M. E. J. Power-law distributions in empirical data. *SIAM Rev. 51*, 4 (Nov. 2009), 661–703.

[11] de Montjoye, Y.-A., Hidalgo, C. A., Verleysen, M., and Blondel, V. D. Unique in the crowd: The privacy bounds of human mobility. *Sci. Rep. 3* (Mar 2013).

[12] Fawaz, K., and Shin, K. G. Location privacy protection for smartphone users. In *Proceedings of CCS '14* (New York, NY, USA, 2014), ACM, pp. 239–250.

[13] Felt, A. P., Egelman, S., Finifter, M., Akhawe, D., and Wagner, D. How to ask for permission. In *Proceedings of HotSec'12* (2012).

[14] Fisher, D., Dorner, L., and Wagner, D. Short paper: Location privacy: User behavior in the field. In *Proceedings of SPSM '12* (2012), pp. 51–56.

[15] Freudiger, J., Shokri, R., and Hubaux, J.-P. Evaluating the Privacy Risk of Location-Based Services. In *Financial Cryptography and Data Security (FC)* (2011).

[16] Fritsch, L. Profiling and location-based services (lbs). In *Profiling the European Citizen*, M. Hildebrandt and S. Gutwirth, Eds. Springer Netherlands, 2008, pp. 147–168.

[17] Fu, H., Yang, Y., Shingte, N., Lindqvist, J., and Gruteser, M. A field study of run-time location access disclosures on android smartphones. In *Proceedings of USEC 2014*.

[18] Golle, P., and Partridge, K. On the anonymity of home/work location pairs. In *Proceedings of PERVASIVE '09* (Berlin, Heidelberg, 2009), Springer-Verlag, pp. 390–397.

[19] Goodwin, C. A conceptualization of motives to seek privacy for nondeviant consumption. *Journal of Consumer Psychology 1*, 3 (1992), 261 – 284.

[20] Guha, S., Jain, M., and Padmanabhan, V. N. Koi: A location-privacy platform for smartphone apps. In *Proceedings of NSDI'12* (2012), USENIX Association, pp. 14–14.

[21] Higgins, E. T. Self-discrepancy: a theory relating self and affect. *Psychological Review 94*, 3 (Jul 1987), 319–340.

[22] Hoh, B., Gruteser, M., Xiong, H., and Alrabady, A. Achieving guaranteed anonymity in gps traces via uncertainty-aware path cloaking. *IEEE TMC 9*, 8 (August 2010), 1089–1107.

[23] Hornyack, P., Han, S., Jung, J., Schechter, S., and Wetherall, D. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of CCS '11* (2011), pp. 639–652.

[24] Jung, J., Han, S., and Wetherall, D. Short paper: Enhancing mobile application permissions with runtime feedback and constraints. In *Proceedings of SPSM '12* (2012), pp. 45–50.

[25] Krumm, J. Inference attacks on location tracks. In *Proceedings of PERVASIVE '07* (2007), Springer-Verlag, pp. 127–143.

[26] Krumm, J. Realistic driving trips for location privacy. In *Proceedings of PERVASIVE '09* (2009), Springer-Verlag, pp. 25–41.

[27] Kullback, S., and Leibler, R. A. On information and sufficiency. *Ann. Math. Statist. 22*, 1 (03 1951), 79–86.

[28] Lu, H., Jensen, C. S., and Yiu, M. L. Pad: privacy-area aware, dummy-based location privacy in mobile services. In *Proceedings of MobiDE '08* (2008), pp. 16–23.

[29] Meyerowitz, J., and Roy Choudhury, R. Hiding stars with fireworks: location privacy through camouflage. In *Proceedings of MobiCom '09* (2009), pp. 345–356.

[30] Micinski, K., Phelps, P., and Foster, J. S. An Empirical Study of Location Truncation on Android. In *Mobile Security Technologies (MoST '13)* (San Francisco, CA, May 2013).

[31] Nandugudi, A., Maiti, A., Ki, T., Bulut, F., Demirbas, M., Kosar, T., Qiao, C., Ko, S. Y., and Challen, G. Phonelab: A large programmable smartphone testbed. In *Proceedings of SENSEMINE'13* (2013), pp. 4:1–4:6.

[32] Palanisamy, B., and Liu, L. Mobimix: Protecting location privacy with mix-zones over road networks. In *ICDE 2011* (april 2011), pp. 494 –505.

[33] Pingley, A., Zhang, N., Fu, X., Choi, H.-A., Subramaniam, S., and Zhao, W. Protection of query privacy for continuous location based services. In *INFOCOM'11* (April 2011), IEEE.

[34] Shepard, C., Rahmati, A., Tossell, C., Zhong, L., and Kortum, P. Livelab: measuring wireless networks and smartphone users in the field. *SIGMETRICS Perform. Eval. Rev. 38*, 3 (Jan. 2011), 15–20.

[35] Shokri, R., Theodorakopoulos, G., Danezis, G., Hubaux, J.-P., and Le Boudec, J.-Y. Quantifying location privacy: the case of sporadic location exposure. In *Proceedings of PETS'11* (2011), pp. 57–76.

[36] Shokri, R., Theodorakopoulos, G., Le Boudec, J., and Hubaux, J. Quantifying location privacy. In *Security and Privacy (SP), 2011 IEEE Symposium on* (may 2011), pp. 247 –262.

[37] Shokri, R., Theodorakopoulos, G., Troncoso, C., Hubaux, J.-P., and Le Boudec, J.-Y. Protecting location privacy: Optimal strategy against localization attacks. In *Proceedings of CCS '12* (2012), pp. 617–627.

[38] Thurm, S., and Kane, Y. I. Your apps are watching you. http://online.wsj.com/article/SB10001424052748704694004576020083703574602.html, December 2010.

[39] Tripp, O., and Rubin, J. A bayesian approach to privacy enforcement in smartphones. In *USENIX Security 14* (San Diego, CA, 2014), USENIX Association, pp. 175–190.

[40] Vajna, S., Tth, B., and Kertsz, J. Modelling bursty time series. *New Journal of Physics 15*, 10 (2013), 103023.

[41] Zang, H., and Bolot, J. Anonymization of location data does not work: a large-scale measurement study. In *Proceedings of MobiCom '11* (New York, NY, USA, 2011), ACM, pp. 145–156.

[42] Zickuhr, K. Location-based services. http://pewinternet.org/Reports/2013/Location.aspx, September 2013.

LinkDroid: Reducing Unregulated Aggregation of App Usage Behaviors

Huan Feng, Kassem Fawaz, and Kang G. Shin
Department of Electrical Engineering and Computer Science
The University of Michigan
{*huanfeng, kmfawaz, kgshin*}*@umich.edu*

Abstract

Usage behaviors of different smartphone apps capture different views of an individual's life, and are largely independent of each other. However, in the current mobile app ecosystem, a curious party can covertly link and aggregate usage behaviors of the same user across different apps. We refer to this as *unregulated aggregation* of app-usage behaviors. In this paper, we present a fresh perspective of unregulated aggregation, focusing on monitoring, characterizing and reducing the underlying linkability across apps. The cornerstone of our study is the *Dynamic Linkability Graph* (DLG) which tracks app-level linkability during runtime. We observed how DLG evolves on real-world users and identified real-world evidence of apps abusing IPCs and OS-level identifying information to establish linkability. Based on these observations, we propose a linkability-aware extension to current mobile operating systems, called `LinkDroid`, which provides runtime monitoring and mediation of linkability across different apps. `LinkDroid` is a client-side solution and compatible with the existing smartphone ecosystem. It helps end-users "sense" this emerging threat and provides them intuitive opt-out options.

1 Introduction

Mobile users run apps for various purposes, and exhibit very different or even unrelated behaviors in running different apps. For example, a user may expose his chatting history to WhatsApp, mobility traces to Maps, and political interests to CNN. Information about a single user, therefore, is scattered across different apps and each app acquires only a partial view of the user. Ideally, these views should remain as 'isolated islands of information' confined within each of the different apps. In practice, however, once the users' behavioral information is at the hands of the apps, it may be shared or leaked in an arbitrary way without the users' control or consent. This makes it possible for a curious adversary to aggregate usage behaviors of the same user across multiple apps without his knowledge and consent, which we refer to as *unregulated aggregation* of app-usage behaviors.

In the current mobile ecosystem, many parties are interested in conducting unregulated aggregation, including:

- *Advertising Agencies* embed ad libraries in different apps, establishing an explicit channel of cross-app usage aggregation. For example, Grindr is a geosocial app geared towards gay users, and BabyBump is a social network for expecting parents. Both apps include the same advertising library, MoPub, which can aggregate their information and recommend related ads, such as on gay parenting books. However, users may not want this type of unsolicited aggregation, especially across sensitive aspects of their lives.
- *Surveillance Agencies* monitor all aspects of the population for various precautionary purposes, some of which may cross the 'red line' of individuals' privacy. It has been widely publicized that NSA and GCHQ are conducting public surveillance by aggregating information leaked via mobile apps, including popular ones such as Angry Birds [3]. A recent study [26] shows that a similar adversary is able to attribute up to 50% of the mobile traffic to the "monitored" users, and extract detailed personal interests, such as political views and sexual orientations.
- *IT Companies* in the mobile industry frequently acquire other app companies, harvesting vast user base and data. Yahoo alone acquired more than 10 mobile app companies in 2013, with Facebook and Google following closely behind [1]. These acquisitions allow an IT company to link and aggregate behaviors of the same user from multiple apps without the user's consent. Moreover, if the acquiring com-

pany (such as Facebook) already knows the users' real identities, usage behaviors of all the apps it acquires become identifiable.

These scenarios of unregulated aggregation are realistic, financially motivated, and are only becoming more prevalent in the foreseeable future. In spite of this grave privacy threat, the process of unregulated aggregation is unobservable and works as a black box — no one knows what information has actually been aggregated and what really happens in the cloud. Users, therefore, are largely unaware of this threat and have no opt-out options. Existing proposals disallow apps from collecting user behaviors and shift part of the app logic (e.g., personalization) to the mobile OS or trusted cloud providers [7, 17]. This, albeit effective, is against the incentive of app developers and requires construction of a new ecosystem. Therefore, there is an urgent need for a practical solution that is compatible with the existing mobile ecosystem.

In this paper, we propose a new way of addressing the unregulated aggregation problem by monitoring, characterizing and reducing the underlying linkability across apps. Two apps are *linkable* if they can associate their usage behaviors of the same user. This linkability is the prerequisite of conducting unregulated aggregation and represents an upper-bound of the potential threat. Researchers studied linkability under domain-specific scenarios, such as on movie reviews [19] and social networks [16]. In contrast, we focus on the linkability that is ubiquitous in the mobile ecosystem and introduced by domain-independent factors, such as device IDs, account numbers, location and inter-app communications. Specifically, we model mobile apps on the same device as a *Dynamic Linkability Graph* (DLG) which monitors apps' access to OS-level identifying information and cross-app communication channels. DLG quantifies the potential threat of unregulated aggregation and allows us to monitor the linkability across apps during runtime.

We implemented DLG as an Android extension and observed how it evolved on 13 users during a period of 47 days. The results reveal an alarming view of the app-level linkability in the wild. Two random apps (installed by the same user) are linkable with a probability of 0.81. Specifically, 86% of the apps a user installed are directly linkable to the Facebook app, namely, his real identity. In particular, we found that apps frequently abuse OS-level information and inter-process communication (IPC) channels in unexpected ways, establishing the linkability that is unrelated to app functionalities. For example, we found that many of the apps requesting account information collect all of the user's accounts even when they only need one to function correctly. We also noticed that some advertising agencies, such as Admob and Facebook, use IPCs to share user identifiers with other apps, completely bypassing system permissions and controls. Furthermore, we identified cases when different apps write and read the same persistent file in shared storage to exchange user identifiers. The end-users should be promptly warned about these unexpected behaviors to reduce unnecessary linkability.

Based on the above observations, we propose `LinkDroid`, a linkability-aware extension to Android which provides runtime monitoring and mediation of the linkability across apps. `LinkDroid` introduces a new dimension to privacy protection on smartphones. Instead of checking whether some app behavior poses direct privacy threat, `LinkDroid` warns users about how it implicitly affects the linkability across apps. Practicality is a main driver for the design of `LinkDroid`. It extends the widely-deployed (both runtime and install-time) permission model on the mobile OS that end-users are already familiar with. Specifically, `LinkDroid` provides the following privacy-enhancing features:

- **Install-Time Obfuscation:** `LinkDroid` obfuscates device-specific identifiers that have no influence on most app functionalities, such as IMEI, Android ID, etc. We perform this during install-time to maintain the consistency of these identifiers within each app.
- **Runtime Linkability Monitoring:** When an app tries to perform a certain action that introduces additional linkability, users will receive a just-in-time prompt and an intuitive risk indicator. Users can then exercise runtime access control and choose any of the opt-out options provided by `LinkDroid`.
- **Unlinkable Mode:** The user can start an app in unlinkable mode. This will create a new instance of the app which is unlinkable with other apps. All actions that may establish a direct association with other apps will be denied by default. This way, users can enjoy finer-grained privacy protection, unlinking only a set of app sessions.

We evaluated `LinkDroid` on the same set of 13 users as in our measurement and found that `LinkDroid` reduces the cross-app linkability substantially with little loss of app performance. The probability of two random apps being linkable is reduced from 0.81 to 0.21, and the percentage of apps that are directly linkable to Facebook drops from 86% to 18%. On average, a user only needs to handle 1.06 prompts per day in the 47-day experiments and the performance overhead is marginal.

This paper makes the following contributions:

1. Introduction of a novel perspective of defending against unregulated aggregation by addressing the underlying linkability across apps (Section 2).

2. Proposal of the Dynamic Linkability Graph (DLG) which enables runtime monitoring of cross-app linkability (Section 3).

3. Identification of real-world evidence of how apps abuse IPCs and OS-level information to establish linkability across apps (Section 4).

4. Addition of a new dimension to access control based on the runtime linkability, and development of a practical countermeasure, `LinkDroid`, to defend against unregulated aggregation (Section 5).

2 Privacy Threats: A New Perspective

In this section, we will first introduce our threat model of unregulated aggregation and then propose a novel perspective of addressing it by monitoring, characterizing and reducing the linkability across apps. We will also summarize the explicit/implicit sources of linkability in the current mobile app ecosystem.

2.1 Threat Model

In this paper, we target unregulated aggregation across app-usage behaviors, i.e., when an adversary aggregates usage behaviors across multiple functionally-independent apps without users' knowledge or consent. In our threat model, an adversary can be any party that collects information from multiple apps or controls multiple apps, such as a widely-adopted advertising agency, an IT company in charge of multiple authentic apps, or a set of malicious colluding apps. We assume the mobile operating system and network operators are trustworthy and will not collude with the adversary.

2.2 Linkability: A New Perspective

There are many parties interested in conducting unregulated aggregation across apps. In practice, however, this process is unobservable and works as a black box — no one knows what information an adversary has collected and whether it has been aggregated in the cloud. Existing studies propose to disable mobile apps from collecting usage behaviors and shift part of the app logic to trusted cloud providers or mobile OS [7, 17]. These solutions, albeit effective, require building a new ecosystem and greatly restrict functionalities of the apps. Here, we address unregulated aggregation from a very different angle by monitoring, characterizing and reducing the underlying linkability across mobile apps. Two apps are *linkable* if they can associate usage behaviors of the same user. This linkability is the prerequisite of conducting unregulated aggregation, and represents an "upper-bound" of the potential threat. In the current mobile app ecosystem, there are various sources of linkability that an adversary can exploit. Researchers have studied linkability under several domain-specific scenarios, such as movie reviews [19] and social networks [16]. Here, we focus on the linkability that is ubiquitous and domain-independent. Specifically, we group its contributing sources into the following two fundamental categories.

Type	*2013-3*	*2013-10*	*2014-8*	*2015-1*
Android ID	80%	84%	87%	91%
IMEI	61%	64%	65%	68%
MAC	28%	42%	51%	55%
Account	24%	29%	32%	35%
Contacts	21%	26%	33%	37%

Table 1: Apps are increasingly interested in requesting persistent and consistent identifying information during the past few years.

OS-Level Information The mobile OS provides apps ubiquitous access to various system information, many of which can be used as consistent user identifiers across apps. These identifiers can be *device-specific*, such as MAC address and IMEI, *user-specific*, such as phone number or account number, or *context-based*, such as location or IP clusters. We conducted a longitudinal measurement study from March 2013 to January 2015, on the top 100 free Android apps in each category. We excluded the apps that are rarely downloaded, and considered only those with more than 1 million downloads. We found that apps are getting increasingly interested in requesting persistent and consistent identifying information, as shown in Table 1. By January 2015, 96% of top free apps request both the Internet access and at least one persistent identifying information. These identifying vectors, either explicit or implicit, allow two apps to link their knowledge of the same user at a remote side without even trying to bypass on-device isolation of the mobile OS.

Inter-Process Communications The mobile OS provides explicit Inter-Process Communication (IPC) channels, allowing apps to communicate with each other and perform certain tasks, such as export a location from Browser and open it with Maps. Since there is no existing control on IPC, colluding apps can exchange identifying information of the user and establish linkability covertly, without the user's knowledge. They can even synchronize and agree on a randomly-generated sequence as a custom user identifier, without accessing any system resource or permission. This problem gets more complex since apps can also conduct IPC implicitly by reading and writing shared persistent storage (SD card

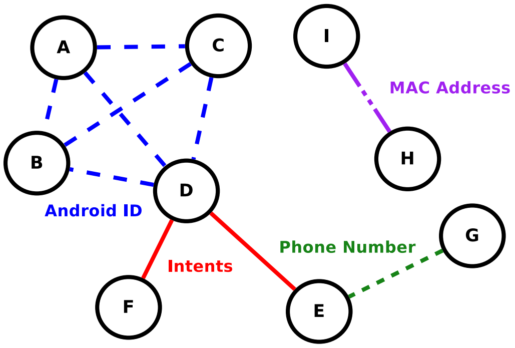


Figure 1: An illustrative example of DLG. Edges of different types represent linkability introduced by different sources.

and databases). As we will show in Section 4, these exploitations are not hypothetical and have already been utilized by real-world apps.

3 Dynamic Linkability Graph

The cornerstone of our work is the Dynamic Linkability Graph (DLG). It enables us to monitor app-level linkability during runtime and quantify the linkability introduced by different contributing sources. In what follows, we will elaborate on the definition of DLG, the linkability sources it considers, and describe how it can be implemented as an extension of Android.

3.1 Basic Concepts

We model linkability across different apps on the same device as an undirected graph, which is called the *Dynamic Linkability Graph* (DLG). Nodes in DLG represent apps and edges represent linkability introduced by different contributing sources. DLG monitors the linkability during runtime by tracking the apps' access to various OS-level information and IPC channels. An edge exists between two apps if they accessed the same identifying information or engaged in an IPC. Fig. 15 presents an illustrative example of DLG.

DLG presents a comprehensive view of the linkability across all installed apps. An individual adversary, however, may only observe a subgraph of the DLG. For example, an advertising agency only controls those apps (nodes) that incorporate the same advertising library; an IT corporate only controls those apps (nodes) it has already acquired. In the rest of the paper, we focus on the generalized case (the entire DLG) instead of considering each adversary individually (subgraphs of DLG).

3.2 Definitions and Metrics

Linkable Two apps a and b are *linkable* if there is a path between them. In Fig. 15, app A and F are linkable, app A and H are not linkable.

Gap is defined as the number of nodes (excluding the end nodes) on the shortest path between two linkable apps a and b. It represents how many additional apps an adversary needs to control in order to link information across a and b. For example, in Fig. 15, $gap_{A,D} = 0$, $gap_{A,E} = 1$, $gap_{A,G} = 2$.

Linking Ratio (LR) of an app is defined as the number of apps it is linkable to, divided by the number of all installed apps. LR ranges from 0 to 1 and characterizes to what extent an app is linkable to others. In DLG, LR equals to the size of the *Largest Connected Component* (LCC) this app resides in, excluding itself, divided by the size of the entire graph, also excluding itself:

$$LR_a = \frac{size(LCC_a) - 1}{size(DLG) - 1}$$

Linking Effort (LE) of an app is defined as the *Linking Effort* (LE) of an app as the average *gap* between it and all the apps it is linkable to. LE_a characterizes the difficulty in establishing linkability with a. $LE_a = 0$ means that to link information from app a and any random app it is linkable to, an adversary does not need additional information from a third app.

$$LE_a = \sum_{\substack{b \in LCC_a \\ b \neq a}} \frac{gap_{a,b}}{size(LCC_a) - 1}$$

LR and LE describe two orthogonal views of the DLG. In general, LR represents the quantity of links, describing the percentage of all installed apps that are linkable to a certain app, whereas LE characterizes the quality of links, describing the average amount of effort an adversary needs to make to link a certain app with other apps. In Fig. 15, $LR_A = 6/8$, $LR_H = 1/8$; $LE_A = \frac{0+0+0+1+1+2}{7-1} = 4/6, LE_H = 0$.

GLR and GLE Both LR and LE are defined for a single app, and we also need two similar definitions for the entire graph. So, we introduce *Global Linking Ratio* (GLR) and *Global Linking Effort* (GLE). *GLR* represents the probability of two randomly selected apps being linkable, while *GLE* represents the number of apps an adversary needs to control to link two random apps.

$$GLR = \sum_a \frac{LR_a}{size(DLG)}$$

$$GLE = \frac{1}{\sum_a size(LCC_a) - 1} \sum_b \sum_{\substack{c \in LCC_b \\ c \neq b}} gap_{b,c}$$

In graph theory, *GLE* is also known as the *Characteristic Path Length* (CPL) of a graph, which is widely used in Social Network Analysis (SNA) to characterize whether the network is easily negotiable or not.

3.3 Sources of Linkability

DLG maintains a dynamic view of app-level linkability by monitoring runtime behaviors of the apps. Specifically, it keeps track of apps' access to *device-specific* identifiers (IMEI, Android ID, MAC), *user-specific* identifiers (Phone Number, Accounts, Subscriber ID, ICC Serial Number), and *context-based* information (IP, Nearby APs, Location). It also monitors explicit IPC channels (Intent, Service Binding) and implicit IPC channel (Indirect RW, i.e., reading and writing the same file or database). This is not an exhaustive list but covers most standard and widely-used aggregating channels. Table 2 presents a list of all the contributing sources we consider and the details of each source will be elaborated in Section 3.4.

The criterion of two apps being linkable differs depending on the linkability source. For consistent identifiers that are obviously unique — Android ID, IMEI, Phone Number, MAC, Subscriber ID, Account, ICC Serial Number — two apps are linkable if they both accessed the same type of identifier. For pair-wise IPCs — intents, service bindings, and indirect RW — the two communicating parties involved are linkable. For implicit and fuzzy information, such as location, nearby APs, and IP, there are well-known ways to establish linkability as well. User-specific location clusters (Points of Interests, or PoIs) is already known to be able to uniquely identify a user [11, 15, 29]. Therefore, an adversary can link different apps by checking whether the location information they collected reveal the same PoIs. Here, the PoIs are extracted using a lightweight algorithm as used in [5, 10]. We select the top 2 PoIs as the linking standard, which typically correspond to home and work addresses. Similarly, the consistency and persistence of a user's PoIs are also reflected on its AP clusters and frequently-used IP addresses. This property allows us to establish linkability across apps using these fuzzy contextual information.

3.4 DLG: A Mobile OS Extension

DLG gives us the capability to construct cross-app linkability from runtime behaviors of the apps. Here, we introduce how it can be implemented as an extension to current mobile operating systems, using Android as an illustrative example. We also considered other implementation options, such as user-level interception (Aurasium [28]) or dynamic OS instrumentation (Xposed Framework [27]). The former is insecure since the extension resides in the attacker's address space and the latter is not comprehensive because it cannot handle the native code of an app. However, the developer can always implement a useful subset of DLG using one of these more deployable techniques.

Category	*Type*	*Source*
OS-level Info.	Device	IMEI Android ID MAC
	Personal	Phone # Account Subscriber ID ICC Serial #
	Contextual	IP Nearby APs Location (PoIs)
IPC Channel	Explicit	Intent Service Binding
	Implicit	Indirect RW

Table 2: DLG considers the linkability introduced by 10 types of OS-level information and 3 IPC channels.

Android Basics Android is a Linux-based mobile OS developed by Google. By default, each app is assigned a different Linux uid and lives in its own sandbox. Inter-Process Communications (IPCs) are provided across different sandboxes, based on the Binder protocol which is inherently a lightweight RPC (Remote Procedure Call) mechanism. There are four different types of components in an Android app: Activity, Service, Content Provider, and Broadcast Receiver. Each component represents a different way to interact with the underlying system: Activity corresponds to a single screen supporting user interactions; Service runs in the background to perform long-running operations and processing; Content Provider is responsible for managing and querying of persistent data such as database; and Broadcast Receiver listens to system-wide broadcasts and filters those it is interested in. Next, we describe how we instrument the Android framework to monitor app's interactions with the system and each other via these components.

Implementation Details In order to construct a DLG in Android, we need to track apps' access to various OS-

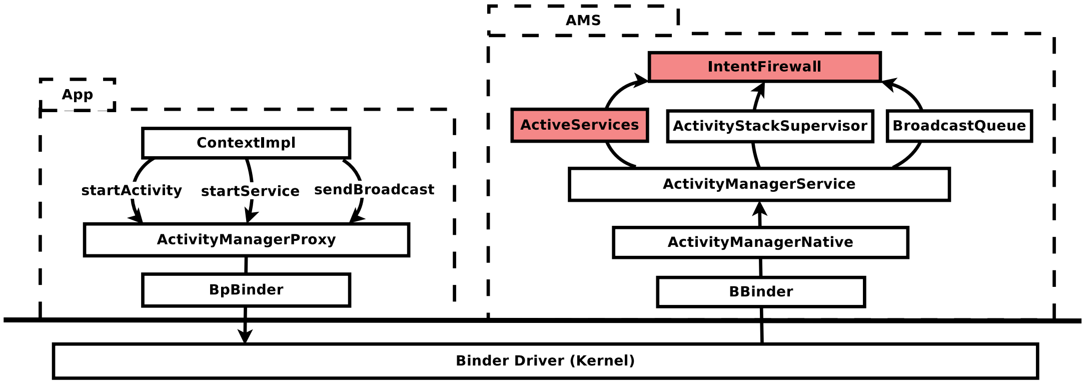


Figure 3: We extend the centralized intent filter in Android (`com.android.server.firewall.IntentFirewall`) to intercept all the intents across apps.

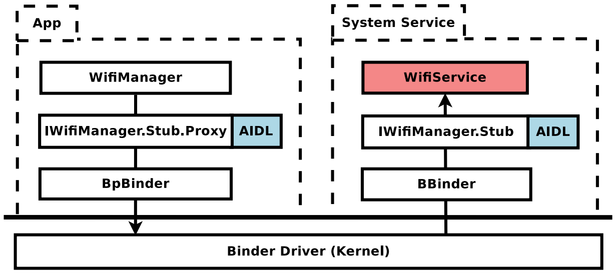


Figure 2: We instrument system services (red shaded region) to record which app accessed which identifier using Wi-Fi service as an example.

level information as well as IPCs between apps. Next, we describe how we achieve this by instrumenting different components of the Android framework.

Apps access most identifying information, such as IMEI and MAC, by interacting with different system services. These system services are parts of the Android framework and have clear interfaces defined in AIDL (Android Interface Definition Language). By instrumenting the public functions in each service that return persistent identifiers, we can have a timestamped record of which app accessed what type of identifying information via which service. Fig. 2 gives a detailed view of where to instrument using the Wi-Fi service as an example.

On the other hand, apps access some identifying information, such as Android ID, by querying system content providers. Android framework has a universal choke point for all access to remote content providers — the server-side stub class `ContentProvider.Transport`. By instrumenting this class, we know which database (uri) an app is accessing and with what parameters and actions. Fig. 4 illustrates how an app accesses remote Content Provider and explains which part to modify in order to log the information we need.

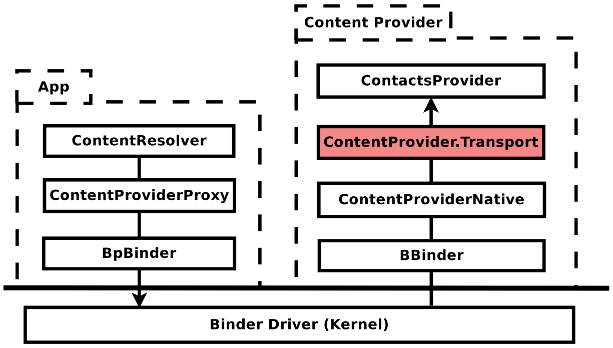


Figure 4: We instrument Content Provider (shaded region) to record which app accessed which database with what parameters.

Apps can launch IPCs explicitly, using Intents. Intent is an abstract description of an operation to be performed. It can either be sent to a specific target (app component), or broadcast to the entire system. Android has a centralized filter which enforces system-wide policies for all Intents. We extend this filter (`com.android.server.firewall.IntentFirewall`) to record and intercept all Intent communications across apps (see Fig. 3). In addition to Intents, Android also allows an app to communicate explicitly with another app by binding to one of the services it exports. Once the binding is established, the two apps can communicate under a client-server model. We instrument `com.android.server.am.ActiveServices` in the Activity Manager to monitor all the attempts to establish service bindings across apps.

Apps can also conduct IPCs implicitly by exploiting shared persistent storage. For example, two apps can write and read the same file in the SD card to exchange identifying information. Therefore, we need to monitor read and write access to persistent storage. External storage in Android are wrapped by a FUSE (Filesystem in

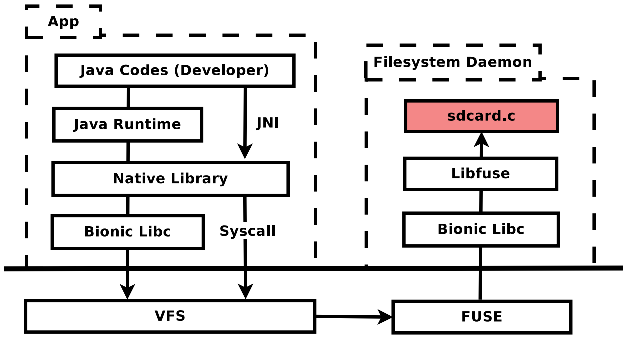


Figure 5: We customize the FUSE daemon under `/system/core/sdcard/sdcard.c` to intercept apps' access to shared external storage.

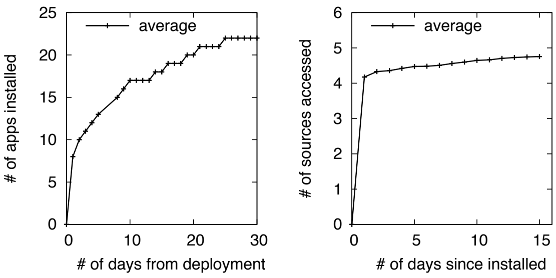


Figure 6: For an average user, more than 80% of the apps are installed in the first two weeks after deployment; each app accesses most of the linkability sources it's interested in during the first day of its installation.

Userspace) daemon which enables user-level permission control. By modifying this daemon, we can track which app reads or writes which files (see Fig. 5). This allows us to implement a Read-Write monitor which captures implicit communications via reading a file which has previously been written by another app. Besides external storage, our Read-Write monitor also considers similar indirect communications via system Content Providers.

We described how to monitor all formal ways an app can interact with system components (Services, Content Providers) and other apps (Intents, service bindings, and indirect RW). This methodology is fundamental and can be extended to cover other potential linkability sources (beyond our list) as long as a clear definition is given. By placing hooks at the aforementioned locations in the system framework, we get all the information needed to construct a DLG. For our measurement study, we simply log and upload these statistics to a remote server for analysis. In our countermeasure solutions, these are used locally to derive dynamic defense decisions.

4 Linkability in Real World

In this section, we study app-level linkability in the real world. We first present an overview of linkability, showing the current threats we're facing. Then, we go through the linkability sources and analyze to what extent each of the sources is contributing to the linkability. Finally, we shed light on how these sources can be or have been exploited for reasons unrelated to app functionalities. This paves the way for us to develop a practical countermeasure.

4.1 Deployment and Settings

We prototyped DLG on Cyanogenmod 11 (based on Android 4.4.1) and installed the extended OS on 7 Samsung Galaxy IV devices and 6 Nexus V devices. We recruited 13 participants from the students and staff in our institution, spanning over 8 different academic departments. Of the 13 participants, 6 of the participants are females and 7 are males. Before using our experimental devices, 7 of them were Android users and 6 were iPhone users. Participants are asked to operate their devices normally without any extra requirement. They are given the option to temporarily turn off our extension if they want more privacy when performing certain tasks. Logs are uploaded once per hour when the device is connected to Wi-Fi. We exclude built-in system apps (since the mobile OS is assumed to be benign in our threat model) and consider only third-party apps that are installed by the users themselves. Note that our study is limited in its size and the results may not generalize.

4.2 Data and Findings

We observed a total of 215 unique apps during a 47-day period for 13 users. On average, each user installed 26 apps and each app accessed 4.8 different linkability sources. We noticed that more than 80% of the apps are installed within the first two weeks after deployment, and apps would access most of the linkability sources they are interested in during the first day of their installation (see Fig. 6). This suggests that a relative short-term (a few weeks) measurement would be enough to capture a representative view of the problem.

Overview: Our measurement indicates an alarming view of the threat: two random apps are linkable with a probability of 0.81, and an adversary only needs to control 2.2 apps (0.2 additional app), on average, to link them. This means that an adversary in the current ecosystem can aggregate information from most apps without additional efforts (i.e., controlling a third app). Specifically, we found that 86% of the apps a user installed on his device are directly linkable to the Facebook app, namely, his real identity. This means almost all the activ-

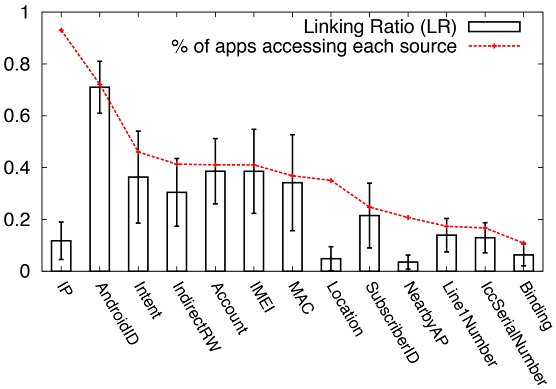


Figure 7: The percentage of apps accessing each source, and the linkability (LR) an app can get by exploiting each source.

ities a user exhibited using mobile apps are identifiable, and can be linked to the real person.

Breakdown by Source: This vast linkability is contributed by various sources in the mobile ecosystem. Here, we report the percentage of apps accessing each source and the linkability (LR) an app can acquire by exploiting each source. The results are provided in Fig. 7. We observed that except for device identifiers, many other sources contributed to the linkability substantially. For example, an app can be linked to 39% of all installed apps (LR=0.39) using only account information, and 36% (LR=0.36) using only Intents. The linkability an app can get from a source is roughly equal to the percentage of apps that accessed that source, except for the case of contextual information: IP, Location and Nearby APs. This is because the contextual information an app collected does not always contain effectively identifying information. For example, Yelp is mostly used at infrequent locations to find nearby restaurants, but is rarely used at consistent PoIs, such as home or office. This renders location information useless in establishing linkability with Yelp.

The effort required to aggregate two apps also differs for different linkability sources, as shown in Fig. 8. Device identifiers have *LE=0*, meaning that any two apps accessing the same device identifier can be directly aggregated without requiring control of an additional third app. Linking apps using IPC channels, such as Intents and Indirect RW, requires the adversary to control an average of 0.6 additional app as the connecting nodes. This indicates that, from an adversary's perspective, exploiting consistent identifiers is easier than building pair-wise associations.

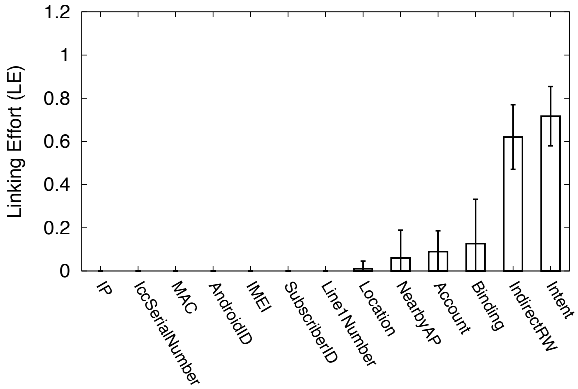


Figure 8: The (average) Linking Efforts (LE) of all the apps that are linkable due to a certain linkability source.

Category	*GLR*	*GLE*	$LR_{Facebook}$
Device	0.52 (0.13)	0.03 (0.03)	0.68 (0.12)
Personal	0.30 (0.10)	0.30 (0.11)	0.54 (0.11)
Contextual	0.20 (0.13)	0.33 (0.20)	0.44 (0.25)
IPC	0.32 (0.13)	0.78 (0.06)	0.59 (0.15)

Table 3: Linkability contributed by different categories of sources.

Breakdown by Category: We group the linkability sources into four categories — device, personal, contextual, and IPC — and study the linkability contributed by each category (see Table 3). As expected, device-specific information introduces substantial linkability and allows the adversary to conduct across-app aggregation effortlessly. Surprisingly, the other three categories of linkability sources also introduce considerable linkability. In particular, only using fuzzy contextual information, an adversary can link more than 40% of the installed apps to Facebook, the user's real identity. This suggests the naive solution of anonymizing device ids is not enough, and hence a comprehensive solution is needed to make a trade-off between app functionality and privacy.

4.3 Functional Analysis

Device identifiers (IMEI, Android ID, MAC) introduce vast amount of linkability. We manually went through 162 mobile apps that request these device-specific identifiers, but could rarely identify any explicit functionality that requires accessing the actual identifier. In fact, for the majority of these apps, their functionalities are device-independent, and therefore independent of device IDs. This indicates that device-specific identifier can be obfuscated across apps without noticeable loss of app functionality. The only requirement for device ID is that it should be unique to each device.

As to personal information (Account Number, Phone

```
<?xml version=’1.0’ encoding=’utf-8’ standalone=’yes’ ?>
<map>
    <long name="timestamp" value="1419049777098" />
    <long name="t2" value="1419049776889" />
    <string name="UTDID">VJT7MTV268gDACiZN6xEh8af</string>
    <string name="DID">356565055348652</string>
    <long name="S" value="1634341681" />
    <string name="SI">310260981039000</string>
    <string name="EI">356565055348652</string>
</map>
```

Figure 9: Real-world example of indirect RW: an app (fm.qingting.qradio) writes user identifiers to an xml file in SD card which was later read by three other apps. This file contains the IMEI (DID) and SubscriberID (SI) of the user.

Number, Installed Apps, etc.), we also observed many unexpected accesses that resulted in unnecessary linkability. We found that many apps that request account information collected all user accounts even when they only needed one to function correctly; many apps request access to phone number even when it is unrelated to their app functionalities. Since the legitimacy of a request depends both on the user's functional needs and the specific app context, end-users should be prompted about the access and make the final decision.

The linkability introduced by contextual information (Location, Nearby AP) also requires better regulation. Many apps request permission for precise location, but not all of them actually need it to function properly. In many scenarios, apps only require coarse-grained location information and shouldn't reveal any identifying points of interest (PoIs). Nearby AP information, which is only expected to be used by Wi-Fi tools/managing apps, is also abused for other purposes. We noticed that many apps frequently collect Nearby AP information to build an internal mapping between locations and access points (APs). For example, we found that even if we turn off all system location services, WeChat (an instant messaging app) can still infer the user's location only with Nearby AP information. To reduce the linkability introduced by these unexpected usages, the users should have finer-grained control on when and how the contextual information can be used.

Moreover, we found that IPC channels can be exploited in various ways to establish linkability across apps. Apps can establish linkabililty using Intents, sharing and aggregating app-specific information. For instance, we observed that WeChat receives Intents from three different apps right after their installations, reporting their existence on the same device. Apps can also establish linkability with each other via service binding. For example, both AdMob and Facebook allow an app to bind to its service and exchanging the user identifier, completely bypassing the system permissions and controls. Apps can also establish linkabililty through Indirect RW, by writing and reading the same persistent file. Fig. 9 shows a real-world example: an app (fm.qingting.qradio) writes user identifiers to an xml file in the SD card which was later read by three other apps. The end-user should be promptly warned about these unexpected communications across apps to reduce unnecessary linkability.

5 LinkDroid: A Practical Countermeasure

Based on our observation and findings on linkability across real-world apps, we propose a practical countermeasure, `LinkDroid`, on top of DLG. We first introduce the basic design principle of `LinkDroid` and its three major privacy-enhancing features: *install-time obfuscation*, *runtime linkability monitoring*, and *unlinkable mode support*. We then evaluate the effectiveness of `LinkDroid` with the same set of participants as in our measurement study.

5.1 Design Overview

`LinkDroid` is designed with practicality in mind. Numerous extensions, paradigms and ecosystems have been proposed for mobile privacy, but access control (runtime for iOS and install-time for Android) is the only deployed mechanism. `LinkDroid` adds a new dimension to access control on smartphone devices. Unlike existing approaches that check if some app behavior poses direct privacy threats, `LinkDroid` warns users about how it implicitly builds the linkability across apps. This helps users reduce unnecessary links introduced by abusing OS-level information and IPCs, which happens frequently in reality as our measurement study indicated.

As shown in Fig. 10, `LinkDroid` provides runtime monitoring and mediation of linkability by

- monitoring and intercepting app behaviors that may introduce linkability (including interactions with various system services, content providers, shared external storage and other apps);
- querying a standalone linkability service to get the user's decision regarding this app behavior;
- prompting the user about the potential risk if the user has not yet made a decision, getting his decision and updating the linkability graph (DLG).

We have already described in Section 3.4 how to instrument the Android framework to build the monitoring components (corresponding to boxes A, B, C, D in Fig. 10). In this section, we focus on how the linkability service operates.

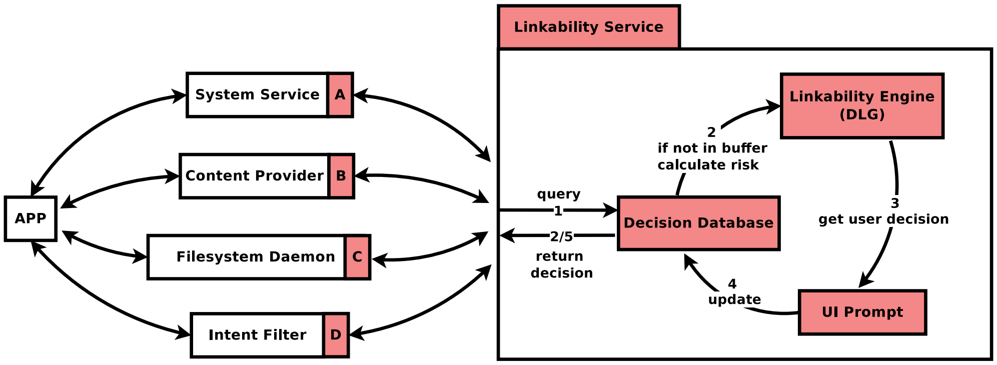


Figure 10: An overview of `LinkDroid`. Shaded areas (red) represent the parts we need to extend/add in Android. (We already explained how to extend A, B, C and D in Section 3.4.)

5.2 Install-Time Obfuscation

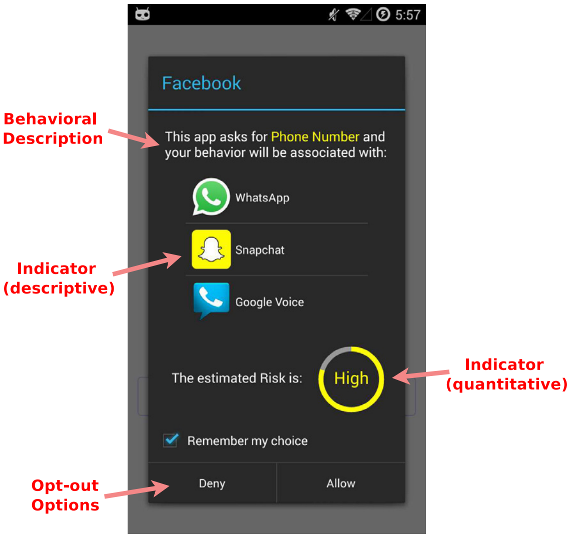
As mentioned earlier, app functionalities are largely independent of device identifiers. This allows us to obfuscate these identifiers and cut off many unnecessary edges in the DLG. In our case, the list of device identifiers includes IMEI, Android ID and MAC. Every time an app gets installed, the linkability service receives the app's uid and then generates a random mask code for it. The mask code together with the types of obfuscated device identifiers will be pushed into the decision database. This way, when an app a tries to fetch the device identifier of a certain type t, it will only get a hash of the real identifier salted with the app-specific mask code:

$$ID_t^a = hash(ID_t + mask_a).$$

Note that we do this at install-time instead of during each session because we still want to guarantee the relative consistency of the device identifiers within each app. Otherwise, it will let the app think the user is switching to a different device and trigger some security/verification mechanisms. The user can always cancel this default obfuscation in the privacy manager (Fig. 12) if he finds it necessary to reveal real device identifiers to certain apps.

5.3 Runtime Linkability Monitoring

Except for device-specific identifiers, obfuscating other sources of linkability is likely to interfere with the app functionalities. Whether there is a functional interference or not is highly user-specific and context-dependent. To make a useful trade-off, the user should be involved in this decision-making process. Here, `LinkDroid` provides just-in-time prompts before an edge creates in the DLG. Specifically, if the linkability service could not find an existing decision regarding some app behavior, it will issue the user a prompt, informing him: 1) what app behavior triggers the prompt; 2) what's the quantitative risk of allowing this behavior; and 3) what're the opt-out options. Fig. 11 gives an illustrative example of the UI of the prompt.



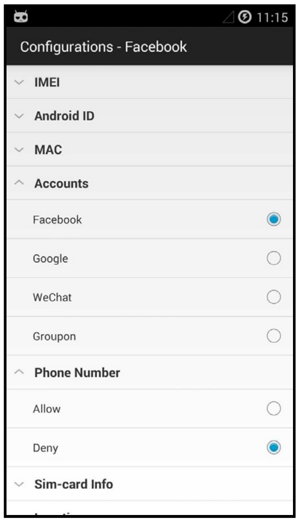
This app asks for Phone Number and your behavior will be associated with:

The estimated Risk is: High

☑ Remember my choice

Figure 11: The UI prompt of `LinkDroid`'s runtime access control, consisting of a behavioral description, descriptive and quantitative risk indicators, and opt-out options.

Description of App Behavior Before the user can make a decision, he first needs to know what app behavior triggers the prompt. Basically, we report two types of description: access to OS-level information and cross-app communications. To help the user understand the situation, we use a high-level descriptive language instead of the exact technical terms. For example, when an app tries to access Subscriber ID or IccSerialNumber, we report that "App X asks for sim-card information." When an app tries to send Intents to other apps, we report "App X tries to share content with App Y". During our experiments with real users (introduced later in the evaluation), 11 out of the 13 participants find these descriptions clear



11:15

Configurations - Facebook

IMEI

Android ID

MAC

Accounts

Facebook ●

Google ○

WeChat ○

Groupon ○

Phone Number

Allow ○

Deny ●

Sim-card Info

Figure 12: LinkDroid provides a centralized linkability manager. The user can review and modify all of his previous decisions regarding each app.

and informative.

Risk Indicator LinkDroid reports two types of risk indicators to users: one is descriptive and the other is quantitative. The descriptive indicator tells what apps will be directly linkable to an app if the user allows its current behavior. By 'directly linkable,' we mean without requiring a third app as the connecting nodes. The quantitative indicator, on the other hand, reflects the influence on the overall linkability of the running app, including those apps that are not directly linkable to it. Here, the overall linkability is reported as a combination of the linking ratio (LR) and linking effort (LE):

$$L_a = LR_a \times e^{-LE_a}.$$

The quantitative risk indicator is defined as ΔL_a. A user will be warned of a larger risk if the total number of linkable apps significantly increases, or the average linking effort decreases substantially. We transform the quantitative risk linearly into a scale of 4 and report the risk as Low, Medium, High, and Severe.

Opt-out Options In each prompt, the user has at least two options: Allow or Deny. If the user chooses Deny, LinkDroid will obfuscate the information this app tries to get or shut down the communication channel this app requests. For some types of identifying information, such as Accounts and Location, we provide finer-grained trade-offs. For Location, the user can select from zip-code level (1km) or city-level (10km) precision; for Accounts, the user can choose which specific account he wants to share instead of exposing all his accounts. LinkDroid also allows the user to set up a VPN (Virtual Private Network) service to anonymize network identifiers. When the user switches from a cellular network to Wi-Fi, LinkDroid will automatically initialize the VPN service to hide the user's public IP. This may incur additional energy consumption and latency (see Section 5.5). All choices made by the user will be stored in the decision database for future reuse. We provide a centralized privacy manager such that the user can review and change all previously made decisions (see Fig. 12).

5.4 Unlinkable Mode

Once a link is established in DLG, it cannot be removed. This is because once a piece of identifying information is accessed or a communication channel is established, it can never be revoked. However, the user may sometimes want to perform privacy-preserving tasks which have no interference with the links that have already been introduced. For example, when the user wants to write an anonymous post in Reddit, he doesn't want it to be linkable with any of his previous posts as well as other apps. LinkDroid provides an unlinkable mode to meet such a need. The user can start an app in unlinkable mode by pressing its icon for long in the app launcher. A new uid as well as isolated storage will be allocated to this unlinkable app instance. By default, access to all OS-level identifying information and inter-app communications will be denied. This way, LinkDroid creates the illusion that this app has just been installed on a brand-new device. The unlinkable mode allows LinkDroid to provide finer-grained (session-level) control, unlinking only a certain set of app sessions.

5.5 Evaluation

We evaluate LinkDroid in terms of its overheads in usability and performance, as well as its effectiveness in reducing linkability. We replay the traces of the 13 participants of our measurement study (see Section 4), prompt them about the privacy threat and ask for their decisions. This gives us the exact picture of the same set of users using LinkDroid during the same period of time. We instruct the user to make a decision in the most conservative way: the user will Deny a request only when he believes the prompted app behavior is not applicable to any useful scenario; otherwise, he will Accept the request.

The overhead of LinkDroid mainly comes from two parts: the usability burden of dealing with UI prompts and the performance degradation of querying the linkability service. Our experimental results show that, on average, each user was prompted only 1.06 times per day during the 47-day period. The performance degradation

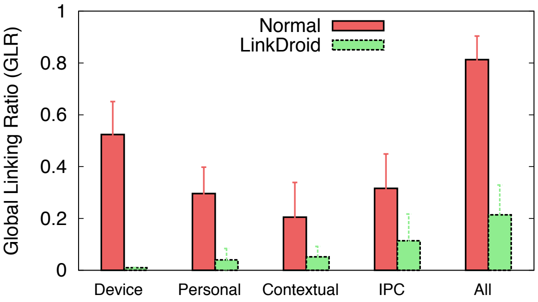


Figure 13: The Global Linking Ratio (GLR) of different categories of sources before and after using LinkDroid.

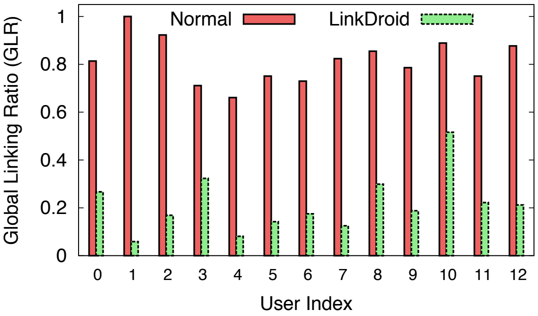


Figure 14: The Global Linking Ratio (GLR) of different users before and after using LinkDroid.

introduced by the linkability service is also marginal. It only occurs when apps access certain OS-level information or conduct cross-app IPCs. These sensitive operations happened rather infrequently — once every 12.7 seconds during our experiments. These results suggest that LinkDroid has limited impact on system performance and usability.

We found that after applying LinkDroid, the Global Linking Ratio (GLR) dropped from 81% to 21%. Fig. 13 shows the breakdown of linkability drop in different categories of sources. The majority of the remaining linkability comes from inter-app communications, most of which are genuine from the user's perspective. Not only fewer apps are linkable, LinkDroid also makes it harder for an adversary to aggregate information from two linkable apps. The Global Linking Effort (GLE) increases significantly after applying LinkDroid: from 0.22 to 0.68. Specifically, the percentage of apps that are directly linkable to Facebook dropped from 86% to 18%. Fig. 15 gives an illustrative example of how DLG changes after applying LinkDroid. We also noticed that that the effectiveness of LinkDroid differs across users, as shown in Fig. 14. In general, LinkDroid is more effective for the users who have diverse mobility patterns, are cautious about sharing information across apps and/or maintain different accounts for different services.

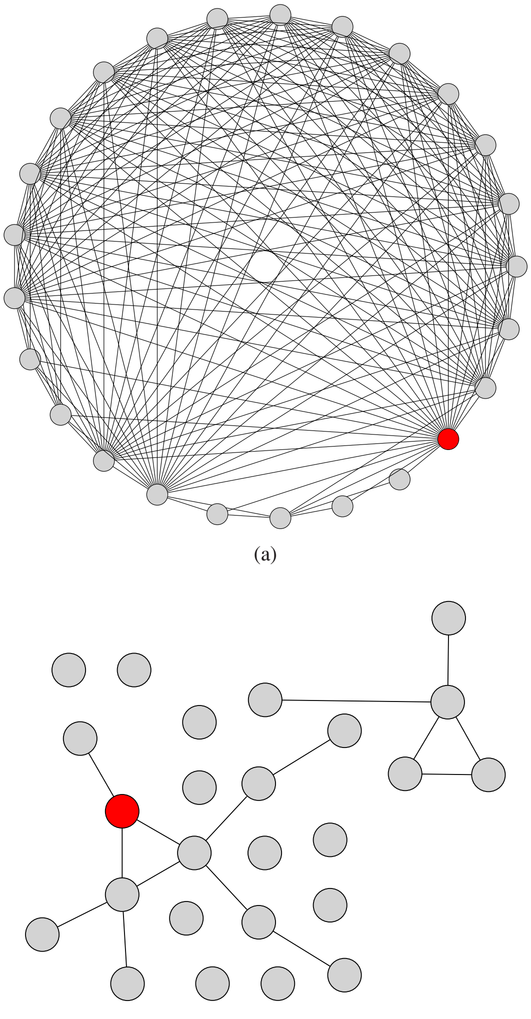


Figure 15: DLG of a representative user before (a) and after (b) applying LinkDroid. Red circle represents the Facebook app.

LinkDroid takes VPN as a plug-in solution to obfuscate network identifiers. The potential drawback of using VPN is its influence on device energy consumption and network latency. We measured the device energy consumption of using VPN on a Samsung Galaxy 4 device, with Monsoon Power Monitor. Specifically, we tested two network-intensive workloads: online videos and browsing. We observed a 5% increase in energy consumption for the first workload, and no observable difference for the second. To measure the network latency, we measured the ping time (average of 10 trials) to Alexa Top 20 domains and found a 13% increase (17ms). These results indicate that the overhead of using VPN on smartphone device is noticeable but not significant. Seven of 13 participants in our evaluation were willing to use VPN services to achieve better privacy.

We interviewed the 13 participants after the experiments. Questions are designed on a scale of 1 to 5 and a score of 4 or higher is regarded as "agree." Eleven of the participants find the UI prompt informative and clear and nine are willing to use LinkDroid on a daily basis to inform them about the risk and provide opt-out options. However, these responses might not be representative due to the limited size and diversity of the participants. We also noticed that users care a lot about the linkability of sensitive apps, such as Snapchat and Facebook. Some participants clearly state that they do not want any app to be associated with the Facebook app, except for very necessary occasions. This also supports the rationale behind the design of LinkDroid's unlinkable mode.

6 Related Work

There have been other proposals [7,17] which also address the privacy threats of information aggregation by mobile apps. They shift the responsibility of information personalization and aggregation from mobile apps to the mobile OS or trusted cloud providers, requiring re-development of mobile apps and extensive modifications on the entire mobile ecosystem. In contrast, LinkDroid is a client-side solution which is compatible with existing ecosystem — it focuses on characterizing the threat in current mobile ecosystem and making a practical trade-off, instead of proposing new computation (advertising) paradigm.

Existing studies investigated linkability under several domain-specific scenarios. Arvind *et al.* [19] showed that a user's profile in Netflix can be effectively linked to his in IMDB, using long-tailed (unpopular) movies. Sebastian *et al.* [16] described how to link the profiles of the same user in different social networks using friends topologies. This type of linkability is restricted to a small scope, and may only exist across different apps in the same domain. Here, we focus on the linkability that are domain-independent and ubiquitous to all apps, regardless of the type and semantics of each app.

The capability of advertising agency on conducting profiling and aggregation has been extensively studied [12,23]. Various countermeasures have been proposed, such as enforcing finer-grained isolation between ad library and the app [21,22], or adopting a privacy-preserving advertising paradigm [4]. However, unlike LinkDroid, they only consider a very specific and restricted scenario — advertising library — which involves few functional trade-offs. LinkDroid, instead, introduces a general linkability model, considers various sources of linkability and suits a diverse set of adversaries.

There have also been numerous studies on information access control on smartphone [6,8,9,13,14,20,24]. Many of these studies have already proposed to provide apps with fake identifiers and other types of sensitive information [13,20,27]. These studies focus on the explicit privacy concern of accessing and leaking sensitive user information, by malicious mobile apps or third-party libraries. Our work addresses information access control from a very different perspective, investigating the implicit linkability introduced by accessing various OS-level information and IPC channels.

Many modern browsers provide a private (incognito) mode. These are used to defend against local attackers, such as users sharing the same computer, from stealing cookies or browse history from each other [2]. This is inherently different from LinkDroid's *unlinkable mode* which targets unregulated aggregation by remote attackers.

7 Discussion

In this paper, we proposed a new metric, *linkability*, to quantify the ability of different apps to link and aggregate their usage behaviors. This metric, albeit useful, is only a coarse upper-bound of the actual privacy threat, especially in the case of IPCs. Communication between two apps does not necessarily mean that they have conducted, or are capable of conducting, information aggregation. However, deciding on the actual intention of each IPC is by itself a difficult task. It requires an automatic and extensible way of conducting semantic introspection on IPCs, and is a challenging research problem on its own.

LinkDroid aims to reduce the linkability introduced covertly without the user's consent or knowledge — it couldn't and doesn't try to eliminate the linkability explicitly introduced by users. For example, a user may post photos of himself or exhibit very identifiable purchasing behavior in two different apps, thus establishing linkability. This type of linkability is app-specific, domain-dependent and beyond the control of LinkDroid. Identifiability or linkability of these domain-specific usage behaviors are of particular interest to other areas, such as anonymous payment [25], anonymous query processing [18] and data anonymization techniques.

The list of identifying information we considered in this paper is well-formatted and widely-used. These ubiquitous identifiers contribute the most to information aggregation, since they are persistent and consistent across different apps. We didn't consider some uncommon identifiers, such as walking patterns and microphone signatures, because we haven't yet observed any real-world adoption of these techniques by commercial apps. However, LinkDroid can easily include other types of identifying information, as long as a clear definition is given.

DLG introduces another dimension — linkability — to privacy protection on mobile OS and has some other potential usages. For example, when the user wants to perform a certain task in Android and has multiple optional apps, the OS can recommend him to choose the app which is the least linkable with others. We also noticed some interesting side-effect of LinkDroid's unlinkable mode. Since unlinkable mode allows users to enjoy finer-grained (session-level) unlinkability, it can be used to stop a certain app from continuously identifying a user. This can be exploited to infringe the benefits of app developers in the case of copyright protection, etc. For example, NYTimes only allows an unregistered user to read up to 10 articles every month. However, by restarting the app in unlinkable mode in each session, a user can stop NYTimes from linking himself across different sessions and bypass this quota restriction.

8 Conclusion

In this paper, we addressed the privacy threat of unregulated aggregation from a new perspective by monitoring, characterizing and reducing the underlying linkability across apps. This allows us to measure the potential threat of unregulated aggregation during runtime and promptly warn users of the associated risks. We observed how real-world apps abuse OS-level information and IPCs to establish linkability, and proposed a practical countermeasure, LinkDroid. It provides runtime monitoring and mediation of linkability across apps, introducing a new dimension to privacy protection on mobile device. Our evaluation on real users has shown that LinkDroid is effective in reducing the linkability across apps and only incurs marginal overheads.

Acknowledgements

The work reported in this paper was supported in part by the NSF under grants 0905143 and 1114837, and the ARO under W811NF-12-1-0530.

References

[1] 2013: a look back at the year in acquisitions. http://vator.tv/news/2013-12-07-2013-a-look-back-at-the-year-in-acquisitions.

[2] Aggarwal, G., Bursztein, E., Jackson, C., and Boneh, D. An analysis of private browsing modes in modern browsers. In *Proceedings of the 19th USENIX conference on Security* (2010), USENIX Association, pp. 6–6.

[3] Angry birds and 'leaky' phone apps targeted by nsa and gchq for user data. http://www.theguardian.com/world/2014/jan/27/nsa-gchq-smartphone-app-angry-birds-personal-data.

[4] Backes, M., Kate, A., Maffei, M., and Pecina, K. Obliviad: Provably secure and practical online behavioral advertising. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2012), SP '12, IEEE Computer Society, pp. 257–271.

[5] Bamis, A., and Savvides, A. Lightweight Extraction of Frequent Spatio-Temporal Activities from GPS Traces. In *IEEE Real-Time Systems Symposium* (2010), pp. 281–291.

[6] Bugiel, S., Heuser, S., and Sadeghi, A.-R. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *Presented as part of the 22nd USENIX Security Symposium* (Berkeley, CA, 2013), USENIX, pp. 131–146.

[7] Davidson, D., and Livshits, B. Morepriv: Mobile os support for application personalization and privacy. Tech. rep., MSR-TR, 2012.

[8] Egele, M., Kruegel, C., Kirda, E., and Vigna, G. Pios: Detecting privacy leaks in ios applications. In *NDSS* (2011).

[9] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI* (2010), vol. 10, pp. 255–270.

[10] Fawaz, K., and Shin, K. G. Location privacy protection for smartphone users. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 239–250.

[11] Golle, P., and Partridge, K. On the anonymity of home-/work location pairs. In *Proceedings of Pervasive '09* (Berlin, Heidelberg, 2009), Springer-Verlag, pp. 390–397.

[12] Han, S., Jung, J., and Wetherall, D. A study of third-party tracking by mobile apps in the wild. Tech. rep., UW-CSE, 2011.

[13] Hornyack, P., Han, S., Jung, J., Schechter, S., and Wetherall, D. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and Communications Security* (2011), ACM, pp. 639–652.

[14] Jeon, J., Micinski, K. K., Vaughan, J. A., Fogel, A., Reddy, N., Foster, J. S., and Millstein, T. Dr. android and mr. hide: fine-grained permissions in android applications. In *Proceedings of Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* (2012), ACM, pp. 3–14.

[15] Krumm, J. Inference attacks on location tracks. In *Proceedings of the 5th international conference on Pervasive computing* (Berlin, Heidelberg, 2007), PERVASIVE'07, Springer-Verlag, pp. 127–143.

[16] Labitzke, S., Taranu, I., and Hartenstein, H. What your friends tell others about you: Low cost linkability of social network profiles. In *Proc. 5th International ACM Workshop on Social Network Mining and Analysis, San Diego, CA, USA* (2011).

[17] Lee, S., Wong, E. L., Goel, D., Dahlin, M., and Shmatikov, V. πbox: a platform for privacy-preserving apps. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation* (2013), USENIX Association, pp. 501–514.

[18] Mokbel, M. F., Chow, C.-Y., and Aref, W. G. The new casper: query processing for location services without compromising privacy. In *Proceedings of the 32nd international conference on Very large data bases* (2006), VLDB Endowment, pp. 763–774.

[19] NARAYANAN, A., AND SHMATIKOV, V. Robust de-anonymization of large sparse datasets. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on* (2008), IEEE, pp. 111–125.

[20] NAUMAN, M., KHAN, S., AND ZHANG, X. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security* (2010), ACM, pp. 328–332.

[21] PEARCE, P., FELT, A. P., NUNEZ, G., AND WAGNER, D. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security* (2012), ACM, pp. 71–72.

[22] SHEKHAR, S., DIETZ, M., AND WALLACH, D. S. Adsplit: separating smartphone advertising from applications. In *Proceedings of the 21st USENIX conference on Security symposium* (2012), USENIX Association, pp. 28–28.

[23] STEVENS, R., GIBLER, C., CRUSSELL, J., ERICKSON, J., AND CHEN, H. Investigating user privacy in android ad libraries. *IEEE Mobile Security Technologies (MoST)* (2012).

[24] TRIPP, O., AND RUBIN, J. A bayesian approach to privacy enforcement in smartphones. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2014), SEC'14, USENIX Association, pp. 175–190.

[25] WEI, K., SMITH, A. J., CHEN, Y.-F., AND VO, B. Whopay: A scalable and anonymous payment system for peer-to-peer environments. In *Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on* (2006), IEEE, pp. 13–13.

[26] XIA, N., SONG, H. H., LIAO, Y., ILIOFOTOU, M., NUCCI, A., ZHANG, Z.-L., AND KUZMANOVIC, A. Mosaic: quantifying privacy leakage in mobile networks. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM* (2013), ACM, pp. 279–290.

[27] Xprivacy - the ultimate, yet easy to use, privacy manager for android. `https://github.com/M66B/XPrivacy#xprivacy`.

[28] XU, R., SAÏDI, H., AND ANDERSON, R. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the 21st USENIX conference on Security symposium* (2012), USENIX Association, pp. 27–27.

[29] ZANG, H., AND BOLOT, J. Anonymization of location data does not work: a large-scale measurement study. In *Proceedings of MobiCom '11* (New York, NY, USA, 2011), ACM, pp. 145–156.

PowerSpy: Location Tracking using Mobile Device Power Analysis

Yan Michalevsky, Aaron Schulman,
Gunaa Arumugam Veerapandian and Dan Boneh
Computer Science Department
Stanford University

Gabi Nakibly
National Research and Simulation Center
Rafael Ltd.

Abstract

Modern mobile platforms like Android enable applications to read aggregate power usage on the phone. This information is considered harmless and reading it requires no user permission or notification. We show that by simply reading the phone's aggregate power consumption over a period of a few minutes an application can learn information about the user's location. Aggregate phone power consumption data is extremely noisy due to the multitude of components and applications that simultaneously consume power. Nevertheless, by using machine learning algorithms we are able to successfully infer the phone's location. We discuss several ways in which this privacy leak can be remedied.

1 Introduction

Our phones are always within reach and their location is mostly the same as our location. In effect, tracking the location of a phone is practically the same as tracking the location of its owner. Since users generally prefer that their location not be tracked by arbitrary 3rd parties, all mobile platforms consider the device's location as sensitive information and go to considerable lengths to protect it: applications need explicit user permission to access the phone's GPS and even reading coarse location data based on cellular and WiFi connectivity requires explicit user permission.

In this work we show that despite these restrictions applications can covertly learn the phone's location. They can do so using a seemingly benign sensor: the phone's power meter that measures the phone's power consumption over a period of time. Our work is based on the observation that the phone's location significantly affects the power consumed by the phone's cellular radio. The power consumption is affected both by the distance to the cellular base station to which the phone is currently attached (free-space path loss) and by obstacles, such as buildings and trees, between them (shadowing). The closer the phone is to the base station and the fewer obstacles between them the less power the phone consumes. The strength of the cellular signal is a major factor affecting the power used by the cellular radio [29]. Moreover, the cellular radio is one of the most dominant power consumers on the phone [14].

Suppose an attacker measures in advance the power profile consumed by a phone as it moves along a set of known routes or in a predetermined area such as a city. We show that this enables the attacker to infer the target phone's location over those routes or areas by simply analyzing the target phone's power consumption over a period of time. This can be done with no knowledge of the base stations to which the phone is attached.

A major technical challenge is that power is consumed simultaneously by many components and applications on the phone in addition to the cellular radio. A user may launch applications, listen to music, turn the screen on and off, receive a phone call, and so on. All these activities affect the phone's power consumption and result in a very noisy approximation of the cellular radio's power usage. Moreover, the cellular radio's power consumption itself depends on the phone's activity, as well as the distance to the base-station: during a voice call or data transmission the cellular radio consumes more power than when it is idle. All of these factors contribute to the phone's power consumption variability and add noise to the attacker's view: the power meter only provides aggregate power usage and cannot be used to measure the power used by an individual component such as the cellular radio.

Nevertheless, using machine learning, we show that the phone's aggregate power consumption over time completely reveals the phone's location and movement. Intuitively, the reason why all this noise does not mislead our algorithms is that the noise is not correlated with the phone's location. Therefore, a sufficiently long power measurement (several minutes) enables the learning algorithm to "see" through the noise. We refer to power consumption measurements as time-series and use methods for comparing time-series to obtain classification and pattern matching algorithms for power consumption profiles.

In this work we use machine learning to identify the routes taken by the victim based on previously collected power consumption data. We study three types of user tracking goals:

1. **Route distinguishability:** First, we ask whether an attacker can tell what route the user is taking among a fixed set of possible routes.
2. **Real-time motion tracking:** Assuming the user is taking a certain known route, we ask whether an attacker can identify her location along the route and track the device's position on the route in real-time.
3. **New route inference:** Finally, suppose a user is moving along an arbitrary (long) route. We ask if an attacker can learn the user's route using the previously measured power profile of many (short) road segments in that area. The attacker composes the power profile of the short road segments to identify the user's route and location at the end of the route.

We emphasize that our approach is based on measuring the phone's aggregate power consumption and nothing else. In particular, we do not use the phone's signal strength as this data is protected on Android and iOS devices and reading it requires user permission. In contrast, reading the phone's power meter requires no special permissions.

On Android reading the phone's aggregate power meter is done by repeatedly reading the following two files:

```
/sys/class/power_supply/battery/voltage_now
/sys/class/power_supply/battery/current_now
```

Over a hundred applications in the Play Store access these files. While most of these simply monitor battery usage, our work shows that all of them can also easily track the user's location.

Our contributions. Our work makes the following contributions:

- We show that the power meter available on modern phones can reveal potentially private information.
- We develop the machine learning techniques needed to use data collected from the power meter to infer location information. The technical details of our algorithms are presented in sections 4, 5 and 6, followed by experimental results.
- In sections 8 and 9 we discuss potential continuation to this work, as well as defenses to prevent this type of information leakage.

2 Threat Models

We assume a malicious application is installed on the victim's device and runs in the background. The application has no permission to access the GPS or any other location data such as the cellular or WiFi components. In particular, the application has no permission to query the identity of visible cellular base stations or the SSID of visible WiFi networks.

We only assume access to power data (which requires no special permissions on Android) and permission to communicate with a remote server. Network connectivity is needed to generate dummy low rate traffic to prevent the cellular radio from going into low power state. In our setup we also use network connectivity to send data to a central server for processing. However, it may be possible to do all processing on the phone.[1]

As noted earlier, the application can only read the *aggregate* power consumed by the phone. It cannot measure the power consumed by the cellular radio alone. This presents a significant challenge since many components on the phone consume variable amounts of power at any given time. Consequently, all the measurements are extremely noisy and we need a way to "see" though the noise.

To locate the phone, we assume the attacker has prior knowledge of the area or routes through which the victim is traveling. This knowledge allows the attacker to measure the power consumption profile of different routes in that area in advance. Our system correlates this data with the phone's measured power usage and we show that, despite the noisy measurements, we are able to correctly locate the phone. Alternatively, as for many other machine learning cases, the training data can also be collected after obtaining the unlabeled query data. For instance, an attacker obtained a power consumption profile of a user, the past location of whom it is extremely important to determine. She can still collect, after the fact, reference profiles for a limited area in which the user has likely been driving and carry out the attack.

For this to work we need the tracked phone to be moving by a car or a bus while being tracked. Our system cannot locate a phone that is standing still since that only provides the power profile for a single location. We need multiple adjacent locations for the attack to work.

Given the resources at our disposal, the focus of this work is on locating a phone among a set of local routes in a pre-determined area. A larger effort is needed to scale the system to cover the entire world by pre-measuring the power profile of all road segments worldwide. Nevertheless, our localized experiments already show that tracking users who follow a daily routine is quite possible. For example, a mobile device owner might choose one of a small number of routes to get from home to work. The

[1] It is important to mention here that while a network access permission will appear in the permission list for an installed application, it does not currently appear in the list of required permissions prior to application installation.

system correctly identifies what route was chosen and in real-time identifies where the phone is along that route. This already serves as a cautionary note about the type of information that can be leaked by a seemingly innocuous sensor like the power meter.

We note that scaling the system to cover worldwide road segments can be done by crowd-sourcing: a popular app, or perhaps even the core OS, can record the power profile of streets traveled by different users and report the results to a central server. Over time the resulting dataset will cover a significant fraction of the world. On the positive side, our work shows that service providers can legitimately use this dataset to improve the accuracy of location services. On the negative side, tracking apps can use it to covertly locate users. Given that all that is required is one widespread application, many actors in the mobile space are in a position to build the required dataset of power profiles and use it as they will.

3 Background

In this section we provide technical background on the relation between a phone's location and its cellular power consumption. We start with a description of how location is related to signal strength, then we describe how signal strength is related to power consumption. Finally, we present examples of this phenomenon, and we demonstrate how obtaining access to power measurements could leak information about a phone's location.

3.1 Location affects signal strength and power consumption

Distance to the base station is the primary factor that determines a phone's signal strength. The reason for this is, for signals propagating in free space, the signal's power loss is proportional to the square of the distance it travels over [11]. Signal strength is not only determined by path loss, it is also affected by objects in the signal path, such as trees and buildings, that attenuate the signal. Finally, signal strength also depends on multi-path interference caused by objects that reflect the radio signal back to the phone through various paths having different lengths.

In wireless communication theory signal strength is often modeled as random variation (e.g., log-normal shadowing [11]) to simulate many different environments[2]. However, in one location signal strength can be fairly consistent as base stations, attenuators, and reflectors are mostly stationary.

A phone's received signal strength to its base station affects its cellular modem power consumption. Namely, phone cellular modems consume less instantaneous power when transmitting and receiving at high signal strength compared to low signal strength. Schulman et. al. [29] observed this phenomenon on several different cellular devices operating on different cellular protocols. They showed that communication at a poor signal location can result in a device power draw that is 50% higher than at a good signal location.

The primary reason for this phenomenon is the phone's power amplifier used for transmission which increases its gain as signal strength drops [11]. This effect also occurs when a phone is only receiving packets. The reason for this is cellular protocols which require constant transmission of channel quality and acknowledgments to base stations.

3.2 Power usage can reveal location

The following results from driving experiments demonstrate the potential of leaking location from power measurements.

We first demonstrate that signal strength in each location on a drive can be static over the course of several days. We collected signal strength measurements from a smartphone once, and again several days later. In Figure 1 we plot the signal strength observed on these two drives. In this figure it is apparent that (1) the segments of the drive where signal strength is high (green) and low (red) are in the same locations across both days, and (2) that the progression of signal strength along the drive appears to be a unique irregular pattern.

Next, we demonstrate that just like signal strength, power measurements of a smartphone, while it communicates, can reveal a stable, unique pattern for a particular drive. Unlike signal strength, power measurements are less likely to be stable across drives because power depends on how the cellular modem reacts to changing signal strength: a small difference in signal strength between two drives may put the cellular modem in a mode that has a large difference in power consumption. For example, a small difference in signal strength may cause a phone to hand-off to a different cellular base station and stay attached to it for some time (Section 3.3).

Figure 2 shows power measurements for two Nexus 4 phones in the same vehicle, transmitting packets over their cellular link, while driving on the same path. The power consumption variations of the Nexus 4 phones are similar, indicating that power measurements can be mostly stable across devices.

Finally, we demonstrate that power measurements could be stable across different models of smartphones. This stability would allow an attacker to obtain a reference power measurement for a drive without using the same phone as the victim's. We recorded power

[2] Parameters of the model can be calibrated to better match a specific environment of interest.

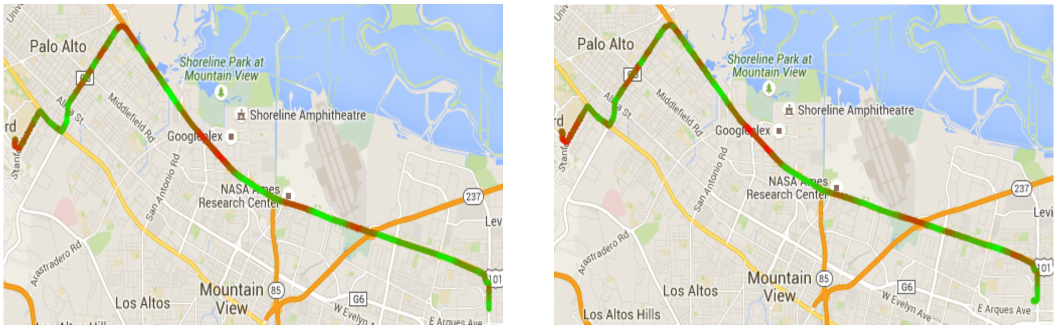


Figure 1: Signal strength profiles measured on two different days are stable.

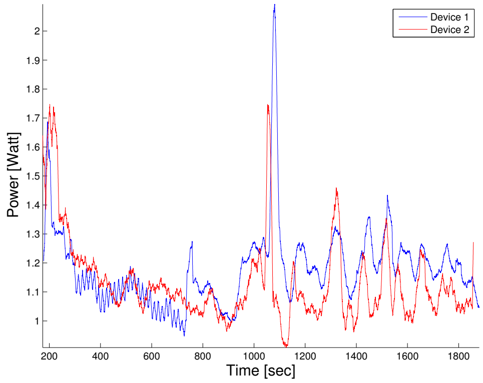


Figure 2: For two phones of the same model, power variations on the same drive are similar.

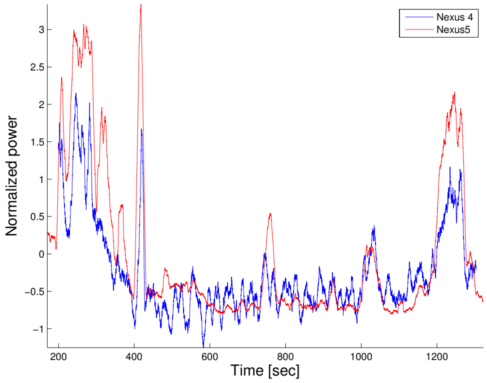


Figure 3: For two different phone models, power variations on the same drive are similar.

measurements, while transmitting packets over cellular, using two different smartphone models (Nexus 4 and Nexus 5) during the same ride, and we aligned the power samples, according to absolute time.

The results presented in Figure 3 indicate that there is similarity between different models that could allow one model to be used as a reference for another. This experiment serves as a proof of concept: we leave further evaluation of such an attack scenario, where the attacker and victim use different phone models, to future work. In this paper, we assume that the attacker can obtain reference power measurements using the same phone model as the victim.

3.3 Hysteresis

A phone attaches to the base station having the strongest signal. Therefore, one might expect that the base station to which a phone is attached and the signal strength will be the same in one location. Nonetheless, it is shown in [29] that signal strength can be significantly different at a location based on how the device arrived there, for example, the direction of arrival. This is due to the hysteresis algorithm used to decide when to hand-off to a new base station. A phone hands-off from its base station only when its received signal strength dips below the signal strength from the next base station by more than a given threshold [26]. Thus, two phones that reside in the same location can be attached to two different base stations.

Hysteresis has two implications for determining a victim's location from power measurements: (1) an attacker can only use the same direction of travel as a reference power measurement, and (2) it will complicate inferring new routes from power measurements collected from individual road segments (Section 6).

3.4 Background summary and challenges

The initial measurements in this section suggest that the power consumed by the cellular radio is a side channel that leaks information about the location of a smartphone. However, there are four significant challenges that must be overcome to infer location from the power meter. First, during the pre-measurement phase the attacker may have traveled at a different speed and encountered different stops than the target phone. Second, the attacker will have to identify the target's power profile from among many pre-collected power profiles along different routes. Third, once the attacker determines the target's path, the exact location of the target on the path may be ambiguous because of similarities in the path's power profile. Finally, the target may travel along a path that the attacker only partially covered during the pre-measurement phase: the attacker may have only pre-collected measurements for a subset of segments in the target's route. In the following sections we describe techniques that address each of these challenges and experiment with their accuracy.

4 Route distinguishability

As a warm-up we show how the phone's power profile can be used to identify what route the user is taking from among a small set of possible routes (say, 30 routes). Although we view it as a warm-up, building towards our main results, route distinguishability is still quite useful. For example, if the attacker is familiar with the user's routine then the attacker can pre-measure all the user's normal routes and then repeatedly locate the user among those routes.

Route distinguishability is a classification problem: we collected power profiles associated with known routes and want to classify new samples based on this training set. We treat each power profile as a time series which needs to be compared to other time series. A score is assigned after each comparison, and based on these scores we select the most likely matching route. Because different rides along the same route can vary in speed at different locations along the ride, and because routes having the same label can vary slightly at certain points (especially before getting to a highway and after exiting it), we need to compare profile features that can vary in time and length and allow for a certain amount of difference. We also have to compensate for different baselines in power consumption due to constant components that depend on the running applications and on differences in device models.

We use a classification method based on Dynamic Time Warping (DTW) [23], an algorithm for measuring similarity between temporal sequences that are misaligned and vary in time or speed. We compute the DTW distance[3] between the new power profile and all reference profiles associated with known routes, selecting the known route that yields the minimal distance. More formally, if the reference profiles are given by sequences $\{X\}_{i=1}^{n}$, and the unclassified profile is given by sequence Y, we choose the route i such that

$$i = \underset{i}{\operatorname{argmin}} \operatorname{DTW}(Y, X_i)$$

which is equivalent to 1-NN classification given DTW metric.

Because the profiles might have different baselines and variability, we perform the following normalization for each profile prior to computing the DTW distance: we calculate the mean and subtract it, and divide the result by the standard deviation. We also apply some preprocessing in the form of smoothing the profiles using a moving average (MA) filter in order to reduce noise and obtain the general power consumption trend, and we downsample by a factor of 10 to reduce computational complexity.

5 Real-time mobile device tracking

In this section we consider the following task: the attacker knows that a mobile user is traveling along a particular route and our objective is to track the mobile device as it is moving along the route. We do not assume a particular starting point along the route, meaning, in probabilistic terms, that our prior on the initial location is uniform. The attacker has reference power profiles collected in advance for the target route, and constantly receives new power measurements from an application installed on the target phone. Its goal is to locate the device along the route, and continue tracking it in real-time as it travels along the route.

5.1 Tracking via Dynamic Time Warping

This approach is similar to that of route distinguishability, but we use only the measurements collected up to this point, which comprise a sub-sequence of the entire route profile. We use the *Subsequence* DTW algorithm [23], rather than the classic DTW, to search a sub-sequence in a larger sequence, and return a distance measure as well as the corresponding start and end offsets.

We search for the sequence of measurements we have accumulated since the beginning of the drive in all our reference profiles and select the profile that yields the minimal DTW distance. The location estimate corresponds to the location associated with the end offset returned by the algorithm.

[3] In fact we compute a normalized DTW distance, as we have to compensate for difference in lengths of different routes - a longer route might yield larger DTW distance despite being more similar to the tested sequence.

5.2 Improved tracking via a motion model

While the previous approach can make mistakes in location estimation due to a match with an incorrect location, we can further improve the estimation by imposing rules based on a sensible motion model. We first need to know when we are "locked" on the target. For this purpose we define a similarity threshold so that if the minimal DTW distance is above this threshold, we are in a *locked* state. Once we are locked on the target, we perform a simple sanity check at each iteration: "Has the target displaced by more than X?"

If the sanity check does not pass we consider the estimate unlikely to be accurate, and simply output the previous estimate as the new estimated location. If the similarity is below the threshold, we switch to an *unlocked* state, and stop performing this sanity check until we are "locked" again. Algorithm 1 presents this logic as pseudocode.

Algorithm 1 Improved tracking using a simple motion model

```
$locked \leftarrow false$          ▷ Are we locked on the target?
while target moving do
    $loc[i], score \leftarrow estimateLocation()$
    $d \leftarrow getDistance(loc[i], loc[i-1])$
    if $locked$ and $d > MAX\_DISP$ then
        $loc[i] \leftarrow loc[i-1]$    ▷ Reuse previous estimate
    end if
    if $score > THRESHOLD$ then
        $locked \leftarrow true$
    end if
end while
```

5.3 Tracking using Optimal Subsequence Bijection

Optimal Subsequence Bijection (OSB) [17] is a technique, similar to DTW, that enables aligning two sequences. In DTW, we align the query sequence with the target sequence without skipping elements in the query sequence, thereby assuming that the query sequence contains no noise. OSB, on the other hand, copes with noise in both sequences by allowing to skip elements. A fixed jump-cost is incurred with every skip in either the query or the target sequence. This extra degree of freedom has potential for aligning noisy subsequences more efficiently in our case. In the evaluation section we present results obtained by using OSB and compare them to those obtained using DTW.

6 Inference of new routes

In Section 4 we addressed the problem of identifying the route traversed by the phone, assuming the potential routes are known in advance. This assumption allowed us to train our algorithm specifically for the potential routes. As previously mentioned, there are indeed many real-world scenarios where it is applicable. Nevertheless, in this section we set out to tackle a broader tracking problem, where the future potential routes are not explicitly known. Here we specifically aim to identify the final location of the phone after it traversed an unknown route. We assume that the area in which the mobile device owner moves is known, however the number of all possible routes in that area may be too large to practically pre-record each one. Such an area can be, for instance, a university campus, a neighborhood, a small town or a highway network.

We address this problem by pre-recording the power profiles of all the road segments within the given area. Each possible route a mobile device may take is a concatenation of some subset of these road segments. Given a power profile of the tracked device, we will reconstruct the unknown route using the reference power profiles corresponding to the road segments. The reconstructed route will enable us to estimate the phone's final location. Note that, due to the hysteresis of hand-offs between cellular base stations, a power consumption is not only dependent on the traveled road segment, but also on the previous road segment the device came from.

In Appendix A we formalize this problem as a hidden Markov model (HMM) [27]. In the following we describe a method to solve the problem using a particle filter. The performance of the algorithm will be examined in the next section.

6.1 Particle Filter

A particle filter [1] is a method that estimates the state of a HMM at each step based on observations up to that step. The estimation is done using a Monte Carlo approximation where a set of samples (particles) is generated at each step that approximate the probability distribution of the states at the corresponding steps. A comprehensive introduction to particle filters and their relation to general state-space models is provided in [28].

We implement the particle filter as follows. We denote $O^r = \{o^r_{xyz}\}$, where o^r_{xyz} is a power profile prerecorded over segment (y,z) while the segment (x,y) had been traversed just before it. We use a discrete time resolution $\tau = 3$ seconds. We denote $\Delta^{yz}_{\min}$ and $\Delta^{yz}_{\max}$ to be the minimum and maximum time duration to traverse road segment (y,z), respectively. We assume these bounds can be derived from prerecordings of the segments. At each iteration i we have a sample set of N routes $P_i = \{(Q,T)\}$. The initial set of routes P_0 are chosen according to Π. At each step, we execute the following algorithm:

Algorithm 2 Particle filter for new routes estimation

```
for all route $p$ in $P$ do
    $t_{\text{end}} \leftarrow$ end time of $p$
    $(x,y) \leftarrow$ last segment of $p$
    $z \leftarrow$ next intersection to traverse (distributed by $A$)
    $W_p \leftarrow \min_{\substack{t\in[\Delta^{yz}_{\text{min}},\Delta^{yz}_{\text{max}}] \\ o^r_{xyz}\in O^r_{xyz}}} \left\{\text{DTW}(O_{[t_{\text{end}},t_{\text{end}}+t]}, o^r_{xyz})\right\}$
    $p \leftarrow p||(y,z)$
    Update the end time of $p$
end for
Resample $P$ according to the weights $W_p$
```

At each iteration, we append a new segment, chosen according to the prior A, to each possible route (represented by a particle). Then, the traversal time of the new segment is chosen so that it will have a minimal DTW distance to the respective time interval of the tracked power profile. We take this minimal distance as the weight of the new route. After normalizing the weights of all routes, a resampling phase takes place. N routes are chosen from the existing set of routes according to the particle weights distribution[4]. The new resampled set of routes is the input to the next iteration of the particle filter. The total number of iterations should not exceed an upper bound on the number of segments that the tracked device can traverse. Note however that a route may exhaust the examined power profile before the last iteration (namely, the end time of that route reached t_{max}). In such a case we do not update the route in all subsequent iterations (this case is not described in Algorithm 2 to facilitate fluency of exposition).

Before calculating the DTW distance of a pair of power profiles the profiles are preprocessed to remove as much noise as possible. We first normalize the power profile by subtracting its mean and dividing by the standard deviation of all values included in that profile. Then, we zero out all power values below a threshold percentile. This last step allows us to focus only on the peaks in power consumption where the radio's power consumption is dominant while ignoring the lower power values for which the radio's power has a lesser effect. The percentile threshold we use in this paper is 90%.

Upon its completion, the particle filter outputs a set of N routes of various lengths. To select the best estimate route the simple approach is to choose the route that appears the most number of times in the output set as it has the highest probability to occur. Nonetheless, since a route is composed of multiple segments chosen at separate steps, at each step the weight of a route is determined solely based on the last segment added to the route. Therefore, the output route set is biased in favor of routes ending with segments that were given higher weights, while the weights of the initial segments have a diminishing effect on the route distribution with every new iteration. To counter this bias, we choose another estimate route using a procedure we call *iterative majority vote*, described is Appendix B.

7 Experiments

7.1 Data collection

Our experiments required collecting real power consumption data from smartphone devices along different routes. We developed the PowerSpy android application[5] that collects various measurements including signal strength, voltage, current, GPS coordinates, temperature, state of discharge (battery level) and cell identifier. The recordings were performed using Nexus 4, Nexus 5 and HTC mobile devices.

7.2 Assumptions and limitations

Exploring the limits of our attack, i.e. establishing the minimal necessary conditions for it to work, is beyond our resources. For this reason, we state the assumptions on which we rely in our methods.

We assume there is enough variability in power consumption along a route to exhibit unique features. Lack of variability may be due to high density of cellular antennas that flatten the signal strength profile. We also assume that enough communication is occurring for the signal strength to have an effect on power consumption. This is a reasonable assumption, since background synchronization of data happens frequently in smartphone devices. Moreover, the driver might be using navigation software or streaming music. However, at this stage, it is difficult to determine how inconsistent phone usage across different rides will affect our attacks.

Identifying which route the user took involves understanding which power measurements collected from her mobile device occurred during driving activity. Here we simply assume that we can identify driving activity. Other works (e.g., [22]) address this question by using data from other sensors that require no permissions, such as gyroscopes and accelerometers.

Some events that occur while driving, such as an incoming phone call, can have a significant effect on power

[4] Note that the resampling of the new routes can have repetitions. Namely, the same route can be chosen more than one time

[5] Source code can be obtained from https://bitbucket.org/ymcrcat/powerspy.

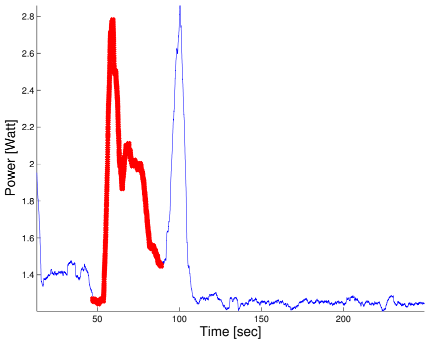


Figure 4: Power profile with a phone call occurring between 50-90 seconds. Profile region during phone call is marked in red.

consumption. Figure 4 shows the power profile of a device at rest when a phone call takes place (the part marked in red). The peak immediately after the phone call is caused by using the phone to terminate the phone call and turn off the display. We can see that this event appears prominently in the power profile and can cope with such transient effects by identifying and truncating peaks that stand out in the profile. In addition, smoothing the profile by a moving average should mitigate these transient effects.

7.3 Route distinguishability

To evaluate the algorithm for distinguishing routes (section 4) we recorded reference profiles for multiple different routes. The profiles include measurements from both Nexus 4 and Nexus 5 models. In total we had a dataset of 294 profiles, representing 36 unique routes. Driving in different directions along the same roads (from point A to B vs. from point B to A) is considered two different routes. We perform cross validation using multiple iterations (100 iterations), each time using a random portion of the profiles as a training set, and requiring equal number of samples for each possible class. The sizes of the training and test sets depend on how many reference routes per profile we require each time. Naturally, the more reference profiles we have, the higher the identification rate.

One evaluation round included 29 unique routes, with only 1 reference profile per route in the training set, and 211 test routes. It resulted in correct identification rate of 40%. That is compared to the random guess probability of only 3%. Another round included 25 unique routes, with 2 reference profiles per route in the training set and 182 routes in the test set, and resulted in correct identification rate of 53% (compared to the random guess probability of only 4%). Having 5 reference profiles per route (for 17 unique routes) raises the identification rate to 71%, compared to the random guess probability of 5.8%. And finally, for 8 reference profiles per route we get 85% correct identification. The results are summarized in table 1.

We can see that an attacker can have a significant advantage in guessing the route taken by a user.

7.4 Real-time mobile device tracking

We evaluate the algorithm for real-time mobile device tracking (section 5) using a set of 10 training profiles and an additional test profile. The evaluation simulates the conditions of real-time tracking by serially feeding samples to the algorithm as if they are received from an application installed on the device. We calculate the estimation error, i.e. the distance between the estimated coordinates and the true location of the mobile device at each step of the simulation. We are interested in the *convergence time*, i.e. the number of samples it takes until the location estimate is close enough to the true location, as well as in the distribution of the estimation errors given by a histogram of the absolute values of the distances.

Figure 5 illustrates the performance of our tracking algorithm for one of the routes, which was about 19 kilometers long. At the beginning, when there are very few power samples, the location estimation is extremely inaccurate, but after two minutes we lock on the true location. We obtained a precise estimate from 2 minutes up until 20 minutes on the route, where our estimate slightly diverges, due to increased velocity on a freeway segment. Around 26 minutes (in figure 5a) we have a large estimation error, but as we mentioned earlier, these kind of errors are easy to prevent by imposing a simple motion model (section 5.2). Most of the errors are small compared to the length of the route: 80% of the estimation errors are less than 1 km.

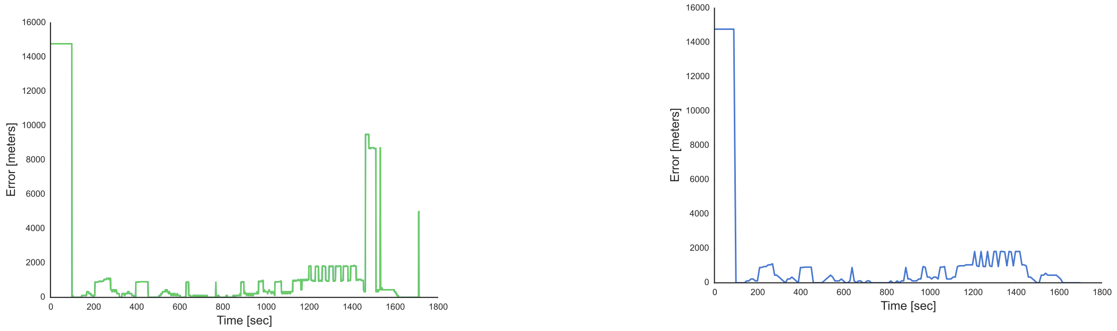
We also tested the improved tracking algorithm explained in section 5.2. Figure 5b presents the estimation error over time, and we can see that the big errors towards the end of the route that appeared in 5a are not present in fig. 5b. Moreover, now almost 90% of the estimation errors are below 1 km (fig. 6).

We provide animations visualizing our results for real-time tracking at the following links. The animations, generated using our estimations of the target's location, depict a moving target along the route and our estimation of its location. The first one corresponds to the method described in 5.1, and the second to the one described in 5.2 that uses the motion model based correction:

crypto.stanford.edu/powerspy/tracking1.mov
crypto.stanford.edu/powerspy/tracking2.mov

# Unique Routes	# Ref. Profiles/Route	# Test Routes	Correct Identification %	Random Guess %
8	10	55	85	13
17	5	119	71	6
17	4	136	68	6
21	3	157	61	5
25	2	182	53	4
29	1	211	40	3

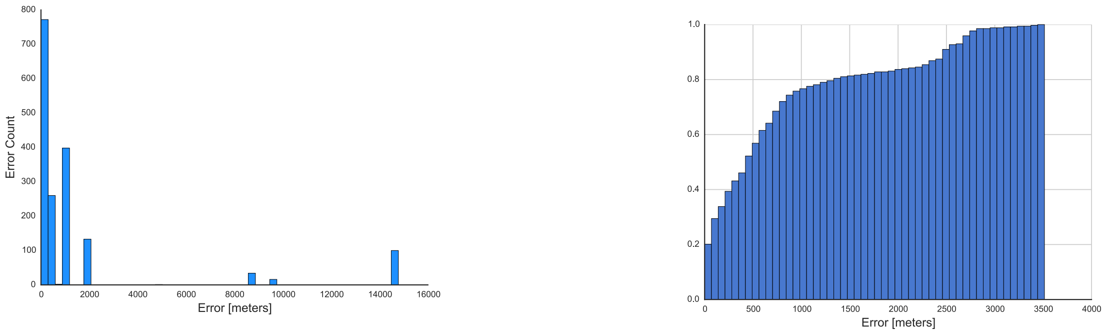
Table 1: Route distinguishability evaluation results. First column indicates the number of unique routes in the training set. Second column indicates the number of training samples per route at the attacker's disposal. Number of test routes indicates the number of power profiles the attacker is trying to classify. Correct identification percentage indicates the percentage of correctly identified routes as a fraction of the third column (test set size), which could be then compared to the expected success of random guessing in the last column.



(a) Convergence to true location.

(b) Location estimation error for improved tracking algorithm.

Figure 5: Location estimation error for online tracking.



(a) Errors histogram. Almost 90% of the errors are less than 1 km.

(b) Error cumulative distribution.

Figure 6: Estimation errors distribution for motion-model tracking.

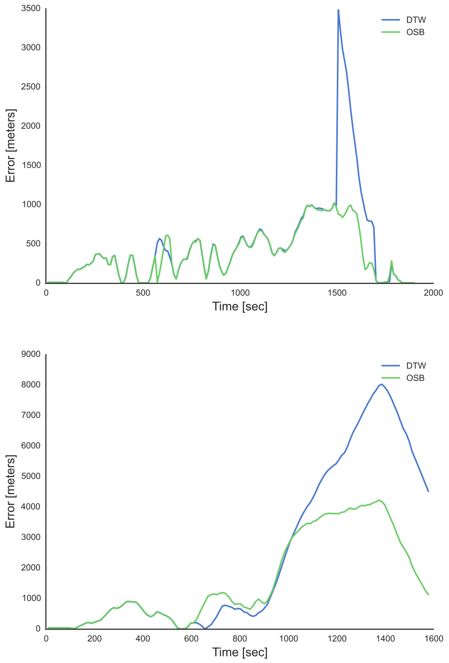


Figure 7: Comparison of DTW and OSB for real-time tracking.

7.4.1 OSB vs. DTW

We compare the performance of Dynamic Time Warping to that of Optimal Subsequence Bijection (section 5.3). Figure 7 present such a comparison for the same route, using two different recordings. The tracking was performed without compensating for errors using a motion model, to evaluate the performance of the subsequence matching algorithms as they are. We can see that, in both cases, Optimal Subsequence Bijection outperforms the standard Subsequence-DTW most of the time. Therefore, we suggest that further experimentation with OSB could potentially be beneficial for this task.

7.5 Inference of new routes

7.5.1 Setup

For the evaluation of the particle filter presented in Section 6 we considered an area depicted in Figure 8. The area has 13 intersections having 35 road segments[6]. The average length of a road segment is about 400 meters. The average travel time over the segments is around 70 seconds. The area is located in the center of Haifa, a city located in northern Israel, having a population density comparable to Philadelphia or Miami. Traffic congestion in this area varies across segments and time of day. For each power recording, the track traversed at least one congested segment. Most of the 13 intersections have traffic lights, and about a quarter of the road segments pass through them.

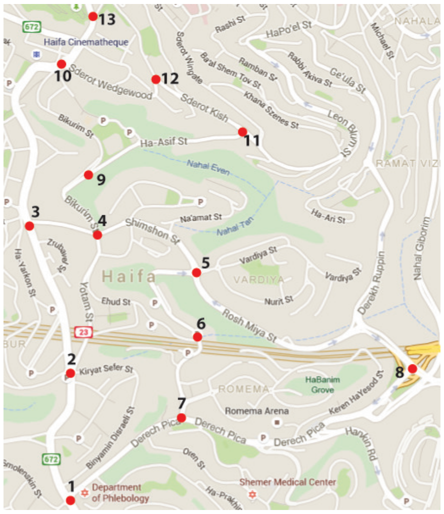


Figure 8: Map of area and intersections for route inference.

We had three pre-recording sessions which in total covered all segments. Each road segment was entered from every possible direction to account for the hysteresis effects. The pre-recording sessions were done using the same Nexus 4 phone.

We set the following parameters of the HMM (as they are defined in Appendix A):

1. A – This set defines the transition probabilities between the road segments. We set these probabilities to be uniformly distributed over all possible transitions. Namely, $a_{xyz} = \{1/|I_y| \; |I_y = \{w|(y,w) \in R, w \neq x\}\}$.

2. B – This set defines the distribution of power profile observations over each state. These probabilities depend on the road segments and their location relative to the nearby based stations. We do not need an explicit formulation of these probabilities to employ the particle filter. The likelihood of a a power profile to be associated with a road segment is estimated by the DTW distance of the power profile to prerecorded power profiles of that segment.

3. Π – This set defines the initial state distribution. We assume that the starting intersection of the tracked device is known. This applies to scenarios where the tracking begins from well-known locations, such as the user's home, office, or another location the attacker knows in advance.

For testing, we used 4 phones: two Nexus 4 (different from the one used for the pre-recordings), a Nexus 5

[6] Three of the segments are one way streets.

Phone	Track
Nexus 4 #1	8-5-6-7-1-2-3-4-5-6-4-3-2-1-7-8
Nexus 4 #2	7-1-2-3-4-5-8-7-6-5-4-2-1-7-8
Nexus 5	3-2-4-9-10-12-11-9-4-5-6-4-3-2-1-7-6-5-8-7
HTC Desire	10-12-11-9-4-2-1-7-6-5-8

Table 2: Test Routes

	random	frequent	Alg. 3	combined
Nexus 4 #1	33%	65%	48%	80%
Nexus 4 #2	31%	48%	56%	72%
Nexus 5	20%	33%	32%	55%
HTC Desire	22%	40%	41%	65%

Table 3: Destination localization

and an HTC Desire. Each phone was used to record the power profile of a different route. The four routes combined cover almost all of the road segments in the area. Table 2 details the routes by their corresponding sequences of intersection identifiers. These route recordings were done on different days, different time of day and varying weather conditions.

As noted, we can only measure the aggregate power consumption which can be significantly affected by applications that run continuously. To have a better sense of the effects of these applications the phones were run with different number of background applications. Nexus 4 #1, Nexus 5 and HTC Desire have a relatively modest number of applications which included (beyond the default Android apps): Email (corporate account), Gmail, and Google Calender. Nexus 4 #2 has a much higher number of application which included on top of the applications of phone #1: Facebook, Twitter, Skype, Waze, and WhatsApp. All those applications periodically send and receive traffic.

For each of the four tracks we derived all possible subtracks having 3 to 7 road segments. We estimated each such sub-track. In total we estimated around 200 subtracks. For each sub-track we employed Algorithms 2 and 3 to get two best estimates for the sub-track.

Tables 3 to 5 summarize the results of route estimation for each of the four phones. For each route we have two alternatives for picking an estimate (1) the most frequent route in the particle set as output by Algorithm 2; (2) the route output by Algorithm 3. For each alternative we note the road segment in which the phone is estimated to be after the completion of its track and compare it with the final road segment of the true route. This allows us to measure the accuracy of the algorithm for estimating the location of the user's destination (the end of the track). This is the most important metric for many attack scenarios where the attacker wishes to learn the destination of the victim.

In some cases it may also be beneficial for the attacker to know the actual route through which the victim traversed on his way to the destination. For this purpose, we also calculate for each alternative estimate the Levenshtein distance between it and the true route. The Levenshtein distance is a standard metric for measuring the difference between two sequences [18]. It equals the minimum number of updates required in order to change one sequence to the next. In this context, we treat a route as a sequence of intersections. The distance is normalized by the length of the longer route of the two. This allows us to measure the accuracy of the algorithm for estimating the full track the user traversed. For each estimate we also note whether it is an exact fit with the true route (i.e., zero distance). The percentage of successful localization of destination, average Levenshtein distance and percentage of exact full route fits are calculated for each type of estimated route. We also calculate these metrics for both estimates combined while taking into account for each track the best of the two estimates. To benchmark the results we note in each table the performance of a random estimation algorithm which simply outputs a random, albeit feasible, route.

The results in Table 3 show the accuracy of destination identification. It is evident that the performance of the most frequent route output by the particle filter is comparable to the performance of the best estimate output by Algorithm 3. However, their combined performance is significantly better than either estimates alone and predict more accurately the final destination of the phone. This result suggests that Algorithm 3 extracts significant amount of information from the routes output by the particle filter beyond the information gleaned from the most frequent route.

Table 3 indicates that for Nexus 4 #1 the combined route estimates were able to identify the final road segment for 80% of all scenarios. For Nexus 4 #2 which was running many applications the final destination estimates are somewhat less accurate (72%). This is attributed to the more noisy measurements of the aggregate power consumption. The accuracy for the two models – Nexus 5 and HTC Desire – is lower than the accuracy achieved for Nexus 4. Remember that all our pre-recordings were done using a Nexus 4. These results may indicate that the power consumption profile of the cellular radio is dependent on the phone's model. Nonetheless, for both phones we achieve significantly higher accuracy of destination localization (55% and 65%) as compared to the random case (about 20%).

Tables 4 and 5 present measures – Levenshtein distance and exact full route fit – of the accuracy of estimates for the full route the phone took to its destination. Here, again, the algorithm presented for Nexus 4 #1 superior performance. It was able to exactly estimate 45%

	random	frequent	Alg. 3	combined
Nexus 4 #1	0.61	0.38	0.27	0.24
Nexus 4 #2	0.63	0.61	0.59	0.52
Nexus 5	0.68	0.6	0.55	0.45
HTC Desire	0.65	0.59	0.5	0.45

Table 4: Levenshtein distance

	random	frequent	Alg. 3	combined
Nexus 4 #1	4%	38%	22%	45%
Nexus 4 #2	5%	8.5%	5%	15%
Nexus 5	3%	15%	9%	20%
HTC Desire	5%	10%	12%	17%

Table 5: Exact full route fit

of the full route to the destination. On the other hand, for the more busy Nexus 4 #2 and the other model phones the performance was worse. It is evident from the results that for these three phones the algorithm had difficulties producing an accurate estimate of the full route. Nonetheless, in all cases the accuracy is always markedly higher than that of the random case.

To have a better sense of the distance metric used to evaluate the quality of the estimated routes Figure 9 depicts three cases of estimation errors and their corresponding distance values in increasing order. It can be seen that even estimation error having relatively high distances can have a significant amount of information regarding the true route.

8 Future directions

In this section we discuss ideas for further research, improvements, and additions to our method.

8.1 Power consumption inference

While new (yet very common) smartphone models contain an internal ampere-meter and provide access to current data, other models (for instance Galaxy S III) supply voltage but not current measurements. Therefore on these models we cannot directly calculate the power consumption. V-edge [31] proposes using voltage dynamics to model a mobile device's power consumption. That and any other similar technique would extend our method and make it applicable to additional smartphone models.

Ref. [33] presents PowerTutor, an application that estimates power consumption by different components of the smartphone device based on voltage and state of discharge measurements. Isolating the power consumed by the cellular connectivity will improve our method by eliminating the noise introduced by other components such as audio/Bluetooth/WiFi etc. that do not directly depend on the route.

8.2 State of Discharge (SOD)

The time derivative of the State-of-Discharge (the battery level) is basically a very coarse indicator of power consumption. While it seemed to be too inaccurate for our purpose, there is a chance that extracting better features from it or having few possible routes may render distinguishing routes based on SOD profiles feasible. Putting it to the test is even more interesting given the HTML 5 Battery API that enables obtaining certain battery statistics from a web-page via JavaScript. Our findings demonstrate how future increases in the sampling resolution of the battery stats may turn this API even more dangerous, allowing web-based attacks.

8.3 Choice of reference routes

Successful classification depends among other factors on good matching between the power profile we want to classify and the reference power profiles. Optimal matching might be a matter of month, time of day, traffic on the road, and more. We can possibly improve our classification if we tag the reference profiles with those associated conditions and select reference profiles matching the current conditions when trying to distinguish a route. That of course requires collecting many more reference profiles.

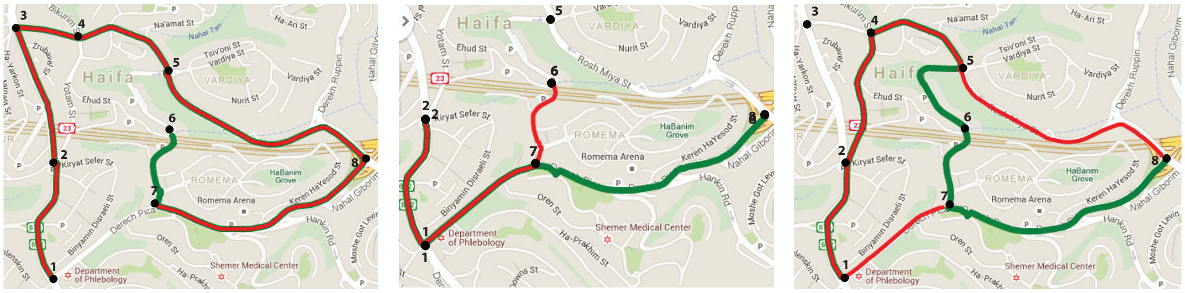
8.4 Collecting a massive dataset

Collecting a massive dataset of power profiles associated with GPS coordinates is a feasible task given vendors' capability to legally collect analytics about users' use of their smartphones. Obtaining such big dataset will enable us to better understand how well our approach can scale and whether it can be used with much less prior knowledge about the users.

9 Defenses

9.1 Non-defenses

One might think that by adding noise or limiting the sampling rate or the resolution of the voltage and current measurements one could protect location privacy. However, our method does not rely on high sampling frequency or resolution. In fact, our method works well with profiles much coarser than what we can directly get from the raw power data, and for the route distinguishing task we actually performed smoothing and downsampling of the data yet obtained good results. Our method also works well with signal strength, which is provided



(a) Distance = 0.125

(b) Distance = 0.25

(c) Distance = 0.43

Figure 9: Examples of estimation errors and their corresponding distances (partial map is depicted). The true route is green and the estimated route is red.

with much lower resolution and sampling frequency[7].

9.2 Risky combination of power data and network access

One way of reporting voltage and current measurements to the attacker is via a network connection to the attacker's server. Warning the user of this risky combination may somewhat raise the bar for this attack. There are of course other ways to leak this information. For instance, a malicious application disguised as a diagnostic software can access power data and log it to a file, without attempting to make a network connection, while another, seemingly unrelated, application reads the data from that file and sends it over the network.

9.3 Secure hardware design

The problem with access to total power consumption is that it leaks the power consumed by the transceiver circuitry and communication related tasks that indicate signal strength. While power measurements can be useful for profiling applications, in many cases, examining the power consumed by the processors executing the software logic might be enough. We therefore suggest that supplying only measurements of the power consumed by the processors (excluding the power consumed by the TX/RX chain) could be a reasonable trade-off between functionality and privacy.

9.4 Requiring superuser privileges

A simple yet effective prevention may be requiring superuser privileges (or being root) to access power supply data on the phone. Thus, developers and power-users can install diagnostic software or run a version of their application that collects power data on a rooted phone, whereas the release version of the software excludes this functionality. This would of course prevent the collection of anonymous performance statistics from the install-base, but as we have shown, such data can indicate much more than performance.

9.5 Power consumption as a coarse location indicator

Same as the cell identifier is defined as a coarse location indicator, and requires appropriate permissions to be accessed, power consumption data can also be defined as one. The user will then be aware, when installing applications that access voltage and current data, of the application's potential capabilities, and the risk potentially posed to her privacy. This defense may actually be the most consistent with the current security policies of smartphone operating systems like Android and iOS, and their current permission schemes.

10 Related work

Power analysis is known to be a powerful side-channel. The most well-known example is the use of high sample rate ($\sim$20 MHz) power traces from externally connected power monitors to recover private encryption keys from a cryptographic system [15]. Prior work has also established the relationship between signal strength and power consumption in smartphones [6, 29]. Further, Bartendr [29] demonstrated that paths of signal strength measurements are stable across several drives.

PowerSpy combines these insights on power analysis and improving smartphone energy efficiency to reveal a new privacy attack. Specifically, we demonstrate that an attacker can determine a user's location simply by monitoring the cellular modem's changes in power consumption with the smartphone's alarmingly unprotected $\sim$100 Hz internal power monitor.

[7] In fact, since it reflects more directly the environmental conditions, signal strength data can provide even better route identification and tracking. We did not focus on signal strength since accessing it requires access permissions and has already drawn research attention to it as useful for localization.

10.1 Many sensors can leak location

Prior work has demonstrated that data from cellular modems can be used to localize a mobile device (an extensive overview appears in Gentile et al. [10]). Similar to PowerSpy, these works fingerprint the area of interest with pre-recorded radio maps. Others use signal strength to calculate distances to base stations at known locations. All of these methods [16, 24, 25, 30] require signal strength measurements and base station ID or WiFi network name (SSID), which is now protected on Android and iOS. Our work does not rely on the signal strength, cell ID, or SSID. PowerSpy only requires access to power measurements, which are currently unprotected on Android.

PowerSpy builds on a large body of work that has shown how a variety of unprotected sensors can leak location information. Zhou et al. [34] reveal that audio on/off status is a side-channel for location tracking without permissions. In particular, they extract a sequence of intervals where audio is on and off while driving instructions are being played by Google's navigation application. By comparing these intervals with reference sequences, the authors were able to identify routes taken by the user. *SurroundSense* [3] demonstrates that ambient sound and light can be used for mobile phone localization. They focus on legitimate use-cases, but the same methods could be leveraged for breaching privacy. *AC-Complice* [12] demonstrates how continuous measurements from unprotected accelerometers in smartphones can reveal a user's location. Hua et al. [13] extend AC-Complice by showing that accelerometers can also reveal where a user is located in a metropolitan train system.

10.2 Other private information leaked from smartphone sensors

An emerging line of work shows that various phone sensors can leak private information other than location. In future work we will continue analyzing power measurements to determine if other private information is leaked.

Prior work has demonstrated how smartphone sensors can be used to fingerprint specific devices. *AccelPrint* [9] shows that smartphones can be fingerprinted by tracking imperfections in their accelerometer measurements. Fingerprinting of mobile devices by the characteristics of their loudspeakers is proposed in [7, 8]. Further, Bojinov et. al. [4] showed that various sensors in smartphones can be used to identify a mobile device by its unique hardware characteristics. Lukas et. al. [20] proposed a method for digital camera fingerprinting by noise patterns present in the images. [19] enhances the method enabling identification of not only the model but also particular cameras.

Sensors can also reveal a user's input such as speech and touch gestures. The *Gyrophone* study [21] showed that gyroscopes on smartphones can be used for eavesdropping on a conversation in the vicinity of the phone and identifying the speakers. Several works [2, 5, 32] have shown that the accelerometer and gyroscope can leak information about touch and swipe inputs to a foreground application.

11 Conclusion

PowerSpy shows that applications with access to a smartphone's power monitor can gain information about the location of a mobile device – without accessing the GPS or any other coarse location indicators. Our approach enables known route identification, real-time tracking, and identification of a new route by only analyzing the phone's power consumption. We evaluated PowerSpy on real-world data collected from popular smartphones that have a significant mobile market share, and demonstrated their effectiveness. We believe that with more data, our approach can be made more accurate and reveal more information about the phone's location.

Our work is an example of the unintended consequences that result from giving 3rd party applications access to sensors. It suggests that even seemingly benign sensors need to be protected by permissions, or at the very least, that more security modeling needs to be done before giving 3rd party applications access to sensors.

Acknowledgments

We would like to thank Gil Shotan and Yoav Shechtman for helping to collect the data used for evaluation, Prof. Mykel J. Kochenderfer from Stanford University for providing advice regarding location tracking techniques, Roy Frostig for providing advice regarding classification and inference on graphs, and finally Katharina Roesler for proofreading the paper. This work was supported by NSF and the DARPA SAFER program. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF or DARPA.

References

[1] ARULAMPALAM, M. S., MASKELL, S., GORDON, N., AND CLAPP, T. A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking. *Signal Processing, IEEE Transactions on 50*, 2 (2002), 174–188.

[2] AVIV, A. J., SAPP, B., BLAZE, M., AND SMITH, J. M. Practicality of accelerometer side channels on smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference* (2012), ACM, pp. 41–50.

[3] Azizyan, M., Constandache, I., and Roy Choudhury, R. Surroundsense: mobile phone localization via ambience fingerprinting. In *Proceedings of the 15th annual international conference on Mobile computing and networking* (2009), ACM, pp. 261–272.

[4] Bojinov, H., Michalevsky, Y., Nakibly, G., and Boneh, D. Mobile device identification via sensor fingerprinting. *arXiv preprint arXiv:1408.1416* (2014).

[5] Cai, L., and Chen, H. Touchlogger: Inferring keystrokes on touch screen from smartphone motion. In *Usenix HotSec* (2011).

[6] Carroll, A., and Heiser, G. An analysis of power consumption in a smartphone. In *USENIX Annual Technical Conference* (2010).

[7] Clarkson, W. B., and Felten, E. W. Breaking assumptions: distinguishing between seemingly identical items using cheap sensors. Tech. rep., Princeton University, 2012.

[8] Das, A., and Borisov, N. Poster: Fingerprinting smartphones through speaker. In *Poster at the IEEE Security and Privacy Symposium* (2014).

[9] Dey, S., Roy, N., Xu, W., Choudhury, R. R., and Nelakuditi, S. Accelprint: Imperfections of accelerometers make smartphones trackable. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2014).

[10] Gentile, C., Alsindi, N., Raulefs, R., and Teolis, C. *Geolocation Techniques*. Springer New York, New York, NY, 2013.

[11] Goldsmith, A. *Wireless communications*. Cambridge university press, 2005.

[12] Han, J., Owusu, E., Nguyen, L. T., Perrig, A., and Zhang, J. ACComplice: Location inference using accelerometers on smartphones. In *Proceedings of the 2012 International Conference on COMmunication Systems & NETworkS* (2012).

[13] Hua, J., Shen, Z., and Zhong, S. We can track you if you take the metro: Tracking metro riders using accelerometers on smartphones. *arXiv:1505.05958* (2015).

[14] Huang, J., Qian, F., Gerber, A., Mao, Z. M., Sen, S., and Spatscheck, O. A close examination of performance and power characteristics of 4G LTE networks. In *MobiSys* (2012).

[15] Kocher, P., Jaffe, J., and Jun, B. Differential power analysis. In *Advances in Cryptology – CRYPTO'99* (1999), Springer, pp. 388–397.

[16] Krumm, J., and Horvitz, E. Locadio: Inferring motion and location from wi-fi signal strengths. In *MobiQuitous* (2004), pp. 4–13.

[17] Latecki, L., Wang, Q., Koknar-Tezel, S., and Megalooikonomou, V. Optimal subsequence bijection. In *Data Mining, 2007. ICDM 2007. Seventh IEEE International Conference on* (Oct 2007), pp. 565–570.

[18] Levenshtein, V. I. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet physics doklady* (1966), vol. 10, p. 707.

[19] Li, C.-T. Source camera identification using enhanced sensor pattern noise. *Information Forensics and Security, IEEE Transactions on 5*, 2 (2010), 280–287.

[20] Lukas, J., Fridrich, J., and Goljan, M. Digital camera identification from sensor pattern noise. *Information Forensics and Security, IEEE Transactions on 1*, 2 (2006), 205–214.

[21] Michalevsky, Y., Boneh, D., and Nakibly, G. Gyrophone: Recognizing speech from gyroscope signals. In *Proc. 23rd USENIX Security Symposium (SEC14), USENIX Association* (2014).

[22] Mohan, P., Padmanabhan, V. N. V., and Ramjee, R. Nericell: rich monitoring of road and traffic conditions using mobile smartphones. In *... of the 6th ACM conference on ...* (New York, New York, USA, Nov. 2008), ACM Press, p. 323.

[23] Müller, M. *Information Retrieval for Music and Motion*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[24] Muthukrishnan, K., van der Zwaag, B. J., and Havinga, P. Inferring motion and location using WLAN RSSI. In *Mobile Entity Localization and Tracking in GPS-less Environnments*. Springer, 2009, pp. 163–182.

[25] Ouyang, R. W., Wong, A.-S., Lea, C.-T., and Zhang, V. Y. Received signal strength-based wireless localization via semidefinite programming. In *Global Telecommunications Conference, 2009. GLOBECOM 2009. IEEE* (2009), IEEE, pp. 1–6.

[26] Pollini, G. P. Trends in handover design. *Communications Magazine, IEEE 34*, 3 (1996), 82–90.

[27] Rabiner, L. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE* (1989).

[28] Ristic, B., Arulampalam, S., and Gordon, N. Beyond the kalman filter. *IEEE AEROSPACE AND ELECTRONIC SYSTEMS MAGAZINE 19*, 7 (2004), 37–38.

[29] Schulman, A., Spring, N., Navda, V., Ramjee, R., Deshpande, P., Grunewald, C., Padmanabhan, V. N., and Jain, K. Bartendr: a practical approach to energy-aware cellular data scheduling. *MOBICOM* (2010).

[30] Sohn, T., Varshavsky, A., LaMarca, A., Chen, M. Y., Choudhury, T., Smith, I., Consolvo, S., Hightower, J., Griswold, W. G., and De Lara, E. Mobility detection using everyday gsm traces. In *UbiComp 2006: Ubiquitous Computing*. Springer, 2006, pp. 212–224.

[31] Xu, F., Liu, Y., Li, Q., and Zhang, Y. V-edge: fast self-constructive power modeling of smartphones based on battery voltage dynamics. *Presented as part of the 10th USENIX ...* (2013).

[32] Xu, Z., Bai, K., and Zhu, S. Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks* (2012), ACM, pp. 113–124.

[33] Zhang, L., Tiwana, B., Qian, Z., and Wang, Z. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. *Proceedings of the ...* (2010).

[34] Zhou, X., Demetriou, S., He, D., Naveed, M., Pan, X., Wang, X., Gunter, C. A., and Nahrstedt, K. Identity, location, disease and more: inferring your secrets from android public resources. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13* (2013), pp. 1017–1028.

A Formal model of new route inference

In this section we formalize the problem of the new route inference (Section 6) as a hidden Markov model (HMM) [27]. Let I denote the set of intersections in an area in which we wish to track a mobile device. A road segment is given by an ordered pair of intersections (x,y), defined to be a continuous road between intersection x and intersection y. We denote the set of road segments as R.

We assume that once a device starts to traverse a road segment it does not change the direction of its movement until it reaches the end of the segment. We define a state for each road segment. We say that the tracked device is in state s_{xy} if the device is currently traversing a road segment (x, y), where $x, y \in I$. We denote the route of the tracked device as a (Q, T), where

$$Q = \left\{q_1 = s_{x_1 x_2}, q_2 = s_{x_2 x_3}, ...\right\} \quad T = \{t_1, t_2, ...\}$$

For such a route the device has traversed from x_i to x_{i+1} during time interval $[t_{i-1}, t_i]$ ($t_0 = 0, t_{i-1} < t_i\ \forall i > 0$).

Let $A = \left\{a_{xyz} | \forall x, y, z \in I\right\}$ be the state transition probability distribution, where

$$a_{xyz} = p\left\{q_{i+1} = s_{yz} | q_i = s_{xy}\right\} \tag{1}$$

Note that $a_{xyz} = 0$ if there is no road between intersections x and y or no road between intersections y and z. A traversal of the device over a road segment yields a power consumption profile of length equal to the duration of that movement. We denote a power consumption profile as an observation o. Let B be the probability distribution of yielding a given power profile while the device traversed a given segment. Due to the hysteresis of hand-offs between cellular base stations, this probability depends on the previous segment the device traversed. Finally, let $\Pi = \left\{\pi_{xy}\right\}$ be the initial state distribution, where π_{xy} is the probability that the device initially traversed segment (x, y). If there is no road segment between intersections x and y, then $\pi_{xy} = 0$. In our model we treat this initial state as the state of the device *before* the start of the observed power profile. We need to take this state into account due to the hysteresis effect. Note that an HMM is characterized by A, B, and Π.

The route inference problem is defined as follows. Given an observation of a power profile O over time interval $[0, t_{\max}]$, and given a model A, B and Π, we need to find a route (Q, T) such that $p\{(Q, T)|O\}$ is maximized. In the following we denote the part of O which begins at time t' and ends at time t'' by $O_{[t', t'']}$. Note that $O = O_{[0, t_{\max}]}$. We consider the time interval $[0, t_{\max}]$ as having a discrete resolution of τ.

B Choosing the best inferred route

Upon its completion, the particle filter described in section 6.1 outputs a set of N routes of various lengths. We denote this set by P_{final}. This set exhibits an estimate of the distribution of routes given the power profile of the tracked device. The simple approach to select the best estimate is to choose the route that appears most frequently in P_{final} as it has the highest probability to occur. Nonetheless, since a route is composed of multiple segments chosen at separate steps, at each step the weight of a route is determined solely based on the last segment added to the route. Therefore, in P_{final} there is a bias in favor of routes ending with segments that were given higher weights, while the weights of the initial segments have a diminishing effect on the route distribution with every new iteration.

To counter this bias, we choose another estimate using a procedure we call *iterative majority vote*. This procedure ranks the routes based on the prevalence of their prefixes. At each iteration i the procedure calculates – Prefix[i] – a list of prefixes of length i ranked by their prevalence out of the all routes that has a prefix in Prefix[i-1]. Prefix[i][n] denotes the prefix of rank n. The operation $p||j$ – where p is a route and j is an intersection – denotes the appending of j to p. At each iteration i algorithm 3 is executed. In the following we denote RoutePrefixed(R, p) to be the subset of routes out of the set R having p as their prefix.

Algorithm 3 Iterative majority vote

```
$I' \leftarrow I$
**while** not all prefixes found **do**
    Prf $\leftarrow$ next prefix from Prefix[i].
    Find $j \in I'$ that maximizes
      RoutePrefixed(RoutePrefixed($P_{\text{final}}$, Prf), Prf$||j$)
    **if** no such $j$ is found **then**
        $I' = I$
        continue loop
    **end if**
    Prefix$[i+1] \leftarrow$ Prefix$[i+1] \cup \{$Prf$||j\}$
    $I' = I' - \{j\}$
**end while**
```

At each iteration i we rank the prefixes based on the ranks of the previous iteration. Namely, prefixes which are extensions of a shorter prefix having a higher rank in a previous iteration will always get higher ranking over prefixes which are extensions of a lower rank prefix. At each iteration the we first find the most common prefixes of length $i+1$, which start with the most common prefix of length i found in the previous iteration, and rank them according to their prevalence. Then we look for common prefixes of length $i+1$, that start with the second most common prefix of length i found in the previous iteration, and so on until all prefixes of length $i+1$ are found. The intuition is as follows. The procedure prefers routes traversing segments that are commonly traversed by other routes. Those received a high score when were chosen. Since we cannot pick the most common segments separately from each step (a continuous route probably will not emerge), we iteratively pick the most common segment out of the routes that are prefixed with the segments that were already chosen.

In the Compression Hornet's Nest: A Security Study of Data Compression in Network Services

Giancarlo Pellegrino
CISPA, Saarland University, Germany
gpellegrino@mmci.uni-saarland.de

Davide Balzarotti
EUROCOM, France
davide.balzarotti@eurecom.fr

Stefan Winter
TU Darmstadt, Germany
sw@cs.tu-darmstadt.de

Neeraj Suri
TU Darmstadt, Germany
suri@cs.tu-darmstadt.de

Abstract

In this paper, we investigate the current use of data compression in network services that are at the core of modern web-based applications. While compression reduces network traffic, if not properly implemented it may make an application vulnerable to DoS attacks. Despite the popularity of similar attacks in the past, such as *zip bombs* or *XML bombs*, current protocol specifications and design patterns indicate that developers are still mostly unaware of the proper way to handle compressed streams in protocols and web applications. In this paper, we show that denial of services due to improper handling of data compression is a persistent and widespread threat. In our experiments, we review three popular communication protocols and test 19 implementations against highly-compressed protocol messages. Based on the results of our analysis, we list 12 common pitfalls that we observed at the implementation, specification, and configuration levels. Additionally, we discuss a number of previously unknown resource exhaustion vulnerabilities that can be exploited to mount DoS attacks against popular network service implementations.

1 Introduction

Modern web-based software applications rely on a number of core network services that provide the basic communication between software components. For instance, the list includes Web servers, email servers, and instant messaging (IM) services, just to name some of the more widespread ones. As a consequence of their popularity, Denial of Service (DoS) may have very severe consequences on the availability of many web services. In fact, according to the 2014 Global Report on the Cost of Cyber Crime [35], the impact of application DoS is dramatic: 50% of the organizations have suffered from such an attack, and the average cost of a single attack is estimated to be over $166K US [35].

For performance reasons, many network services extensively use data compression to reduce the amount of data transferred between the communicating parties. The use of compression can be mandated by protocol specifications or it can be an implementation-dependent feature. While compression indeed reduces network traffic, at the same time, if not properly implemented, it may also make applications vulnerable to DoS attacks. The problem was first brought to users' attention in 1996 in the form of a recursively highly-compressed file archive prepared with the only goal of exhausting the resources of programs that attempt to inspect its content. In the past, these *zip bombs* were used, for example, to mount DoS attacks against bulletin board systems [1] and antivirus software [2, 57].

While this may now seem an old, unsophisticated, and easily avoidable threat, we discovered that developers did not fully learn from prior mistakes. As a result, the risks of supporting data compression are still often overlooked, and descriptions of the proper way to handle compressed messages are either lacking or misleading. In this paper, we investigate the current use of data compression in several popular protocol and network services. Through a number of experiments and by reviewing the source code of several applications, we have identified a number of improper ways to handle data compression at the implementation, specification, and configuration levels. These common mistakes are widespread in many popular applications, including Apache HTTPD and three of the top five most popular XMPP servers. Similar to the zip bombs of 20 years ago, our experiments show that these flaws can easily be exploited to exhaust the server resources and mount a denial of service attack.

The task of handling data compression is not as simple as it may sound. In general, compression amplifies the amount of data that a network service needs to process, and some components may not be designed to handle this volume of data. This may result in the exhaustion of resources for applications that were otherwise considered secure. However, in this paper we show that these mistakes are not only caused by unbounded buffers, and neither are they localized into single components. In fact, as message processing involves different modules, improper communication may result in a lack of synchronization, eventually causing an excessive consumption of resources. Additionally, we show similar mistakes when third-party modules and libraries are used. Here, misleading documentation may create a false sense of security in which the web application developers believe that the data amplification risks are already addressed at the network service level.

To summarize, this paper makes the following contributions:

- We show that resource exhaustion vulnerabilities due to highly-compressed messages are (still) a real threat that can be exploited by remote attackers to mount denial of service attacks;
- We present a list of 12 common pitfalls and susceptibilities that affect the implementation, specification, and configuration levels;
- We tested 11 network services and 10 third-party extensions and web application frameworks for a total of 19 implementations against compression-based DoS attacks;
- We discovered and reported nine previously unknown vulnerabilities, which would allow a remote attacker to mount a denial of service attack.

This paper is organized as follows. In Section 2, we introduce the case studies. Then, in Section 3, we discuss the security risks associated with data compression, revisit popular attacks, and outline the current situation. In Section 4, we detail the current situation and present a list of 12 pitfalls at the implementation, specification, and configuration levels. Then, in Section 5, we describe the experiments and present previously-unknown resource exhaustion vulnerabilities. In Section 6, we review related works, and finally, in Sections 7 and 8, we outline future work and draw some conclusions.

2 Data Compression

Data compression is a coding technique that aims at reducing the number of bits required to represent a string by removing redundant data. Compression is *lossless* when it is possible to reconstruct an exact copy of the original string, or *lossy* otherwise. For a detailed survey on compression algorithms please refer to Salomon et al. [45]. Since the focus of our paper is on the incorrect *use* of compression and it is independent of the algorithm itself, we will discuss our finding and examples using the popular *Deflate* algorithm.

Prot.	Network Service	Native	External
XMPP	ejabberd	✗	-
	Openfire	✗	-
	Prosody	✗	-
	jabberd2	✗	-
	Tigase	✗	-
HTTP	Apache HTTPD	✗	-
	mod-php	-	✗
	CSJRPC	-	✗
	mod-gsoap	-	✗
	mod-dav	✗	-
	Apache Tomcat	-	✗
	Axis2	-	✗
	CXF	✗	✗
	jsonrpc4j	-	✗
	json-rpc	-	✗
	lib-json-rpc	-	✗
	Axis2 standalone	✗	-
	gSOAP standalone	✗	-
IMAP	Dovecot	✗	-
	Cyrus	✗	-

Table 1: Case studies and Implementations

Deflate is a lossless data compression technique that combines together a Huffman encoding with a variant of the `LZ78` algorithm. It is specified in the Request For Comments (RFC) number 1951 [13], released in May 1996, and it is now implemented by the widely used `zlib` library [19], the `gzip` compression tool [18], and the `zip` file archiver tool [22], just to name few popular examples.

Deflate is widely used in many Internet protocols such as the HyperText Transfer Protocol (HTTP) [17], the eXtensible Messaging and Presence Protocol (XMPP) [42], the Internet Message Access Protocol (IMAP) [11], the Transport Layer Security (TLS) protocol [26], the Point-to-Point Protocol (PPP) [60], and the Internet Protocol (IP) [33]. The list includes both text-based and binary protocols. However, since the first category contains fields of arbitrary length where the decompression overhead is more evident, we decided to focus our study on three popular text-based protocols: HTTP, XMPP and IMAP. For each protocol we selected a number of implementations, summarized in Table 1. The columns *Native* and *External* show if the compression is natively supported by the application or if it is provided by an external component.

HTTP - Starting from version 1.1, HTTP supports compression of the HTTP *response* body using different compression algorithms (including Deflate) [17]. While the specification only covers the compression of the response body, we manually verified that several HTTP server im-

HTTP server	No.	Perc.
Apache HTTPD	248	24.8%
NGINX HTTPD	202	20.2%
Google HTTPD	81	8.1%
MS IIS HTTPD	64	6.4%
Apache Tomcat	22	2.2%
Others (20 servers)	102	10.2%
Unknown	218	21.8%
Errors	63	6.3%
Tot. no. of domains	1000	100%

(a) HTTP servers of the first 1000 domains of the Alexa DB of 2013-10-05.

XMPP server	No.	Perc.
ejabberd	56	52.8%
Openfire	11	10.4%
Prosody	9	8.5%
jabberd2	3	2.8%
Tigase	2	1.9%
Other (1 server)	1	0.9%
Unknown	1	0.9%
Errors	23	21.7%
Tot. no. of domains	106	100%

(b) XMPP servers of the 106 domains from xmpp.net of 2013-09-03.

IMAP server	No.	Perc.
Dovecot	31	42.5%
Courier	19	26.0%
Zimbra	6	8.2%
Cyrus	3	4.1%
MS Exchange	2	2.7%
Others (5 servers)	6	8.2%
Unknown	6	8.2%
Servers discovered	73	100.00%

(c) IMAP servers of the first 1000 domains of the Alexa DB

Table 2: Service detection for HTTP, XMPP, and IMAP servers

plementations additionally support the compression of the *request* body. Table 2a shows the result of the HTTP *service detection*[1] in order to identify the most popular HTTP server implementations among the top 1000 domains of the Alexa Top Sites database[2]. From Table 2a, we selected Apache HTTPD 2.2.22 [53] and Apache Tomcat 7 [52] as they are available for GNU/Linux. The former supports message decompression via the module `mod-deflate`, while the latter can be extended with third-party filters. In this paper, we used the 2Way HTTP Compression Servlet Filter 1.2 [37] (2Way for short) and Webutilities 0.0.6 [32].

In our experiments, we considered three use cases that may benefit from request compression: distributed computing, web applications, and sharing static resources. For Apache HTTPD, we selected gSOAP 2.8.7 [59] to develop SOAP-based RPC servers, CSJRPC 0.1 [9] to develop PHP-based JSON RPC servers, the PHP Apache module [55] (mod-php, for short) to develop PHP-based web applications, and WebDAV [21] (as implemented by the built-in Apache module mod-dav) to share static files. For Tomcat, we selected Apache CXF 2.2.7 [51], Apache Axis 2 [50], jsonrpc4j 1.0 [15], json-rpc 1.1 [41], and lib-json-rpc 1.1.2 [7].

We test web servers with the following HTTP request:

```
POST $resource HTTP/1.1\r\n
Host: $domain\r\n
Content-Encoding: gzip\r\n
\r\n
$payload
\r\n
```

where `$resource` is the path to the resource, `$domain` the web server domain, and `$payload` is the compressed payload, or the *compression bomb*. The type of payload varies according to the implementation under test, i.e., JSON or SOAP message requests, and HTML form parameters. For example, the SOAP compression bomb is the following:

```
<soapenv:Envelope [...]>
   $spaces
   <soapenv:Body>[...]</soapenv:
      Body>
</soapenv:Envelope>
```

where `$spaces` can be, for example, a 4GB-long string of blank spaces. Once compressed, this payload is reduced to about 4MB with a compression ratio of about 1:1000, a value close to the maximum compression factor that can be achieved with Zlib, i.e., 1:1024 [19]. It might be possible to generate payloads with higher ratios, for instance, by modifying the compressor to return shorter, but still legal, strings. However, in this paper, we did not investigate this direction and leave this as future work.

XMPP - XMPP is an XML-based protocol offering messaging and presence, and request-response services [43, 44]. XMPP is at the core of several public and IM services, such as Google Talk, in which users exchange text-based messages in real-time. We performed service detection on the list of XMPP services available at `xmpp.net`[3]. Table 2b shows the result of the service detection. We selected the five most popular XMPP servers for our tests: ejabberd 2.1.10 [38], Openfire 3.9.1 [27], Prosody 0.9.3 [56], jabberd2 2.2.8 [54], and Tigase 5.2.0 [58].

To test XMPP servers, we used a similar trick as used in SOAP compression bombs. The highly-compressed XMPP message (i.e., *xmppbomb*) is the following:

```
<?xml version='1.0' ?>
  <stream:stream
```

[1] *Service detection* is a technique to identify the name of a network service by analyzing the server response against a database of fingerprints. In this paper, we used the Nmap Security Scanner tool [30]).

[2] See `http://www.alexa.com/`

[3] `xmpp.net` maintains a publicly accessible list of XMPP services: `http://xmpp.net/services.xml`. We retrieved it on 2013-09-03 and it contained 106 domains.

```
$spaces
to='server' xmlns='jabber:
    client'
[...]>
```

where $spaces can be a 4GB-long string of blank spaces. Also in this case, the compression ratio is about 1:1000.

IMAP - IMAP is a command-response protocol that allows a client to manage emails and mailboxes on a remote server [11]. The protocol supports compression algorithms to reduce the size of both commands and responses [23]. We obtained a list of popular IMAP servers from the first 1000 domains of the Alexa database. We first resolved the Mail eXchange (MX) domain and then we performed service detection. Table 2c shows the results of the service detection. As opposed to HTTP, we report percentages of the total number of the discovered servers because the majority of MX domains do not offer IMAP access to the email boxes. Two of the five IMAP servers were available for GNU/Linux and supported data compression: Dovecot 2.0.19 [16] and Cyrus 2.4.12 [8].

To test for IMAP, we crafted an email which can be compressed with a ratio of about 1:1000. The structure of the email is the following:

```
From: sender@foo\r\n
To:   receiver@foo\r\n
Subject: I am a bomb!\r\n
$spaces
```

where $spaces can be a 4GB-long string of blank spaces[4].

3 Decompression Security Risks

Applications can use the Deflate decompression algorithm in three main ways. First, they can invoke the functions provided by widely available libraries, such as `zlib` for `C` and `java.util.zip` for Java. Second, they can adopt a high-level wrapper built around one of the previously mentioned libraries (e.g., the `zlib` module in Python or the `Zlib` module in PHP). Finally, applications can implement their own version of the Deflate algorithm. Either way, in this paper we show that it is not trivial to properly decompress user-generated data streams. The risk arises from three aspects of the compression/decompression process:

1. Decompression is a computationally intensive task entailing an extensive use of CPU, memory, and disk space. If not properly limited, this process can be abused to stall an application and cause a denial of service;

2. Decompression amplifies the amount of data that software needs to process. Other components may not be designed to handle this volume of data. Therefore, the use of other functionalities on compressed data may result in the exhaustion of resources for components that were otherwise considered secure;

3. Compressed data can often be pre-computed by an attacker, thus creating a largely unbalanced scenario in which the input can be sent very fast, but the server needs to invest a lot of resources to process it. Moreover, the compressed input can often be meaningless or even malformed, because applications are often designed to discard bad inputs only *after* they are entirely decompressed.

3.1 The Past: Zip Bombs and Billion Laughs

Abusing data amplification to cause application or system denial of services is an old trick. The first documented DoS attack via a highly-compressed file archive dates back to 1996 when an attacker uploaded a malicious compressed file archive (a zip bomb) to the Bulletin Board System (BBS) of Fidonet, waiting for the system administrator to decompress it [1]. The classic zip bomb was a 42-kilobyte zip file archive that contained five nested layer of compressed files whose total size amounted to 4.5 petabytes. In 2001, zip bombs were used by attackers as email attachments [57] to disable anti-virus software designed to scan incoming messages [2].

A second popular exploitation of data amplification flaws was the so-called Billion Laughs attack [49] in 2003 (CVE-2003-1564). The Billion Laughs attack, also called the Exponential Entity Expansion attack, is an attack that exploits resource exhaustion vulnerabilities of XML document parsers when processing recursive entity definitions. An attacker may exploit this behavior by crafting a valid XML document (an XML bomb) which will cause the parser to generate an exponential amount of data. This results in CPU monopolization and memory exhaustion that can be exploited to mount a denial of service attack. This vulnerability was first reported in 2003 as a weakness of `libxml2` (CVE-2003-1564), an XML parser library. The same vulnerability was later discovered in some network servers, e.g., in 2009 in WebDAV as implemented by Apache HTTPD (CVE-2009-1955), and in a number of XMPP servers in 2011 (see, for example, CVE-2011-1755 and CVE-2011-3288).

[4] Due to limitations of the IMAP servers, in our experiments we used a 2GB-long string.

3.2 The present

Despite the popularity of previous attacks, a quick look at current protocol specifications, coding rules, and design patterns suggests that developers remain mostly unaware of the risks of using data compression. The risks of using compression are often overlooked, and guidelines on the proper way to handle compressed messages are either misleading or completely missing. In the rest of this section we briefly describe what protocol specifications, design patterns, and secure coding practices mention about the security issues related to data compression. Then, in Section 4, we show how this lack of common knowledge and understanding about the possible decompression attack vectors leads to a multitude of mistakes in many popular applications and protocols. Finally, in Section 5, we show experiments and the software vulnerabilities that we discovered on our case studies.

3.2.1 Protocol Specifications

A closer look at the specifications of the case studies revealed that none of them discuss potential security issues related to the use of data compression at the protocol level.

The Deflate specifications are mainly concerned with data integrity issues, suggesting developers implement means of validating the integrity of compressed data [13]. The HTTP protocol is concerned with loss of data confidentiality and unauthorized access, e.g., via path traversal attacks [17]. HTTP also addresses other DoS-related issues, such as broken clients when handling the status code 100 and with HTTP proxies [17]. The XMPP stream compression [25] does not describe any XMPP-specific security concern due to the use of data compression. Instead, it refers to SSL/TLS [26], which is concerned with data leakage, buffer overrun in the compression library, and enforcing packet size limits for uncompressed data. However, the specifications do not elaborate on how these concerns apply to XMPP, and therefore the developer may be left to personal interpretation that, ultimately, translates into vulnerable implementations. Finally, the IMAP compression specification [23] refers again to the SSL/TLS specifications for everything related to the decompression process.

3.2.2 Security Design Patterns

Security design patterns [3] are used to prevent vulnerabilities during the software design phase as well as to mitigate security risks. They address security concerns at a high level of abstraction (i.e., *DoS Safety*, *Compartmentalization*, and *Small Process* [24]) but unfortunately lack the details to address the specific concerns at the implementation level.

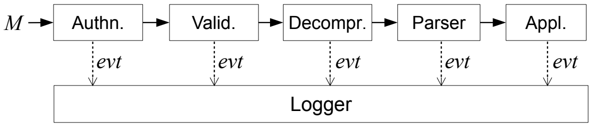


Figure 1: Message Pipeline

3.2.3 Secure Development Rules

Secure development rules suggest ways to write secure source code via secure code patterns (See, for example, coding rules for Java [29] and C++ [46]) or by means of software testing (see, for example, the OWASP Testing Guide [31]). The only existing rule on processing securely compressed data suggests a technique to validate a zip archive file before decompressing it. Sadly, the rule proposes an insecure technique, making the applications that implement it vulnerable to DoS via disk space exhaustion. We give more details of this rule in Section 4.2.

4 Common Pitfalls

In our study, we analyzed the implementation and documentation of our case studies, the protocol specifications (i.e., [11, 13, 17, 23, 25, 42]), and the software development best practices (i.e., [3, 24, 29, 31, 46]) looking for proper and incorrect ways to handle data compression. In this section, we distill our findings into 12 common pitfalls we observed at the implementation, specification, and configuration levels. Table 3 shows the list of pitfalls and maps them to the implementations of Table 1 that are affected by them.

4.1 Implementation Level

We start our survey of common compression-related mistakes by looking at the software implementation. As mentioned in Section 3, the decompression of a user-provided input is a delicate task that is prone to many errors. Software developers may be unaware of or underestimate the risks involved in this process, leading to implementation mistakes that can introduce denial of service vulnerabilities in the final product.

In this section, we list common pitfalls using a pipeline that processes incoming compressed requests. A pipeline is a linear chain of processing units in which the output of a unit is the input of the following one. Data pipelines are used to process incoming messages in blocks, in which each unit processes a single piece of information at the time, and provides an input to the next unit. Data compression can be used at different stages of message processing. For instance, it can be the first processing unit,

			Implementation						Specification			Configuration		
Prot.	Network Service		Impr. Input Val.	No Authn.	Int.-Unit Comm.	Log. Msgs.	Unbound. Mem.	Unbound. CPU	Misl. Doc.	Err. Best-Pract.	API Incons.	Insuf. Options	Default Values	Decentr. Pars.
XMPP	ejabberd		-	-	-	-	-	-	-	-	-	-	-	-
	Openfire		-	-	-	✗	✗	✗	-	-	-	✗	✗	-
	Prosody		-	✗	-	-	✗	✗	-	-	-	✗	-	-
	jabberd2		-	-	-	-	-	-	-	-	-	-	-	-
	Tigase		-	-	-	-	✗	✗	-	-	-	✗	✗	-
HTTP	Apache HTTPD	Static document	✗	-	-	-	✗	✗	✗	-	-	✗	-	-
		mod-php scripts	✗	-	✗	-	-	✗	✗	-	-	✗	-	-
		mod-php CSJRPC	✗	-	✗	-	-	✗	✗	-	-	✗	-	-
		mod-gsoap	✗	-	✗	-	-	✗	✗	-	-	✗	-	✗
		mod-dav	-	-	-	-	-	✗	-	-	-	-	-	-
	Apache Tomcat	Axis2	✗	-	-	-	✗	✗	✗	-	✗	✗	-	✗
		CXF	✗	-	-	✗	-	✗	✗	-	✗	✗	-	✗
		jsonrpc4j	✗	-	-	-	-	✗	✗	-	✗	✗	-	✗
		json-rpc	✗	-	-	-	✗	-	✗	-	✗	✗	-	✗
		lib-json-rpc	✗	-	-	-	✗	-	✗	-	✗	✗	-	✗
	Axis2 standalone		✗	-	-	-	✗	✗	✗	-	✗	✗	-	✗
	gSOAP standalone		✗	-	✗	-	-	✗	✗	-	✗	✗	-	✗
IMAP	Dovecot		-	-	-	-	-	-	-	-	-	-	-	-
	Cyrus		-	-	-	-	-	-	-	-	-	-	-	-

Table 3: Distribution of the pitfalls within the implementations under test

as it is for XMPP messages, or it can be placed after the message parsing, as implemented by the HTTP message processing.

Figure 1 shows a generic data pipeline to process an incoming message M. This pipeline is extracted from our case studies and is used to guide the description of the pitfalls. The pipeline has four processing units: user *authentication*, message *validation*, message *decompressor*, and message *parser*. The user authentication verifies that the request is sent by an authenticated user. This step may not be present in certain protocols, e.g., HTTP. The message validation unit implements a decision procedure to establish whether the incoming message can be accepted. The decompressor implements the data decompression algorithm to reconstruct the original message. Finally, the message parser performs a syntactic analysis of the message according to the rules of the communication protocol. As we already mentioned, the actual order of the blocks can be different, to reflect the protocol specification. The four units can send messages to the logger in order to store errors or unusual events in a log file. Finally, the output of the pipeline is consumed by the *application logic*. Here, by application we refer to an abstract representation of a message consumer. The consumer can be the rest of the software, e.g., a web application, as well as an additional component used to support application software, e.g., a web-based RPC framework.

4.1.1 Improper Input Validation during Decompression

One of the first mistakes we observed in our study is related to the erroneous way in which the size of incoming compressed messages is validated. We observed three ways to validate the message size: validation of the compressed message size, validation of the decompressed message size, and validation of the compression ratio of the message.

The first approach is the most straightforward to implement. Unfortunately, it is hard to estimate the size of a message by looking at its compressed form. For instance, while accepting an input no longer than 1MB could be insufficient for many types of data (e.g., when uploading a compressed picture), the same value is already sufficient for an attacker to generate extremely large decompressed output (e.g., by compressing a very repetitive string of bytes) that may cause an application denial of service.

The second approach consists in checking the size of the message by setting a limit to the amount of decompressed data. While this is a better solution, the validation of the decompressed size of an object is not straightforward to implement. To the best of our knowledge, we are not aware of any technique that allows one to compute the uncompressed size without first decompressing the input. Unfortunately, if the application needs to fully decompress the data before checking its size, it cannot protect itself against DoS attacks. However, many libraries (such as zlib and other language-specific wrappers) allow the application to decompress the incoming message in chunks. The size of each chunk is controlled by the application, which creates two buffers to store the compressed and the decompressed data streams. These two buffers are passed to the decompressor function of the library, which reads from one and writes to the other. The decompressor function returns when there are no more input bytes to process or when the output buffer is full. At this point, the application can provide more data or empty the output buffer. This interface allows an application to decompress a large message a piece at a time, constantly monitoring the amount of decompressed data. If a threshold is reached, the application can reject the input without the need to fully decompress the entire message.

The third approach consists of calculating the compression ratio. The compression ratio is the ratio between the sizes of the compressed and the decompressed messages. If the value exceeds a certain threshold, the decompression is halted and the message discarded. However, the compression ratio may vary according to the redundancies contained in the original message and it may cause the rejection of valid inputs. Moreover, the problem of deciding the appropriate threshold for the ratio must be solved. This may not be a simple task, as its value may depend on the protocol itself and on the way it is used by the application. Developers may leave the choice to educated guesses, experience, or experiments. This can result in under- or overestimation of this parameter. The former may increase the risk of rejecting valid messages, while the latter may introduce an uncontrolled use of resources.

To summarize, the first approach is a security mistake, the second is correct (if properly implemented), and the third one is potentially risky.

Apache HTTPD and gSOAP standalone provide examples of all three of these approaches. The first, vulnerable, approach is used by mod-deflate when it decompresses data for mod-php or mod-gsoap. This approach is insecure and we will discuss the details of this vulnerability in Section 5.4. When a message enters Apache HTTPD, it is processed by a chain of filters which transform the message and perform additional checks. The core module of Apache HTTPD offers a basic filter that allows setting a limit on the size of any incoming request body[5]. If the body exceeds the limit, then the message is rejected. However, the limit refers only to the compressed body size and it may render applications vulnerable to resource exhaustion.

The second approach is also implemented by mod-deflate, this time when it decompresses XML messages for mod-dav. The core module of Apache HTTPD offers a second parameter to limit the size of XML body objects[6]. As opposed to the previous one, the limit correctly applies to the decompressed form of the body. The third approach is now implemented by the patched versions of mod-deflate and gSOAP standalone. mod-deflate allows the user to configure a threshold for the compression ratio and the number of violations of the ratio that are allowed. gSOAP instead has the ratio built into the source code. In this case, the decompression is halted upon one violation.

4.1.2 Use of Compression before Authentication

We observed a great variety of practices for enforcing user authentication before the message decompression. For example, authentication is mandatory in SSL/TLS [14], recommended in XMPP [25], and undefined in IMAP [23]. In some cases, the implementation may even diverge from the specifications to postpone the use of compression (e.g., as done by OpenSSH), or to use compression where not prescribed (e.g., decompression of HTTP requests). This great variety of cases clearly indicates the lack of a consistent best practice. This may lead developers to underestimate the risk and overlook the recommendation. For example, we discovered that Prosody accepts compressed messages before user authentication, thus violating the recommendation of the XMPP protocol [25].

4.1.3 Improper Inter-Units Communication

When a processing unit in the pipeline detects a fault, i.e., the buffer limit is reached, then the unit should halt and notify the other units and the logger of this event. The communication between units can be direct or indirect via a third-party component, i.e., the pipeline manager. If a unit does not halt the execution, then the application may continue to consume resources until the resources are exhausted.

We observed this problem in Apache HTTPD in the interaction between mod-deflate and mod-php. mod-php can limit the size of the incoming request body via the parameter post_max_size. This parameter applies to the amount of data received in input by mod-php. However, if the incoming message is compressed, then it is first decompressed and then passed to mod-php. In this case, once the limit is reached, mod-php has no means to signal mod-deflate to stop processing the incoming data. As result, mod-deflate will keep on decompressing data, thus wasting system resources. The same problem was also observed between mod-deflate and mod-gsoap. The

[5] Via the configuration parameter LimitRequestBody

[6] Via the configuration parameter LimitXMLRequestBody

developers of mod-php and mod-gsoap confirmed this behavior.

4.1.4 Logging Decompressed Messages

Log files are used to store events generated by a running program (or an entire operating system), and they are particularly important to monitor the execution of background processes that have little interaction with the user, as is the case for web services. The information stored in log files can cover a wide range of events, including warning messages, malfunction errors, and verbose reports of the current activity of a given process.

While the use of log files is a good practice, both developers and users should carefully select the frequency and the verbosity of the generated events. An excessive level of verbosity may be useful to debug unusual behaviors, but it may have side effects from a security perspective. In particular, when the unusual behavior is caused by compressed data, developers and users may underestimate the resources needed to generate and store the event and, as a result, the application can exhaust all the available resources.

For instance, we observed this type of issue in Apache CXF. Upon receiving an invalid request, Apache CXF stores the request in a temporary file, and then adds an entry in the log file containing the first 100 KB of the request. However, if the invalid message is compressed, Apache CXF decompresses the entire request on disk just to extract the 100KB header to log. As a result, in our experiments we observed that a single request of 4MB (containing 4GB of data after decompression) can cause Apache CXF to store on the disk 8 GB of temporary data.

4.1.5 Unbounded Resource Usage

In general, the best way to avoid DoS attacks against an application is to properly limit the size of decompressed inputs. However, whenever such thresholds need to be set very high because the application has to accept large amounts of user-provided data, the developers should carefully design the code to bound the CPU and memory usage of the decompression routine. We found unbounded CPU and memory usage in different applications. Below, we discuss these pitfalls separately.

Unbounded Memory - The data amplification introduced by a decompressor may be underestimated by the developers who may leave buffers uncontrolled on the size both in peripheral (e.g., input validation components) and internal components (e.g., message parsers). For example, we observed this type of mistake in Prosody and json-rpc. Prosody decompresses incoming messages in chunks. Each chunk is passed to the XMPP message parser, which internally accumulates the chunks without bounds before the processing. A single highly-compressed message results in consuming all the available memory and, finally, being terminated by the operating system. A similar behavior was observed in json-rpc. Upon receiving a JSON request, json-rpc accumulates the uncompressed request into a memory buffer until no more memory is available.

Unbounded CPU - Decompressors are *CPU-bound* software procedures, as they tend to monopolize the CPU usage for the entire duration of the operation. If an attacker can influence their execution, then she may degrade system performance and mount a CPU DoS attack.

Unfortunately, best practices in secure software coding [29, 46] do not provide techniques for controlling CPU-bound tasks. As a result, developers may not be aware of the risks and leave the CPU usage unbounded. One way to control CPU-bound operations is to introduce idle time intervals in which a task is suspended for a period of time. The size of the interval and the moment in which it is introduced can be decided at run-time by taking into account the current status of the process. For example, a task may introduce idle times in order to keep constant the bandwidth of the decompressor throughput.

We observed an unbounded CPU usage in many implementations of our case studies, including Apache mod-deflate, Apache CXF, Webutilities, 2Way, Prosody, and Tigase. On the other hand, we found that CPU controls via idle time intervals were already implemented by ejabberd and jabberd2.

4.2 Specification Level

In this section, we review common data compression pitfalls stemming from imprecise protocol specifications, misleading documentation, and erroneous best practices.

4.2.1 Misleading Documentation

Modern software is a collection of reusable components. The developers of each component should carefully document the security risks related to the usage of their own components in order to allow a more secure integration. For this purpose, we reviewed the documentation of Apache mod-deflate, Webutilities, 2Way Filter, Axis 2, and gSOAP. None of the above components discuss the security risks related to the use of data compression. Even worse, the user documentation of Webutilities and 2Way even reassure their users that (i) a developer can plug in the decompression "*without changing the source code*" [37], and (ii) that "*nothing [else is] needed*" [32] from the user. In general, misleading documentation may create a false sense of security in which a developer may believe that she does not need to address the problem in her application because the possible security concerns are already addressed at the underlying level.

4.2.2 Erroneous Best Practices

Unawareness and underestimation of the risks in using data compression may also affect best practices. In our review of secure coding rules, we found out that the security risks specific to the decompression of compressed messages are not properly addressed. Design patterns are too generic to address the specificity of data compression, and secure code patterns address only the risks of storage exhaustion due to zip archive bombs [29]. Finally, testing guides only propose tests against information leakage vulnerabilities caused by the simultaneous usage of data compression and data encryption [31].

Interestingly enough, we discovered that the only available pattern on the topic is also insecure[7]. In fact, it suggests developers verify the decompressed size reported in the file headers before accepting a Zip archive. Unfortunately, this information can be easily forged by an attacker to contain any arbitrary value, thus successfully bypassing the security checks. We disclosed the flaw in the pattern to the authors, who marked it as vulnerable and provided a newer, secure version.

4.2.3 API Specifications Inconsistency

Data compression is an optional feature and it is transparent from the point of view of the application. However, in our review we found out that the use of compression may also violate the contract of other APIs. For example, the method `ServletRequest.getContentLength()` of J2EE 7 is supposed to return the length of the request body as it is made available through the input stream. The *input stream* refers to the object used by the servlet developer to access the content of the body of the message. This parameter may be used in the logic of the servlet to allocate a buffer, or to accept or reject the resource. Unfortunately, when the HTTP decompression is enabled, `getContentLength` returns a wrong value. In our experiments, we verified that `getContentLength` returns the value stored in the `Content-Length` HTTP header, while the input stream contains the much larger uncompressed body.

4.3 Configuration Level

In this section, we list common pitfalls in the way compression can be configured in different services.

4.3.1 Insufficient Configuration Options

In Section 4.1, we described a number of secure approaches to handle data compression at the implementation level. These solutions allow one to control the resource consumption by setting limits to the amount of resources to be used. The actual threshold may vary depending from a number of factors. For example, a web application that manages an online storage service may require the web server to accept large input messages to upload big files. In order to allow use of network services in different scenarios, the resource limits need to be parametric and the proper thresholds should be selected by the user during the deployment phase.

However, we observed that the number of configuration parameters provided by common servers is often insufficient. This is mainly due to the lack of implementations of the resource consumption controls. For example, Prosody only allows the use of data compression to be enabled or disabled; it does not offer any parameters to specify the maximum size of decompressed data to be accepted, nor the output bandwidth of the decompressor.

4.3.2 Insecure Default Values

Recently, it has been demonstrated that compression may be problematic when used together with data encryption, as it can lead to information leakage (e.g., CRIME [39] and BREACH [36]). The exploitation of these flaws may depend on the deployment scenario and the capability of the attacker to choose the plaintext. For these reasons, the use of data compression should be at the discretion of the user, who should assess the characteristics of the deployment scenario and the usage of the service.

While data compression should be an optional feature, in our survey we observed network service configurations in which data compression was enabled by default, such as in Openfire and Tigase.

4.3.3 Decentralized Configuration Parameters

The time to identify and resolve an attack is critical to contain the costs of a cyber-incident [34]. The response to an attack may require changing the configuration of a running system, and this task is simplified if the security-relevant configuration parameters are easily accessible to the security response team. In our survey, we verified that this is not always the case. For example, compression in CXF can be enabled in two ways. First, it can be enabled by adding the decompressor filter in the configuration of the servlet (i.e., the `web.xml` file). Second, it can be enabled within the Java code of the service using a Java annotation. In both cases, in order to disable the compression, the security response team needs to modify the configuration of all the servlets or, in the worst

[7] See https://www.securecoding.cert.org/confluence/display/java/IDS04-J.+Safely+extract+files+from+ZipInputStream

case, to modify even their source code. However, the response team may not have access to the source code. As a result, they may need to involve developers in this activity, which may increase the time required to react to an attack.

5 Analysis

In this section, we describe the experiments that we performed to detect resource exhaustion vulnerabilities. This section is organized as follows. In Section 5.1, we describe the experiment setup. Then, in Section 5.2 we present our results, and in Section 5.3 we discuss them. Finally, in Section 5.4, we present the complete list of vulnerabilities.

5.1 Experiment Setup

The services we tested in our experiments are developed using different programming languages, including C, C++, Java, PHP, Erlang, and Lua. While automatic code-based techniques could be used to detect software vulnerabilities, these techniques are language-dependent and therefore cannot be used in our analysis. For this reason, we followed a black-box testing approach in which we probed the implementation with malicious inputs, and then measured the resource consumption and the service availability.

Tests - Each test consists of two parts: *baseline test* and *attack*. The baseline test measures the point of reference for resource consumption and service availability when compression is not used and there are no attacks. Baseline measurements are sampled over a period of 60 seconds by probing the target with 4MB-long, honest protocol messages. The attack measures instead the resource consumption and service availability when stressing the implementation with malicious messages sent by one, 12, and 24 simultaneous attackers—a very low number compared with the number of clients that participate in most of the distributed DoS attacks. Malicious messages are presented in Section 2. Both the baseline and the attack requests have a 4MB payload in order to rule out the overhead of transferring the data over the network.

Testbed - We performed the experiments on a testbed of three machines to host, respectively, the server, the attackers, and the honest client. The server machine runs the IUT and the internal monitor, the attackers execute the test cases to send highly-compressed messages to the server, and the client executes baseline tests to measure the availability of the service.

To test the HTTP services, we used (i) a 4MB static file resource; (ii) two PHP scripts for mod-php, each using a different PHP interface[8] to read the content of the request body; and (iii) PHP, Java, and C++ classes and functions to be deployed as a web services with CSJRPC, Axis 2, CXF, jsonrpc4j, json-rpc, lib-json-rpc, and gSOAP (both mod-gsoap and standalone). To test the XMPP and the IMAP servers, we created user accounts for both the attackers and the honest client. In our tests, we considered different IUT configurations. For example, we tested Apache HTTPD, mod-php, Apache Tomcat, ejabberd, and jabberd2 with different maximum message sizes.

Prot.	Network Service		CPU	Mem.	Disk
XMPP	ejabberd		-	-	-
	Openfire		✗	✗	✗
	Prosody		✗	✗	-
	jabberd2		-	-	-
	Tigase		✗	✗	-
HTTP	Apache HTTPD	Static document	✗	✗	-
		mod-php scripts	✗	-	-
		mod-php CSJRPC	✗	-	-
		mod-gsoap	✗	-	-
		mod-dav	✗	-	-
	Apache Tomcat	Axis2	✗	✗	-
		CXF	✗	-	✗
		jsonrpc4j	✗	-	-
		json-rpc	-	✗	-
		lib-json-rpc	-	✗	-
	Axis2 standalone		✗	✗	-
	gSOAP standalone		✗	-	-
IMAP	Dovecot		-	-	✗
	Cyrus		-	-	✗

Table 4: Resource Consumption Vulnerabilities

Monitoring - We monitored the IUT with a combination of internal and external monitors. The external monitor measures the service availability in terms of number of honest messages processed per second. We used the client to continuously provide the server with honest messages and measure the server's response time. The internal monitor is a modified version of `pidstat` from the sysstat tool suite [20], which repeatedly polls the `/proc` filesystem. It measures (1) CPU usage, (2) virtual size (VSZ) and the resident set size (RSS) memory and (3) disk I/O of the processes associated with the IUT.

5.2 Results

Table 4 shows a summary of the results of our experiments on the 19 implementations. Out of them, only four implemented the compression in a secure way. Both ejabberd and jabberd2 keep a constant resource usage even during multiple simultaneous attacks. In fact, through a manual source code analysis, we were able to

[8] `php://input` interface and `$HTTP_RAW_POST_DATA`

verify that both servers implement two separate mechanisms to limit the use of memory and CPU usage during decompression. Table 4 shows a possible disk-based DoS attack against Dovecot and Cyrus; however, this is not to be considered a vulnerability, as IMAP servers are designed to store on disk the email used as the attack vector. All the other 15 services we tested showed an uncontrolled increase in at least one of the three system resources, making them potentially vulnerable to decompression-based DoS attacks. All results were reported to (and confirmed by) the developers of the corresponding applications and libraries.

Table 5 shows an excerpt of our experiments on three vulnerable implementations: Prosody, Apache HTTPD, and Apache CXF with the WebUtilities filter. For each implementation in Table 5, we performed four experiments (col. *Attackers*): the baseline and three attacks respectively with one, 12, and 24 parallel attackers. For each experiment, we report the requests response time (col. *Resp.*), the median value of the CPU usage (col. *Mdn*), the maximum virtual size memory allocated (col. *VSZ*), the maximum resident set size[9] allocated (col. *RSS*), and the total amount of data written to disk (col. *WR*). The columns *mult* report the ratio between the measured value during the attack and the baseline. In the rest of this section we detail the results of Table 5.

Prosody allocates up to 7.8GB of RSS memory and 22GB of VSZ memory when processing a single malicious request. The process is then killed by the operating system due to a system out of memory error. Even worse, Prosody also exhibits the same behavior when we sent the malicious message before the user authentication. Similarly as seen for Prosody, the measurements for Apache CXF show a significant resource utilization: starting from 0.03 GB of the baseline, Apache CXF can write about 1 TB, which is 3243 times the baseline. These value indicates that Apache CXF may be vulnerable to disk space exhaustion. Other services, while still potentially vulnerable, had a more controlled behavior. For instance, Apache HTTPD monopolizes the CPU at about 100% for 17 seconds with a single attacker, and up to 140 seconds by sending 24 malicious payloads in parallel.

5.3 Experiment Results Discussion

In this section, we discuss three factors that play an important role in our black-box experiments.

First, the baseline sets the amplitude of the proportions with the measurements done during the attack. The choice of the baseline is crucial because it can affect the conclusion of the analysis. As we already explained, the choice of the baseline was to offer a reference point that rules out network delay. This results in ratio values that cannot be directly transferred to real-size servers.

Second, the quantification of the severity of the observed degradation heavily depends on a number of variables that our testbed does not realistically reproduce, e.g., number of CPU cores, size of main memory, disk space, and average load of the server. Small-size servers can be DoSed with few requests, while large and powerful servers may be able to sustain a higher load before showing signs of resource exhaustion. As a result, to obtain an externally visible effect, it may be necessary to use a larger number of simultaneous attackers.

Finally, it is hard to develop an automated procedure to detect DoS vulnerabilities on the basis of the data we collected. The measures do not offer an accurate view of the internal behavior of the application, and the figures depend on so many factors that sometimes it is hard to make a final conclusion. For this reason, we manually verified each case, often complementing the experiments with a source code analysis of the affected components. Moreover, we discussed each problem with the developers, and obtained confirmation of each vulnerability reported in this paper.

5.4 Vulnerabilities

Our experiments led to the discovery of nine vulnerabilities. After we completed our experiments, our results were also reproduced on other three additional XMPP network services (M-Link, Metronome, and MongooseIM), discovering resource exhaustion vulnerabilities also in these products as well.

We followed the principle of responsible disclosure and informed the developers, the community, and the security response teams. In most of the cases, developers reacted to our first reports and worked on a patch. If developers were unresponsive for over a month, we tried a second time and then alerted the US CERT to support the disclosure. Eventually, all the developers acknowledge the reported vulnerabilities. The way in which each product was patched is described in the rest of this section.

5.4.1 HTTP

Apache HTTPD - The component that caused CPU and memory consumption is mod-deflate. This affected mod-php, CSJRPC, and mod-gsoap applications. Unfortunately, mod-php and mod-gsoap developers are unable to solve this issue on their components, as they are unaware of a suitable interface to control mod-deflate. As a result, we escalated the issue to the Apache Security Team. The security team acknowledged the presence of the vulnerability, fixed it in Apache HTTPD 2.4.10, and disclosed

[9] The total RAM size of the server machine is 8 GB.

		Attacks		CPU		Memory				Disk	
Prot.	Network Service	No.	Resp.	Mdn	mult.	RSS	mult.	VSZ	mult.	WR	mult.
XMPP	Prosody	*baseline*	> 0 s	1%	x1	0.01 GB	x1	0.05 GB	x1	0.00 GB	-
		1	204 s	18%	x18	7.68 GB	x1397	10.71 GB	x210	0.00 GB	-
		12	222 s	21%	x21	7.60 GB	x1383	18.40 GB	x361	0.00 GB	-
		24	531 s	34%	x34	7.71 GB	x1402	22.96 GB	x451	0.00 GB	-
HTTP	Apache HTTPD	*baseline*	1 s	21%	x1	0.05 GB	x1	0.55 GB	x1	0.00 GB	-
		1	17 s	114%	x6	0.10 GB	x2	0.78 GB	x1	0.00 GB	-
		12	72 s	297%	x14	0.58 GB	x11	2.36 GB	x4	0.00 GB	-
		24	142 s	229%	x11	0.84 GB	x15	2.75 GB	x5	0.00 GB	-
	Apache CXF WU	*baseline*	1 s	57%	x1	0.33 GB	x1	3.03 GB	x1	0.03 GB	x1
		1	149 s	55%	x1	0.38 GB	x1	3.03 GB	x1	9.10 GB	x317
		12	1135 s	109%	x2	0.73 GB	x2	6.09 GB	x2	84.90 GB	x2958
		24	1296 s	109%	x2	0.44 GB	x1	3.08 GB	x1	93.07 GB	x3243

Table 5: Excerpt of the experiment results

it publicly (CVE-2014-0118). Our contribution was also rewarded by the bounty program of Hackerone[10].

Apache HTTPD developers implemented two new mechanisms to control CPU and memory consumptions that can be configured via the following new parameters: `DeflateInflateLimitRequestBody`, `DeflateInflateRatioLimit`, and `DeflateInflateRatioBurst`. The first parameter limits the maximum amount of memory that can be allocated to decompress incoming HTTP requests. The second enforces a ratio between the compressed and decompressed message. This mechanism also allows specifying the number of tolerated violations of the ratio before halting the decompression. While this mechanism limits the use of resources in the presence of highly-compressed messages (not necessarily compression bombs), it does not limit the amount of CPU used by the decompressor.

gSOAP - gSOAP standalone suffers from uncontrolled CPU usage. The developers acknowledged the presence of the vulnerability and released a patch. The patch implements a ratio-based technique similar to the mechanism implemented now by mod-deflate. However, as opposed to mod-deflate, the ratio is not parametric but it is built into the source code.

Webutilities and 2Way - Webutilities and 2Way filters are the components that cause unbounded CPU usage in Axis 2, CXF, and jsonrpc4j. After our reports, the developers of Webutilities fixed the vulnerability using a CPU throttling mechanism that can be configured via a new parameter, `decompressMaxBytesPerSecond`. This mechanism monitors the throughput in bytes per second of the decompressor and, if the limit is reached, it introduces idle time intervals in which the decompressor is suspended from execution for a short period of time. The developers of 2Way acknowledged the presence of the issue and are working on a patch (still unavailable at the time of writing).

Apache CXF - Apache CXF suffers from a disk exhaustion vulnerability. We reported the issue to the Apache Security Team. The security team acknowledged the presence of the vulnerability in two branches of the software, fixed it in version 2.6.14 and 2.7.11, and disclosed it publicly (CVE-2014-0109 and CVE-2014-0110). This vulnerability is described in Section 4.1.4.

json-rpc and lib-json-rpc - json-rpc and lib-json-rpc suffer from an uncontrolled memory vulnerability. Upon receiving a JSON request, both frameworks try to store all of the uncompressed data in a single memory buffer, causing an out of memory error. The developers acknowledged the issue and are currently working on a patch.

5.4.2 XMPP

The disclosure of the XMPP vulnerabilities was conducted with the involvement of the XMPP community. We supported the community in coordinating the disclosure and preparing a common secure notice about the multiple vulnerabilities. In total, our experiments directly discovered four vulnerabilities in three XMPP servers, and three other servers were found vulnerable during the disclosure. All the vulnerabilities are fixed and new versions of the servers are already available[11]. Also in this case, our contribution and efforts were rewarded by the bounty program of Hackerone[12].

Openfire - Openfire does not properly restrict the resources used in processing incoming XMPP messages (see, CVE-2014-2741 and VU#495476).

Prosody - Prosody suffers from memory exhaustion due

[10] See https://hackerone.com/reports/20861

[11] See `http://xmpp.org/resources/security-notices/`

[12] See https://hackerone.com/reports/5928

to an uncontrolled buffer (CVE-2014-2745) and allows unauthenticated users to use data compression (CVE-2014-2744).

Tigase - Tigase does not properly limit the memory used to process incoming XMPP messages (CVE-2014-2746).

M-Link - M-Link does not properly restrict the resources used in processing incoming XMPP messages (CVE-2014-2742).

Metronome - Metronome suffers from unbounded memory consumption (CVE-2014-2743) and allows the use of compression before user authentication (CVE-2014-2744, shared with Prosody).

MongooseIM - MongooseIM does not properly restrict the resources used in processing incoming XMPP messages (CVE-2014-2829).

6 Related Work

In this section, we review works that are related to this paper from different perspectives. In particular, we discuss attacks that exploit data amplification, leakage due to the use of data compression, worst-case complexity, and bandwidth exhaustion.

Data Amplification Attacks - To the best of our knowledge, zip bombs [1] and XML bombs [49] are among the first documented abuses of data amplification. We already discussed the details of these attacks in Section 3. Data amplification can also be achieved by using external servers for which the response size is bigger than the request size [40]. An attacker can spoof the network address of a victim and sends small request packets to a large number of servers. These requests trigger voluminous response traffic that accumulates on the network link of the victim and leads to bandwidth exhaustion. Similar to an asymmetry in the request/response traffic volume, our work considers an asymmetry in creating and processing compressed data for attack amplification.

Compression and Encryption - Data compression can lead to information leakage when used together with encryption. CRIME [39] and BREACH [36] are two attacks that exploit the change of size of a ciphertext due to the compression of the plaintext. These attacks target the SSL/TLS layer when used to carry HTTP conversations and rely on an attacker that is capable of performing a chosen-plaintext attack. These attacks and our work show orthogonal security issues in using data compression. While CRIME and BEAST aim at breaking the SSL/TLS encryption layer, our paper addresses software vulnerabilities due to the data amplification of decompression algorithms.

Algorithmic Attacks - Resource exhaustion can also result from the worst-case performance of the data structure algorithms [12] and rule matching algorithms [47]. Similarly to these attacks, in this paper we exploit the worst-case scenario of decompression algorithms in which the attacker can cause resource exhaustion with a compression rate of 1:1000.

Bandwidth Exhaustion - A variety of DoS attacks target network bandwidth exhaustion [6, 28, 48]. The Coremelt attack [48] and the Crossfire attack [28] achieve DoS through network bandwidth exhaustion of a targeted Autonomous System backbone routers. The attacker does not connect to the victim, but instead she uses machines under her control to exchange data over the link used by the victim. As both Coremelt and Crossfire work by exposing network links to high traffic, their effects can arguably be mitigated by using compression. However, the results presented in this paper demonstrate that there is a catch to such strategies, as compression bears risks for the communication's end nodes. Büscher and Holz [6] show that the vast majority of attacks launched by the DirtJumper/Ruskill botnet target HTTP port 80 [6] and that an average number of 185 DoS threads is sufficient to saturate the link between the botnet and the victim. Our results indicate that service disruptions can be achieved with a much lesser number if compression is used, thereby circumventing the detection and analysis proposed in the paper [6].

7 Future Work

As future work, we plan to investigate two directions: secure data compression, and automated detection techniques for resource exhaustion vulnerabilities.

Solving the mistakes presented in Section 4 is only part of the problem. Data compression introduces an unbalanced scenario in which the sender can generate *offline* compressed messages while the receiver needs to perform *online* decompression. This gives a large advantage to the attacker. The first direction of our research is to tackle this problem by studying new techniques that introduce *fairness* in data compression. The idea is to allow the receiver to decompress an incoming message when it has evidence that the sender has performed the compression online. The evidence can be provided by the means of single-use or session-based compression keys.

The problem of detecting resource exhaustion vulnerabilities has been addressed in the past [4, 5, 10]. However, existing techniques are fragmented and suffer from a number of limitations which hinder their use: they cannot scale to real programs (See [5]), are not fully automated (See [4]), and can be applied only to a subset of the resource exhaustion vulnerabilities (See [10]). As a second research direction, we plan to develop an intelligent fuzzer which combines code analysis and blackbox testing: The code analysis can be responsible for extracting a set of program constraints that the fuzzer would then use to generate inputs using a constraint solver.

8 Conclusion

In this paper, we presented a study on the current use of data decompression in three popular network services that are at the core of modern web-based applications. We analyzed 19 network services and extensions, protocol specifications, and documentation looking for proper and incorrect ways to handle data compression. We grouped our findings into 12 common pitfalls that we observed at the implementation, specification, and configuration levels. Furthermore, in our tests we discovered and reported nine previously unknown vulnerabilities. While these problems have been now being patched, we believe that this paper shows how the risks of supporting data compression are still too often overlooked, even by very popular web and network services.

Acknowledgments

We thank the anonymous reviewers for the valuable comments. We also thank Gabriel Serme (SAP) for the valuable discussion that originated this line of research, and various members of the standardization and software development groups for fruitful discussions. In no specific order, we thank the Apache Security Team, Daniel Kulp (CXF), Stanislav Malyshev (PHP), Matthew Wild (Prosody), Andrzej Wójcik (Tigase), Guus der Kinderen (Openfire), Peter Saint-Andre (XMPP Standards Foundation), Philipp Hancke (XMPP Standards Foundation), Kevin Smith (XMPP Standards Foundation), Waqas Hussain (XMPP Standards Foundation), Robert van Engelen (gSOAP), and Rajendra Patil (Webutilities). This project was supported in part by the German Ministry for Education and Research (BMBF) through funding for the project 13N13250, H2020 ESCUDO-Cloud, and TU Darmstadt CASED/EC-SPRIDE.

References

[1] ACCESS DENIED. DFS Issue 55. `http://textfiles.com/magazines/DFS/dfs055.txt`, 1996.

[2] AERASEC NETWORK SERVICES AND SECURITY GMBH. Decompression Bomb Vulnerabilities. `http://www.aerasec.de/security/advisories/decompression-bomb-vulnerability.html`, 2009.

[3] ALUR, D., MALKS, D., AND CRUPI, J. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall PTR, 2001.

[4] ANTUNES, J., NEVES, N., AND VERISSIMO, P. Detection and prediction of resource-exhaustion vulnerabilities. In *ISSRE 2008* (Nov 2008), pp. 87–96.

[5] BURNIM, J., JUVEKAR, S., AND SEN, K. Wise: Automated test generation for worst-case complexity. In *ICSE 2009* (May 2009), pp. 463–473.

[6] BÜSCHER, A., AND HOLZ, T. Tracking DDoS Attacks: Insights into the business of disrupting the Web. In *Proc. LEET* (2012).

[7] BUTKO, A. JSONRPC Java implementation. `https://code.google.com/p/libjsonrpc/`, 2014.

[8] CARNEGIE MELLON UNIVERSITY. Project Cyrus. `https://cyrusimap.org/`, 2014.

[9] CAZI, M. CSJSONRPC - A PHP JSON-RPC Server. `https://github.com/mojtabacazi/CSJRPC`, 2014.

[10] CHANG, R., JIANG, G., IVANCIC, F., SANKARANARAYANAN, S., AND SHMATIKOV, V. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *CSF '09* (July 2009), pp. 186–199.

[11] CRISPIN, M. Internet Message Access Protocol - Version 4rev1. RFC 3501 (Proposed Standard), Mar. 2003. Updated by RFCs 4466, 4469, 4551, 5032, 5182, 5738, 6186, 6858.

[12] CROSBY, S. A., AND WALLACH, D. S. Algorithmic DoS. In *Encyclopedia of Cryptography and Security (2nd Ed.)*, H. C. A. van Tilborg and S. Jajodia, Eds. Springer, 2011, pp. 32–33.

[13] DEUTSCH, P. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951 (Informational), May 1996.

[14] DIERKS, T., AND ALLEN, C. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), Jan. 1999. Obsoleted by RFC 4346, updated by RFCs 3546, 5746, 6176.

[15] DILLEY, B. JSON-RPC for Java. `https://github.com/briandilley/jsonrpc4j`, 2014.

[16] DOVECOT SOLUTIONS OY. Dovecot - Secure IMAP Server. `http://www.dovecot.org/`, 2014.

[17] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785, 6266, 6585.

[18] Gailly, J.-L., and Adler, M. gzip. http://www.gzip.org/, 2014.

[19] Gailly, J.-L., and Adler, M. zlib. http://www.zlib.net/, 2014.

[20] Godard, S. SYSSTAT. http://sebastien.godard.pagesperso-orange.fr/, 2014.

[21] Goland, Y., Whitehead, E., Faizi, A., Carter, S., and Jensen, D. HTTP Extensions for Distributed Authoring – WEBDAV. RFC 2518 (Proposed Standard), Feb. 1999. Obsoleted by RFC 4918.

[22] Gordon, E., Spieler, C., White, M., and Haase, D. Info-ZIP. http://http://www.info-zip.org//, 2014.

[23] Gulbrandsen, A. The IMAP COMPRESS Extension. RFC 4978 (Proposed Standard), Aug. 2007.

[24] Hafiz, M., Adamczyk, P., and Johnson, R. E. Growing a pattern language (for security). In *Proc. Onward! '12* (2012), ACM, pp. 139–158.

[25] Hildebrand, J., and Saint-Andre, P. XEP-0138: Stream Compression. http://xmpp.org/extensions/xep-0138.html, 2009.

[26] Hollenbeck, S. Transport Layer Security Protocol Compression Methods. RFC 3749 (Proposed Standard), May 2004.

[27] Igniterealtime. Igniterealtime Openfire. http://www.igniterealtime.org/projects/openfire/, 2014.

[28] Kang, M. S., Lee, S. B., and Gligor, V. The Crossfire attack. In *Proc. SP* (2013), pp. 127–141.

[29] Long, F., Mohlndra, D., Seacord, R. C., Sutherland, D. F., and Svoboda, D. *The CERT Oracle Secure Coding Standard For Java*. SEI Series In Software Engineering. Addison-Wesley, 2012.

[30] Lyon, G. Nmap Security Scanner. http://nmap.org/, 2014.

[31] Meucci, M. et al. The OWASP Testing Guide 4.0. https://www.owasp.org/images/5/52/OWASP_Testing_Guide_v4.pdf, 2014.

[32] Patil, R. webutilities. https://code.google.com/p/webutilities/, 2014.

[33] Pereira, R. IP Payload Compression Using DEFLATE. RFC 2394 (Informational), Dec. 1998.

[34] Ponemon Institute LLC. 2013 Cost of Cyber Crime Study: United States. Tech. rep., Ponemon Institute LLC, Traverse City, Michigan 49629 USA, October 2013.

[35] Ponemon Institute LLC. 2014 Global Report on the Cost of Cyber Crime. Tech. rep., Ponemon Institute LLC, Traverse City, Michigan 49629 USA, October 2014.

[36] Prado, A., Harris, N., and Gluck, Y. SSL, Gone in 30 Seconds - A BREACH Beyond CRIME. http://breachattack.com/#resources, 2013.

[37] predic8 GmbH. 2Way HTTP Compression Servlet Filter. http://predic8.com/gzip-compression-filter.htm, 2014.

[38] ProcessOne. ejabberd - the Erlang Jabber/XMPP Daemon. http://www.ejabberd.im/, 2014.

[39] Rizzo, J., and Duong, T. The CRIME Attack. https://docs.google.com/a/trouge.net/presentation/d/11eBmGiHbYcHR9gL5nDyZChu_-lCa2GizeuOfaLU2HOU/edit#slide=id.g1e59c14c_1_54, 2012.

[40] Rossow, C. Amplification hell: Revisiting network protocols for ddos abuse. In *Proc. NDSS* (2014).

[41] Saikia, R. JsonRpc - Easy and Lightweight Json-Rpc Client/Server. https://github.com/RitwikSaikia/jsonrpc/, 2014.

[42] Saint-Andre, P. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 3920 (Proposed Standard), Oct. 2004. Obsoleted by RFC 6120, updated by RFC 6122.

[43] Saint-Andre, P. Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence. RFC 3921 (Proposed Standard), Oct. 2004. Obsoleted by RFC 6121.

[44] Saint-Andre, P. Mapping the Extensible Messaging and Presence Protocol (XMPP) to Common Presence and Instant Messaging (CPIM). RFC 3922 (Proposed Standard), Oct. 2004.

[45] Salomon, D. *Data Compression: The Complete Reference*. Springer-Verlang, 2007.

[46] Seacord, R. C. *Secure Coding In C And C++*. SEI Series In Software Engineering. Addison-Wesley, 2006.

[47] Smith, R., Estan, C., and Jha, S. Backtracking algorithmic complexity attacks against a NIDS. In *Proc. ACSAC* (2006), IEEE Computer Society, pp. 89–98.

[48] Studer, A., and Perrig, A. The Coremelt attack. In *Proc. ESORICS*, M. Backes and P. Ning, Eds., vol. 5789 of *LNCS*. Springer, 2009, pp. 37–52.

[49] Sullivan, B. XML Denial of Service Attacks and Defenses. http://msdn.microsoft.com/en-us/magazine/ee335713.aspx, 2009.

[50] The Apache Foundation. Apache Axis. http://axis.apache.org/, 2014.

[51] The Apache Foundation. Apache CXF: An Open-Source Services Framework. http://cxf.apache.org/, 2014.

[52] The Apache Foundation. Apache Tomcat. http://tomcat.apache.org/, 2014.

[53] The Apache Foundation. HTTP Server Project. http://httpd.apache.org/, 2014.

[54] The JabberD Team. JabberD XMPP Server. http://jabberd2.org/, 2014.

[55] The PHP Group. PHP. http://www.php.net/, 2014.

[56] The Prosody Team. Prosody IM. http://prosody.im/, 2014.

[57] The Register. DoS Risk from Zip of Death Attacks on AV Software? http://www.theregister.co.uk/2001/07/23/dos_risk_from_zip/, 2001.

[58] Tigase Inc. Open Source and Free XMPP/Jabber Software. http://www.tigase.org/, 2014.

[59] van Engelen, R. A. The gSOAP Toolkit for SOAP Web Services and XML-Based Applications. http://www.cs.fsu.edu/~engelen/soap.html, 2014.

[60] Woods, J. PPP Deflate Protocol. RFC 1979 (Informational), Aug. 1996.

Bohatei: Flexible and Elastic DDoS Defense

*Seyed K. Fayaz** *Yoshiaki Tobioka** *Vyas Sekar** *Michael Bailey*†
**CMU* †*UIUC*

Abstract

DDoS defense today relies on expensive and proprietary hardware appliances deployed at fixed locations. This introduces key limitations with respect to flexibility (e.g., complex routing to get traffic to these "chokepoints") and elasticity in handling changing attack patterns. We observe an opportunity to address these limitations using new networking paradigms such as software-defined networking (SDN) and network functions virtualization (NFV). Based on this observation, we design and implement Bohatei, a *flexible* and *elastic* DDoS defense system. In designing Bohatei, we address key challenges with respect to scalability, responsiveness, and adversary-resilience. We have implemented defenses for several DDoS attacks using Bohatei. Our evaluations show that Bohatei is scalable (handling 500 Gbps attacks), responsive (mitigating attacks within one minute), and resilient to dynamic adversaries.

1 Introduction

In spite of extensive industrial and academic efforts (e.g., [3, 41, 42]), distributed denial-of-service (DDoS) attacks continue to plague the Internet. Over the last few years, we have observed a dramatic escalation in the number, scale, and diversity of DDoS attacks. For instance, recent estimates suggest that over 20,000 DDoS attacks occur per day [44], with peak volumes of 0.5 Tbps [14, 30]. At the same time, new vectors [37, 55] and variations of known attacks [49] are constantly emerging. The damage that these DDoS attacks cause to organizations is well-known and include both monetary losses (e.g., $40,000 per hour [12]) and loss of customer trust.

DDoS defense today is implemented using expensive and proprietary hardware appliances (deployed in-house or in the cloud [8, 19]) that are *fixed* in terms of placement, functionality, and capacity. First, they are typically deployed at fixed network aggregation points (e.g., a peering edge link of an ISP). Second, they provide fixed functionality with respect to the types of DDoS attacks they can handle. Third, they have a fixed capacity with respect to the maximum volume of traffic they can process. This fixed nature of today's approach leaves network operators with two unpleasant options: (1) to overprovision by deploying defense appliances that can handle a high (but pre-defined) volume of every known attack type at each of the aggregation points, or (2) to deploy a smaller number of defense appliances at a central location (e.g., a scrubbing center) and reroute traffic to this location. While option (2) might be more cost-effective, it raises two other challenges. First, operators run the risk of underprovisioning. Second, traffic needs to be explicitly routed through a fixed central location, which introduces additional traffic latency and requires complex routing hacks (e.g., [57]). Either way, handling larger volumes or new types of attacks typically mandates purchasing and deploying new hardware appliances.

Ideally, a DDoS defense architecture should provide the *flexibility* to seamlessly place defense mechanisms where they are needed and the *elasticity* to launch defenses as needed depending on the type and scale of the attack. We observe that similar problems in other areas of network management have been tackled by taking advantage of two new paradigms: software-defined networking (SDN) [32, 40] and network functions virtualization (NFV) [43]. SDN simplifies routing by decoupling the control plane (i.e., routing policy) from the data plane (i.e., switches). In parallel, the use of virtualized network functions via NFV reduces cost and enables elastic scaling and reduced time-to-deploy akin to cloud computing [43]. These potential benefits have led major industry players (e.g., Verizon, AT&T) to embrace SDN and NFV [4, 6, 15, 23].[1]

In this paper, we present Bohatei[2], a flexible and

[1] To quote the SEVP of AT&T: "To say that we are both feet in [on SDN] would be an understatement. We are literally all in [4]."

[2] It means breakwater in Japanese, used to defend against tsunamis.

elastic DDoS defense system that demonstrates the benefits of these new network management paradigms in the context of DDoS defense. Bohatei leverages NFV capabilities to elastically vary the required scale (e.g., 10 Gbps vs. 100 Gbps attacks) and type (e.g., SYN proxy vs. DNS reflector defense) of DDoS defense realized by defense virtual machines (VMs). Using the flexibility of SDN, Bohatei steers suspicious traffic through the defense VMs while minimizing user-perceived latency and network congestion.

In designing Bohatei, we address three key algorithmic and system design challenges. First, the resource management problem to determine the number and location of defense VMs is NP-hard and takes hours to solve. Second, existing SDN solutions are fundamentally unsuitable for DDoS defense (and even introduce new attack avenues) because they rely on a per-flow orchestration paradigm, where switches need to contact a network controller each time they receive a new flow. Finally, an intelligent DDoS adversary can attempt to evade an elastic defense, or alternatively induce provisioning inefficiencies by dynamically changing attack patterns.

We have implemented a Bohatei controller using `OpenDaylight` [17], an industry-grade SDN platform. We have used a combination of open source tools (e.g., OpenvSwitch [16], Snort [48], Bro [46], iptables [13]) as defense modules. We have developed a scalable resource management algorithm. Our evaluation, performed on a real testbed as well as using simulations, shows that Bohatei effectively defends against several different DDoS attack types, scales to scenarios involving 500 Gbps attacks and ISPs with about 200 backbone routers, and can effectively cope with dynamic adversaries.

Contributions and roadmap: In summary, this paper makes the following contributions:

- Identifying new opportunities via SDN/NFV to improve the current DDoS defense practice (§2);
- Highlighting the challenges of applying existing SDN/NFV techniques in the context of DDoS defense(§3);
- Designing a responsive resource management algorithm that is 4-5 orders of magnitude faster than the state-of-the-art solvers (§4);
- Engineering a practical and scalable network orchestration mechanism using proactive tag-based forwarding that avoids the pitfalls of existing SDN solutions (§5);
- An adaptation strategy to handle dynamic adversaries that can change the DDoS attack mix over time (§6);
- A proof-of-concept implementation to handle several known DDoS attack types using industry-grade SDN/NFV platforms (§7); and
- A systematic demonstration of the scalability and effectiveness of Bohatei (§8).

We discuss related work (§9) before concluding (§10).

2 Background and Motivation

In this section, we give a brief overview of software-defined networking (SDN) and network functions virtualization (NFV) and discuss new opportunities these can enable in the context of DDoS defense.

2.1 New network management trends

Software-defined networking (SDN): Traditionally, network control tasks (e.g., routing, traffic engineering, and access control) have been tightly coupled with their data plane implementations (e.g., distributed routing protocols, ad hoc ACLs). This practice has made network management complex, brittle, and error-prone [32]. SDN simplifies network management by decoupling the network control plane (e.g., an intended routing policy) from the network data plane (e.g., packet forwarding by individual switches). Using SDN, a network operator can centrally program the network behavior through APIs such as OpenFlow [40]. This flexibility has motivated several real world deployments to transition to SDN-based architectures (e.g., [34]).

Network functions virtualization (NFV): Today, network functions (e.g., firewalls, IDSes) are implemented using specialized hardware. While this practice was necessary for performance reasons, it leads to high cost and inflexibility. These limitations have motivated the use of virtual network functions (e.g., a virtual firewall) on general-purpose servers [43]. Similar to traditional virtualization, NFV reduces costs and enables new opportunities (e.g., elastic scaling). Indeed, leading vendors already offer virtual appliance products (e.g., [24]). Given these benefits, major ISPs have deployed (or are planning to deploy) datacenters to run virtualized functions that replace existing specialized hardware [6, 15, 23]. One potential concern with NFV is low packet processing performance. Fortunately, several recent advances enable line-rate (e.g., 10-40Gbps) packet processing by software running on commodity hardware [47]. Thus, such performance concerns are increasingly a non-issue and will further diminish given constantly improving hardware support [18].

2.2 New opportunities in DDoS defense

Next, we briefly highlight new opportunities that SDN and NFV can enable for DDoS defense.

Lower capital costs: Current DDoS defense is based on specialized hardware appliances (e.g., [3, 20]). Network operators either deploy them on-premises, or outsource DDoS defense to a remote packet scrubbing site (e.g., [8]). In either case, DDoS defense is expensive.

For instance, based on public estimates from the General Services Administration (GSA) Schedule, a 10 Gbps DDoS defense appliance costs ≈$128,000 [11]. To put this in context, a commodity server with a 10 Gbps Network Interface Card (NIC) costs about $3,000 [10]. This suggests roughly 1-2 orders of magnitude potential reduction in capital expenses (ignoring software and development costs) by moving from specialized appliances to commodity hardware.[3]

Time to market: As new and larger attacks emerge, enterprises today need to frequently purchase more capable hardware appliances and integrate them into the network infrastructure. This is an expensive and tedious process [43]. In contrast, launching a VM customized for a new type of attack, or launching more VMs to handle larger-scale attacks, is trivial using SDN and NFV.

Elasticity with respect to attack volume: Today, DDoS defense appliances deployed at network chokepoints need to be provisioned to handle a predefined maximum attack volume. As an illustrative example, consider an enterprise network where a DDoS scrubber appliance is deployed at each ingress point. Suppose the projected resource footprint (i.e., defense resource usage over time) to defend against a SYN flood attack at times t_1, t_2, and t_3 is 40, 80, and 10 Gbps, respectively.[4] The total resource footprint over this entire time period is $3 \times max\{40,80,10\} = 240$ Gbps, as we need to provision for the worst case. However, if we could elastically scale the defense capacity, we would only introduce a resource footprint of $40+80+10 = 130$ Gbps—a 45% reduction in defense resource footprint. This reduced hardware footprint can yield energy savings and allow ISPs to repurpose the hardware for other services.

Flexibility with respect to attack types: Building on the above example, suppose in addition to the SYN flood attack, the projected resource footprint for a DNS amplification attack in time intervals t_1, t_2, and t_3 is 20, 40, and 80 Gbps, respectively. Launching only the required types of defense VMs as opposed to using monolithic appliances (which handle both attacks), drops the hardware footprint by 40%; i.e., from $3 \times (max\{40,80,10\} + max\{20,40,80\}) = 480$ to 270.

Flexibility with respect to vendors: Today, network operators are locked-in to the defense capabilities offered by specific vendors. In contrast, with SDN and NFV, they can launch appropriate best-of-breed defenses. For example, suppose vendor 1 is better for SYN flood defense, but vendor 2 is better for DNS flood defense. The physical constraints today may force an ISP to pick only one hardware appliance. With SDN/NFV we can avoid the undesirable situation of picking only one vendor and rather have a deployment with both types of VMs each for a certain type of attack. Looking even further, we also envision that network operators can mix and match capabilities from different vendors; e.g., if vendor 1 has better detection capabilities but vendor 2's blocking algorithm is more effective, then we can flexibly combine these two to create a more powerful defense platform.

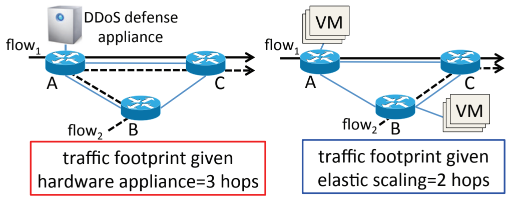

DDoS defense appliance
flow$_1$
A
C
flow$_2$
B
traffic footprint given hardware appliance=3 hops
VM
flow$_1$
A
C
flow$_2$
B
VM
traffic footprint given elastic scaling=2 hops

Figure 1: DDoS defense routing efficiency enabled by SDN and NFV.

Simplified and efficient routing: Network operators today need to employ complex routing hacks (e.g., [57]) to steer traffic through a fixed-location DDoS hardware appliance (deployed either on-premises or in a remote site). As Figure 1 illustrates, this causes additional latency. Consider two end-to-end flows $flow_1$ and $flow_2$. Way-pointing $flow_2$ through the appliance (the left hand side of the figure) makes the total path lengths 3 hops. But if we could launch VMs where they are needed (the right hand side of the figure), we could drop the total path lengths to 2 hops—a 33% decrease in traffic footprint. Using NFV we can launch defense VMs on the closest location to where they are currently needed, and using SDN we can flexibly route traffic through them.

In summary, we observe new opportunities to build a flexible and elastic DDoS defense mechanism via SDN/NFV. In the next section, we highlight the challenges in realizing these benefits.

3 System Overview

In this section, we envision the deployment model and workflow of Bohatei, highlight the challenges in realizing our vision, and outline our key ideas to address these challenges.

3.1 Problem scope

Deployment scenario: For concreteness, we focus on an ISP-centric deployment model, where an ISP offers DDoS-defense-as-a-service to its customers. Note that several ISPs already have such commercial offerings (e.g., [5]). We envision different monetization avenues. For example, an ISP can offer a value-added security service to its customers that can replace the customers' in-house DDoS defense hardware. Alternatively, the ISP can allow its customers to use Bohatei as a cloudbursting option when the attack exceeds the customers' on-

[3] Operational expenses are harder to compare due to the lack of publicly available data.

[4] For brevity, we use the traffic volume as a proxy for the memory consumption and CPU cycles required to handle the traffic.

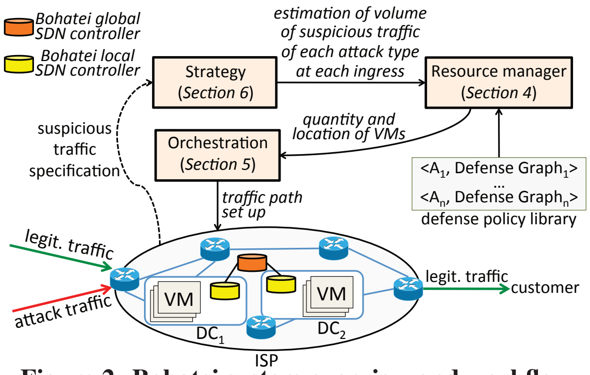

Bohatei global SDN controller
Bohatei local SDN controller
estimation of volume of suspicious traffic of each attack type at each ingress
Strategy (Section 6)
Resource manager (Section 4)
suspicious traffic specification
quantity and location of VMs
Orchestration (Section 5)
<A1, Defense Graph1>
...
<An, Defense Graphn>
defense policy library
traffic path set up
legit. traffic
attack traffic
VM
VM
DC1
DC2
legit. traffic
customer
ISP

Figure 2: Bohatei system overview and workflow.

premise hardware. While we describe our work in an ISP setting, our ideas are general and can be applied to other deployment models; e.g., CDN-based DDoS defense or deployments inside cloud providers [19].

In addition to traditional backbone routers and interconnecting links, we envision the ISP has deployed multiple datacenters as shown in Figure 2. Note that this is not a new requirement; ISPs already have several in-network datacenters and are planning additional rollouts in the near future [15, 23]. Each datacenter has commodity hardware servers and can run standard virtualized network functions [45].

Threat model: We focus on a general DDoS threat against the victim, who is a customer of the ISP. The adversary's aim is to exhaust the network bandwidth of the victim. The adversary can flexibly choose from a *set of candidate attacks* $AttackSet = \{A_a\}_a$. As a concrete starting point, we consider the following types of DDoS attacks: TCP SYN flood, UDP flood, DNS amplification, and elephant flow. We assume the adversary controls a large number of bots, but the total *budget* in terms of the maximum volume of attack traffic it can launch at any given time is fixed. Given the budget, the adversary has a complete control over the choice of (1) type and mix of attacks from the *AttackSet* (e.g., 60% SYN and 40% DNS) and (2) the set of ISP ingress locations at which the attack traffic enters the ISP. For instance, a simple adversary may launch a single fixed attack A_a arriving at a single ingress, while an advanced adversary may choose a mix of various attack types and multiple ingresses. For clarity, we restrict our presentation to focus on a single customer noting that it is straightforward to extend our design to support multiple customers.

Defenses: We assume the ISP has a pre-defined *library of defenses* specifying a defense strategy for each attack type. For each attack type A_a, the defense strategy is specified as a directed acyclic graph DAG_a representing a typical multi-stage attack analysis and mitigation procedure. Each node of the graph represents a logical module and the edges are tagged with the result of the previous nodes processing (e.g., "benign" or "attack" or "analyze further"). Each logical node will be realized by one (or more) *virtual appliance(s)* depending on the attack volume. Figure 3 shows an example strategy graph with 4 modules used for defending against a UDP flood attack. Here, the first module tracks the number of UDP packets each source sends and performs a simple threshold-based check to decide whether the source needs to be let through or throttled.

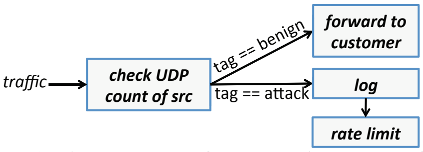


Figure 3: A sample defense against UDP flood.

Our goal here is not to develop new defense algorithms but to develop the system orchestration capabilities to enable flexible and elastic defense. As such, we assume the *DAG*s have been provided by domain experts, DDoS defense vendors, or by consulting best practices.

3.2 Bohatei workflow and challenges

The workflow of Bohatei has four steps (see Figure 2):

1. *Attack detection:* We assume the ISP uses some out-of-band anomaly detection technique to flag whether a customer is under a DDoS attack [27]. The design of this detection algorithm is outside the scope of this paper. The detection algorithm gives a coarse-grained specification of the suspicious traffic, indicating the customer under attack and some coarse identifications of the type and sources of the attack; e.g., "srcprefix=*,dstprefix=cust,type=SYN".
2. *Attack estimation:* Once suspicious traffic is detected, the strategy module estimates the volume of suspicious traffic of each attack type arriving at each ingress.
3. *Resource management:* The resource manager then uses these estimates as well as the library of defenses to determine the type, number, and the location of defense VMs that need to be instantiated. The goal of the resource manager is to efficiently assign available network resources to the defense while minimizing user-perceived latency and network congestion.
4. *Network orchestration:* Finally, the network orchestration module sets up the required network forwarding rules to steer suspicious traffic to the defense VMs as mandated by the resource manager.

Given this workflow, we highlight the three challenges we need to address to realize our vision:

C1. Responsive resource management: We need an efficient way of assigning the ISP's available compute and network resources to DDoS defense. Specifically, we need to decide how many VMs of each type to run on each server of each datacenter location so that attack traffic is handled properly while minimizing the latency experienced by legitimate traffic. Doing so in a *responsive* manner (e.g., within tens of seconds), however, is challenging. Specifically, this entails solving a large NP-hard optimization problem, which can take several hours to solve even with state-of-the-art solvers.

C2. Scalable network orchestration: The canonical view in SDN is to set up switch forwarding rules in a *per-flow* and *reactive* manner [40]. That is, every time a switch receives a flow for which it does not have a forwarding entry, the switch queries the SDN controller to get the forwarding rule. Unfortunately, this per-flow and reactive paradigm is fundamentally unsuitable for DDoS defense. First, an adversary can easily saturate the control plane bandwidth as well as the controller compute resources [54]. Second, installing per-flow rules on the switches will quickly exhaust the limited rule space (≈4K TCAM rules). Note that unlike traffic engineering applications of SDN [34], coarse-grained IP prefix-based forwarding policies would not suffice in the context of DDoS defense, as we cannot predict the IP prefixes of future attack traffic.

C3. Dynamic adversaries: Consider a dynamic adversary who can rapidly change the attack mix (i.e., attack type, volume, and ingress point). This behavior can make the ISP choose between two undesirable choices: (1) wasting compute resources by overprovisioning for attack scenarios that may not ever arrive, (2) not instantiating the required defenses (to save resources), which will let attack traffic reach the customer.

3.3 High-level approach

Next we highlight our key ideas to address C1–C3:

- **Hierarchical optimization decomposition** (§**4**)**:** To address C1, we use a hierarchical decomposition of the resource optimization problem into two stages. First, the Bohatei global (i.e., ISP-wide) controller uses coarse-grained information (e.g., total spare capacity of each datacenter) to determine how many and what types of VMs to run in each datacenter. Then, each local (i.e., per-datacenter) controller uses more fine-grained information (e.g., location of available servers) to determine the specific server on which each defense VM will run.
- **Proactive tag-based forwarding** (§**5**)**:** To address C2, we design a scalable orchestration mechanism using two key ideas. First, switch forwarding rules are based on per-VM tags rather than per-flow to dramatically reduce the size of the forwarding tables. Second, we proactively configure the switches to eliminate frequent interactions between the switches and the control plane [40].

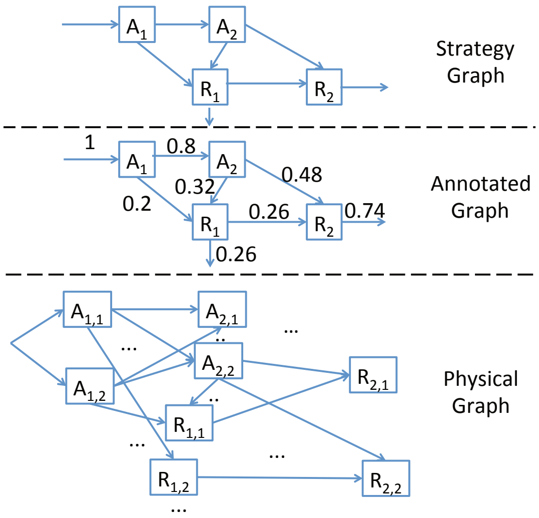


Figure 4: An illustration of strategy vs. annotated vs. physical graphs. Given annotated graphs and suspicious traffic volumes, the resource manager computes physical graphs.

- **Online adaptation** (§**6**)**:** To handle a dynamic adversary that changes the attack mix (C3), we design a defense strategy adaptation approach inspired by classical online algorithms for regret minimization [36].

4 Resource Manager

The goal of the resource management module is to efficiently determine network and compute resources to analyze and take action on suspicious traffic. The key here is responsiveness—a slow algorithm enables adversaries to nullify the defense by rapidly changing their attack characteristics. In this section, we describe the optimization problem that Bohatei needs to solve and then present a scalable heuristic that achieves near optimal results.

4.1 Problem inputs

Before we describe the resource management problem, we establish the main input parameters: the ISP's compute and network parameters and the defense processing requirements of traffic of different attack types. We consider an ISP composed of a set of edge PoPs[5] $E = \{E_e\}_e$ and a set of datacenters $D = \{D_d\}_d$.

ISP constraints: Each datacenter's traffic processing capacity is determined by a pre-provisioned uplink capacity C_d^{link} and compute capacity $C_d^{compute}$. The compute capacity is specified in terms of the number of VM slots, where each VM slot has a given capacity specification (e.g., instance sizes in EC2 [2]).

Processing requirements: As discussed earlier in §3.1, different attacks require different *strategy graphs*. However, the notion of a strategy graph by itself will not suf-

[5] We use the terms "edge PoP" and "ingress" interchangeably.

fice for resource management, as it is does not specify the traffic volume that at each module should process.

The input to the resource manager is in form of *annotated graphs* as shown in Figure 4. An annotated graph $DAG_a^{annotated}$ is a strategy graph annotated with edge weights, where each weight represents the fraction of the total input traffic to the graph that is expected to traverse the corresponding edge. These weights are precomputed based on prior network monitoring data (e.g., using NetFlow) and from our adaptation module (§6). $T_{e,a}$ denotes the volume of suspicious traffic of type a arriving at edge PoP e. For example, in Figure 4, weight 0.48 from node A_2 to node R_2 means 48% of the total input traffic to the graph (i.e., to A_1) is expected to traverse edge $A_2 \rightarrow R_2$.

Since modules may vary in terms of compute complexity and the traffic rate that can be handled per VM-slot, we need to account for the parameter $P_{a,i}$ that is the traffic processing capacity of a VM (e.g., in terms of compute requirements) for the logical module $v_{a,i}$, where $v_{a,i}$ is node i of graph $DAG_a^{annotated}$.

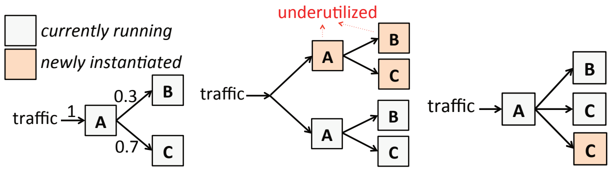
Network footprint: We denote the network-level cost of transferring the unit of traffic from ingress e to datacenter d by $L_{e,d}$; e.g., this can represent the path latency per byte of traffic. Similarly, within a datacenter, the units of intra-rack and inter-rack traffic costs are denoted by *IntraUnitCost* and *InterUnitCost*, respectively (e.g., they may represent latency such that *IntraUnitCost* < *InterUnitCost*).

4.2 Problem statement

Our resource management problem is to translate the annotated graph into a *physical graph* (see Figure 4); i.e., each node i of the annotated graph $DAG_a^{annotated}$ will be realized by one or more VMs each of which implement the logical module $v_{a,i}$.

Fine-grained scaling: To generate physical graphs given annotated graphs in a resource-efficient manner, we adopt a *fine-grained scaling* approach, where each logical module is scaled independently. We illustrate this idea in Figure 5. Figure 5a shows an annotated graph with three logical modules A, B, and C, receiving different amounts of traffic and consuming different amounts of compute resources. Once implemented as a physical graph, suppose module C becomes the bottleneck due to its processing capacity and input traffic volume. Using a monolithic approach (e.g., running A, B, and C within a single VM), we will need to scale the entire graph as shown in Figure 5b. Instead, we decouple the modules to enable scaling out individual VMs; this yields higher resource efficiency as shown in Figure 5c.

Goals: Our objective here is to (a) instantiate the VMs across the compute servers throughout the ISP, and (b) distribute the processing load across these servers to minimize the expected latency for legitimate traffic. Further, we want to achieve (a) and (b) while minimizing the footprint of suspicious traffic.[6]



(a) Annotated graph. (b) Monolithic. (c) Fine-grained.

Figure 5: An illustration of fine-grained elastic scaling when module C becomes the bottleneck.

To this end, we need to assign values to two key sets of decision variables: (1) the fraction of traffic $T_{e,a}$ to send to each datacenter D_d (denoted by $f_{e,a,d}$), and (2) the number of VMs of type $v_{a,i}$ to run on server s of datacenter D_d. Naturally, these decisions must respect the datacenters' bandwidth and compute constraints.

Theoretically, we can formulate this resource management problem as a constrained optimization via an Integer Linear Program (ILP). For completeness, we describe the full ILP in Appendix A. Solving the ILP formulation gives an optimal solution to the resource management problem. However, if the ILP-based solution is incorporated into Bohatei, an adversary can easily overwhelm the system. This is because the ILP approach takes several hours (see Table 2). By the time it computes a solution, the adversary may have radically changed the attack mix.

4.3 Hierarchical decomposition

To solve the resource management problem, we decompose the optimization problem into two subproblems: (1) the Bohatei global controller solves a *Datacenter Selection Problem (DSP)* to choose datacenters responsible for processing suspicious traffic, and (2) given the solution to the DSP, each local controller solves a *Server Selection Problem (SSP)* to assign servers inside each selected datacenter to run the required VMs. This decomposition is naturally scalable as the individual SSP problems can be solved independently by datacenter controllers. Next, we describe practical greedy heuristics for the DSP and SSP problems that yield close-to-optimal solutions (see Table 2).

Datacenter selection problem (DSP): We design a greedy algorithm to solve DSP with the goal of reducing ISP-wide suspicious traffic footprint. To this end, the algorithm first sorts suspicious traffic volumes (i.e.,

[6] While it is possible to explicitly minimize network congestion [33], minimizing suspicious traffic footprint naturally helps reduce network congestion as well.

$T_{e,a}$ values) in a decreasing order. Then, for each suspicious traffic volume $T_{e,a}$ from the sorted list, the algorithm tries to assign the traffic volume to the datacenter with the least cost based on $L_{e,d}$ values. The algorithm has two outputs: (1) $f_{e,a,d}$ values denoting what fraction of suspicious traffic from each ingress should be steered to each datacenter (as we will see in §5, these values will be used by network orchestration to steer traffic correspondingly), (2) the physical graph corresponding to attack type a to be deployed by datacenter d. For completeness, we show the pseudocode for the DSP algorithm in Figure 16 in Appendix B.

Server selection problem (SSP): Intuitively, the SSP algorithm attempts to preserve traffic locality by instantiating nodes adjacent in the physical graph as close as possible within the datacenter. Specifically, given the physical graph, the SSP algorithm greedily tries to assign nodes with higher capacities (based on $P_{a,i}$ values) along with its predecessors to the same server, or the same rack. For completeness we show the pseudocode for the SSP algorithm in Figure 17 in Appendix B.

5 Network Orchestration

Given the outputs of the resource manager module (i.e., assignment of datacenters to incoming suspicious traffic and assignment of servers to defense VMs), the role of the network orchestration module is to configure the network to implement these decisions. This includes setting up forwarding rules in the ISP backbone and inside the datacenters. The main requirement is scalability in the presence of attack traffic. In this section, we present our *tag-based* and *proactive* forwarding approach to address the limitations of the per-flow and reactive SDN approach.

5.1 High-level idea

As discussed earlier in §3.2, the canonical SDN view of setting up switch forwarding rules in a per-flow and reactive manner is not suitable in the presence of DDoS attacks. Furthermore, there are practical scalability and deployability concerns with using SDN in ISP backbones [21, 29]. There are two main ideas in our approach to address these limitations:

- Following the hierarchical decomposition in resource management, we also decompose the network orchestration problem into two-sub-problems: (1) Wide-area routing to get traffic to datacenters, and (2) Intra-datacenter routing to get traffic to the right VM instances. This decomposition allows us to use different network-layer techniques; e.g., SDN is more suitable inside the datacenter while traditional MPLS-style routing is better suited for wide-area routing.
- Instead of the controller reacting to each flow arrival, we *proactively* install forwarding rules before traffic arrives. Since we do not know the specific IP-level suspicious flows that will arrive in the future, we use logical *tag-based* forwarding rules with per-VM tags instead of per-flow rules.

5.2 Wide-area orchestration

The Bohatei global controller sets up forwarding rules on backbone routers so that traffic detected as suspicious is steered from edge PoPs to datacenters according to the resource management decisions specified by the $f_{e,a,d}$ values (see §4.3).[7]

To avoid a forklift upgrade of the ISP backbone and enable an immediate adoption of Bohatei, we use traditional tunneling mechanisms in the backbone (e.g., MPLS or IP tunneling). We proactively set up static tunnels from each edge PoP to each datacenter. Once the global controller has solved the DSP problem, the controller configures backbone routers to split the traffic according to the $f_{e,a,d}$ values. While our design is not tied to any specific tunneling scheme, the widespread use of MPLS and IP tunneling make them natural candidates [34].

5.3 Intra-datacenter orchestration

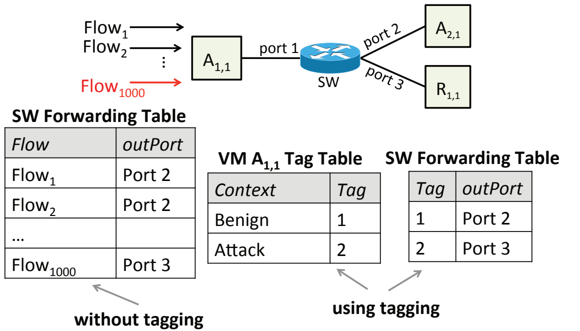
Inside each datacenter, the traffic needs to be steered through the intended sequence of VMs. There are two main considerations here:

1. The next VM a packet needs to be sent to depends on the *context* of the current VM. For example, the node *check UDP count of src* in the graph shown in Figure 3 may send traffic to either *forward to customer* or *log* depending on its analysis outcome.
2. With elastic scaling, we may instantiate several physical VMs for each logical node depending on the demand. Conceptually, we need a “load balancer” at every level of our annotated graph to distribute traffic across different VM instances of a given logical node.

Note that we can trivially address both requirements using a per-flow and reactive solution. Specifically, a local controller can track a packet as it traverses the physical graph, obtain the relevant context information from each VM, and determine the next VM to route the traffic to. However, this approach is clearly not scalable and can introduce avenues for new attacks. The challenge here is to meet these requirements without incurring the overhead of this per-flow and reactive approach.

Encoding processing context: Instead of having the controller track the context, our high-level idea is to en-

[7] We assume the ISP uses legacy mechanisms for forwarding non-attack traffic and traffic to non-Bohatei customers, so these are not the focus of our work.



SW Forwarding Table

Flow	outPort
Flow$_1$	Port 2
Flow$_2$	Port 2
…	
Flow$_{1000}$	Port 3

VM $A_{1,1}$ Tag Table

Context	Tag
Benign	1
Attack	2

SW Forwarding Table

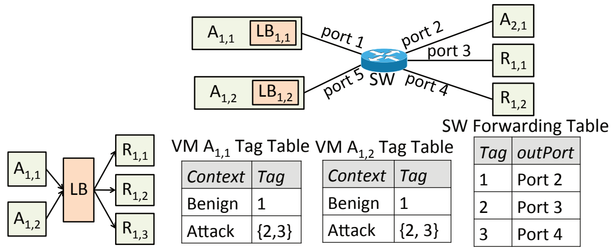
Tag	outPort
1	Port 2
2	Port 3

Figure 6: Context-dependent forwarding using tags.

code the necessary context as *tags* inside packet headers [31]. Consider the example shown in Figure 6 composed of VMs $A_{1,1}$, $A_{2,1}$, and $R_{1,1}$. $A_{1,1}$ encodes the processing context of outgoing traffic as tag values embedded in its outgoing packets (i.e., tag values 1 and 2 denote benign and attack traffic, respectively). The switch then uses this tag value to forward each packet to the correct next VM.

Tag-based forwarding addresses the control channel bottleneck and switch rule explosion. First, the tag generation and tag-based forwarding behavior of each VM and switch is configured proactively once the local controller has solved the SSP. We proactively assign a tag for each VM and populate forwarding rules before flows arrive; e.g., in Figure 6, the tag table of $A_{1,1}$ and the forwarding table of the router have been already populated as shown. Second, this reduces router forwarding rules as illustrated in Figure 6. Without tagging, there will be one rule for each of the 1000 flows. Using tag-based forwarding, we achieve the same forwarding behavior using only two forwarding rules.

Scale-out load balancing: One could interconnect VMs of the same physical graph as shown in Figure 7a using a dedicated load balancer (load balancer). However, such a load balancer may itself become a bottleneck, as it is on the path of every packet from any VM in the set $\{A_{1,1}, A_{1,2}\}$ to any VM in the set $\{R_{1,1}, R_{1,2}, , R_{1,3}\}$. To circumvent this problem, we implement the distribution strategy *inside each VM* so that the load balancer capability scales proportional to the current number of VMs. Consider the example shown in Figure 7b where due to an increase in attack traffic volume we have added one more VM of type A_1 (denoted by $A_{1,2}$) and one more VM of type R_1 (denoted by $R_{1,2}$). To load balance traffic between the two VMs of type R_1, the load balancer of A_1 VMs (shown as $LB_{1,1}$ and $LB_{1,2}$ in the figure) pick a tag value from a *tag pool* (shown by $\{2,3\}$ in the figure) based on the processing context of the outgoing packet and the intended load balancing scheme (e.g., uniformly at random to distribute load equally). Note that this tag pool is pre-populated by the local controller (given the defense library and the output of the resource manager module). This scheme, thus, satisfies the load balancing requirement in a scalable manner.



VM $A_{1,1}$ Tag Table

Context	Tag
Benign	1
Attack	{2,3}

VM $A_{1,2}$ Tag Table

Context	Tag
Benign	1
Attack	{2, 3}

SW Forwarding Table

Tag	outPort
1	Port 2
2	Port 3
3	Port 4

(a) A naive load balancer design.

(b) A distributed load balancer design.

Figure 7: Different load balancer design points.

Other issues: There are two remaining practical issues:

- *Number of tag bits:* We give a simple upper bound on the required number of bits to encode tags. First, to support context-dependent forwarding out of a VM with k relevant contexts, we need k distinct tag values. Second. to support load balancing among l VMs of the same logical type, each VM needs to be populated with a tag pool including l tags. Thus, at each VM we need at most $k \times l$ distinct tag values. Therefore, an upper bound on the total number of unique tag values is $k_{max} \times l_{max} \times \sum_{a} | V_a^{annotated} |$, where k_{max} and l_{max} are the maximum number of contexts and VMs of the same type in a graph, and $V_a^{annotated}$ is the set of vertices of annotated graph for attack type a. To make this concrete, across the evaluation experiments §8, the maximum value required tags was 800, that can be encoded in $\lceil log_2(800) \rceil = 10$ bits. In practice, this tag space requirement of Bohatei can be easily satisfied given that datacenter grade networking platforms already have extensible header fields [56].
- *Bidirectional processing:* Some logical modules may have bidirectional semantics. For example, in case of a DNS amplification attack, request and response traffic must be processed by the same VM. (In other cases, such as the UDP flood attack, bidirectionality is not required.). To enforce bidirectionality, ISP edge switches use tag values of outgoing traffic so that when the corresponding incoming traffic comes back, edge switches sends it to the datacenter within which the VM that processed the outgoing traffic is located. Within the datacenter, using this tag value, the traffic is steered to the VM.

6 Strategy Layer

As we saw in §4, a key input to the resource manager module is the set of $T_{e,a}$ values, which represents the volume of suspicious traffic of each attack type a arriving at each edge PoP e. This means we need to estimate the future attack mix based on observed measurements of the network and then instantiate the required defenses. We begin by describing an adversary that intends to thwart a Bohatei-like system. Then, we discuss limitations of strawman solutions before describing our *online adaptation* mechanism.

Interaction model: We model the interaction between the ISP running Bohatei and the adversary as a repeated interaction over several *epochs*. The ISP's "move" is one epoch behind the adversary; i.e., it takes Bohatei an epoch to react to a new attack scenario due to implementation delays in Bohatei operations. The epoch duration is simply the sum of the time to detect the attack, run the resource manager, and execute the network orchestration logic. While we can engineer the system to minimize this lag, there will still be non-zero delays in practice and thus we need an adaptation strategy.

Objectives: Given this interaction model, the ISP has to pre-allocate VMs and hardware resources for a specific attack mix. An intelligent and dynamic adversary can change its attack mix to meet two goals:

G1 *Increase hardware resource consumption:* The adversary can cause ISP to overprovision defense VMs. This may impact the ISP's ability to accommodate other attack types or reduce profits from other services that could have used the infrastructure.

G2 *Succeed in delivering attack traffic:* If the ISP's detection and estimation logic is sub-optimal and does not have the required defenses installed, then the adversary can maximize the volume of attack traffic delivered to the target.

The adversary's goal is to maximize these objectives, while the ISPs goal is to minimize these to the extent possible. One could also consider a third objective of collateral damage on legitimate traffic; e.g., introduce needless delays. We do not discuss this dimension because our optimization algorithm from §4 will naturally push the defense as close to the ISP edge (i.e., traffic ingress points) as possible to minimize the impact on legitimate traffic.

Threat model: We consider an adversary with a fixed budget in terms of the total volume of attack traffic it can launch at any given time. Note that the adversary can apportion this budget across the *types* of attacks and the *ingress* locations from which the attacks are launched. Formally, we have $\sum\limits_{e}\sum\limits_{a} T_{e,a} \leq B$, but there are no constraints on the specific $T_{e,a}$ values.

Limitations of strawman solutions: For simplicity, let us consider a single ingress point. Let us consider a strawman solution called *PrevEpoch* where we measure the attack observed in the previous epoch and use it as the estimate for the next epoch. Unfortunately, this can have serious issues w.r.t. goals **G1** and **G2**. To see why, consider a simple scenario where we have two attack types with a budget of 30 units and three epochs with the attack volumes as follows: T1: A1= 10, A2=0; T2: A1=20, A2=0; T3: A1=0; A2=30. Now consider the *PrevEpoch* strategy starting at the 0,0 configuration. It has a total wastage of 0,0,20 units and a total evasion of 10,10,30 units because it has overfit to the previous measurement. We can also consider other strategies; e.g., a *Uniform* strategy that provisions 15 units each for A1 and A2 or extensions of these to *overprovision* where we multiply the number of VMs given by the resource manager in the last epoch by a fixed value $\gamma > 1$. However, these suffer from the same problems and are not competitive.

Online adaptation: Our metric of success here is to have *low regret* measured with respect to the best static solution computed in hindsight [36]. Note that in general, it is not possible to be competitive w.r.t. the best dynamic solution since that presumes oracle knowledge of the adversary, which is not practical.

Intuitively, if we have a non-adaptive adversary, using the observed empirical average is the best possible static hindsight estimation strategy; i.e., $T^*_{e,a} = \frac{\sum_t T_{e,a,t}}{|t|}$ would be the optimal solution ($|t|$ denotes the total number of epochs). However, an attacker who knows that we are using this strategy can game the system by changing the attack mix. To address this, we use a follow the perturbed leader (FPL) strategy [36] where our estimation uses a combination of the past observed behavior of the adversary and a randomized component. Intuitively, the random component makes it impossible for the attacker to predict the ISP's estimates. This is a well-known approach in online algorithms to minimize the regret [36]. Specifically, the traffic estimates for the *next* epoch $t+1$, denoted by $\widehat{T_{e,a,t+1}}$ values, are calculated based on the average of the past values plus a random component: $\widehat{T_{e,a,t+1}} = \frac{\sum_{t'=1}^{t} T_{e,a,t'}}{|t|} + randperturb$.

Here, $T_{e,a,t'}$ is the empirically observed value of the attack traffic and *randperturb* is a random value drawn uniformly from $[0, \frac{2\times B}{nextEpoch \times |E| \times |A|}]$. (This is assuming a total defense of budget of $2 \times B$.) It can be shown that this is indeed a provably good regret minimization strategy [36]; we do not show the proof for brevity.

7 Implementation

In this section, we briefly describe how we implemented the key functions described in the previous sections. We have made the source code available [1].

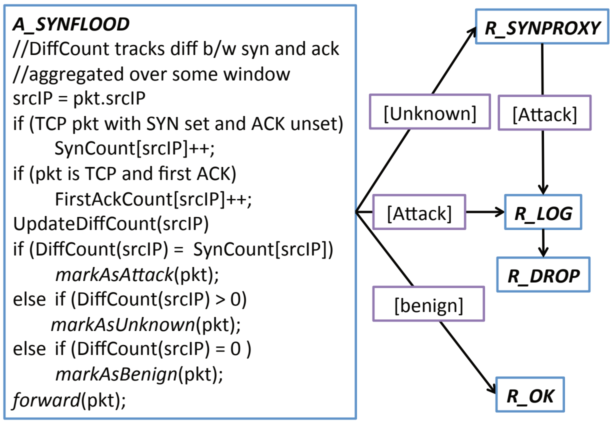
7.1 DDoS defense modules

The design of the Bohatei strategy layer is inspired by the prior modular efforts in Click [7] and Bro [46]. This modularity has two advantages. First, it allows us to adopt best of breed solutions and compose them for different attacks. Second, it enables more fine-grained scaling. At a high level, there are two types of logical building blocks in our defense library:

1. *Analysis (A)*: Each analysis module processes a suspicious flow and determines appropriate action (e.g., more analysis or specific response). It receives a packet and outputs a tagged packet, and the tags are used to steer traffic to subsequent analysis and response module instances as discussed earlier.
2. *Response (R)*: The input to an R module is a tagged packet from some A module. Typical responses include forward to customer (for benign traffic), log, drop, and rate limit. Response functions will depend on the type of attack; e.g., sending RST packets in case of a TCP SYN attack.

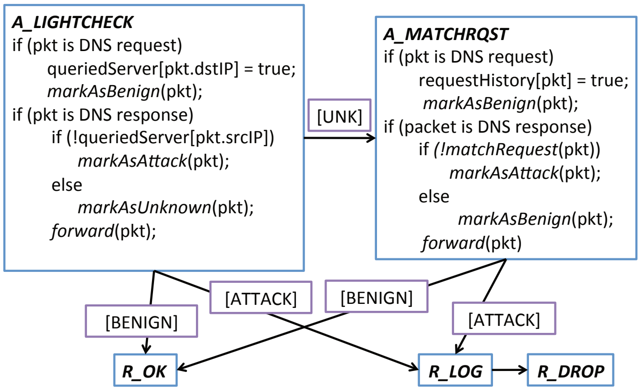
Next, we describe defenses we have implemented for different DDoS attacks. Our goal here is to illustrate the flexibility Bohatei provides in dealing with a diverse set of known attacks rather than develop new defenses.

1. **SYN flood** (Figure 8): We track the number of open TCP sessions for each source IP; if a source IP has no asymmetry between SYNs and ACKs, then mark its packets as benign. If a source IP never completes a connection, then we can mark its future packets as known attack packets. If we see a gray area where the source IP has completed some connections but not others, in which case we use a SYN-Proxy defense (e.g., [9, 28]).
2. **DNS amplification** (Figure 9): We check if the DNS server has been queried by some customer IP. This example highlights another advantage—we can decouple fast (e.g., the header-based *A_LIGHTCHECK* module) and slow path analyses (e.g., the second A module needs to look into payloads). The responses are quite simple and implement logging, dropping, or basic forwarding to the destination. We do not show the code for brevity.
3. **UDP flood:** The analysis node *A_UDP* identifies source IPs that send an anomalously higher number of UDP packets and uses this to categorize each packet as either attack or benign. The function *forward* will direct the packet to the next node in the defense strategy; i.e., *R_OK* if benign, or *R_LOG* if attack.
4. **Elephant flow:** Here, the attacker launches legitimate but very large flows. The A module detects abnormally large flows and flags them as attack flows. The response is to randomly drop packets from these large flows (not shown).



```
A_SYNFLOOD
//DiffCount tracks diff b/w syn and ack
//aggregated over some window
srcIP = pkt.srcIP
if (TCP pkt with SYN set and ACK unset)
    SynCount[srcIP]++;
if (pkt is TCP and first ACK)
    FirstAckCount[srcIP]++;
UpdateDiffCount(srcIP)
if (DiffCount(srcIP) =  SynCount[srcIP])
    markAsAttack(pkt);
else  if (DiffCount(srcIP) > 0)
    markAsUnknown(pkt);
else  if (DiffCount(srcIP) = 0 )
    markAsBenign(pkt);
forward(pkt);
```

Figure 8: SYN Flood defense strategy graph.



```
A_LIGHTCHECK
if (pkt is DNS request)
    queriedServer[pkt.dstIP] = true;
    markAsBenign(pkt);
if (pkt is DNS response)
    if (!queriedServer[pkt.srcIP])
        markAsAttack(pkt);
    else
        markAsUnknown(pkt);
    forward(pkt);
```

```
A_MATCHRQST
if (pkt is DNS request)
    requestHistory[pkt] = true;
    markAsBenign(pkt);
if (packet is DNS response)
    if (!matchRequest(pkt))
        markAsAttack(pkt);
    else
        markAsBenign(pkt);
    forward(pkt)
```

Figure 9: DNS amplification defense strategy graph.

Attack detection: We use simple time series anomaly detection using `nfdump`, a tool that provides NetFlow-like capabilities, and custom code [27]. The output of the detection module is sent to the Bohatei global controller as a 3-tuple $\langle Type, FlowSpec, Volume \rangle$, where *Type* indicates the type of DDoS attack (e.g., SYN flood, DNS amplification), *FlowSpec* provides a generic description of the *flow space* of suspicious traffic (involving wildcards), and *Volume* indicates the suspicious traffic volume based on the flow records. Note that this *FlowSpec* does not pinpoint specific attack flows; rather, it is a coarse-grained hint on characteristics of suspicious traffic that need further processing through the defense graphs.

7.2 SDN/NFV platform

Control plane: We use the `OpenDayLight` network control platform, as it has gained significant traction from key industry players [17]. We implemented the Bohatei global and local control plane modules (i.e., strategy, resource management, and network orchestration) as separate `OpenDayLight` plugins. Bohatei uses OpenFlow [40] for configuring switches; this is purely for ease of prototyping, and it is easy to integrate other network control APIs (e.g., YANG/NetCONF).

Data plane: Each physical node is realized using a VM running on KVM. We use open source tools (e.g., Snort, Bro) to implement the different Analysis (A) and

Attack type	Analysis	Response
UDP flood	A_UDP using Snort (inline mode)	R_LOG using iptables and R_RATELIMIT using tc library
DNS amp.	both LIGHTCHECK and MATCHRQST using netfilter library, iptables, custom code	R_LOG and R_DROP using iptables
SYN flood	A_SYNFLOOD using Bro	R_SYNPROXY using PF firewall, R_LOG and R_DROP using iptables
Elephant flow	A_ELEPHANT using netfilter library, iptables, custom code	R_DROP using iptables

Table 1: Implementation of Bohatei modules.

Topology	#Nodes	Run time (secs)		Optimality Gap
		Baseline	Bohatei	
Heanet	6	205	0.002	0.0003
OTEGlobe	92	2234	0.007	0.0004
Cogent	196	> 1 hr	0.01	0.0005

Table 2: Run time and optimality gap of Bohatei vs. ILP formulation across different topologies.

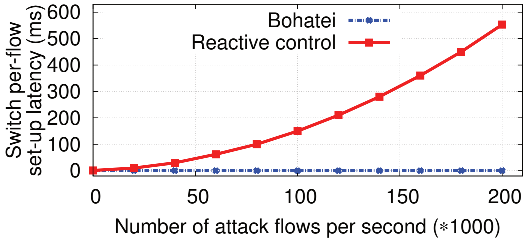


Figure 10: Bohatei control plane scalability.

Response (R) modules. Table 1 summarizes the specific platforms we have used. These tools are instrumented using FlowTags [31] to add tags to outgoing packets to provide contextual information. We used OpenvSwitch [16] to emulate switches in both datacenters and ISP backbone. The choice of OpenvSwitch is for ease of prototyping on our testbed.

Resource management algorithms: We implement the DSP and SSP algorithms using custom `Go` code.

8 Evaluation

In this section, we show that:

1. Bohatei is scalable and handles attacks of hundreds of Gbps in large ISPs and that our design decisions are crucial for its scale and responsiveness (§8.1)
2. Bohatei enables a rapid ($\leq$ 1 minute) response for several canonical DDoS attack scenarios (§8.2)
3. Bohatei can successfully cope with several dynamic attack strategies (§8.3)

Setup and methodology: We use a combination of real testbed and trace-driven evaluations to demonstrate the above benefits. Here we briefly describe our testbed, topologies, and attack configurations:

- *SDN Testbed:* Our testbed has 13 Dell R720 machines (20-core 2.8 GHz Xeon CPUs, 128GB RAM). Each machine runs KVM on CentOS 6.5 (Linux kernel v2.6.32). On each machine, we assigned equal amount of resources to each VM: 1 vCPU (virtual CPU) and 512MB of memory.
- *Network topologies:* We emulate several router-level ISP topologies (6–196 nodes) from the Internet Topology Zoo [22]. We set the bandwidth of each core link to be 100Gbps and link latency to be 10ms. The number of datacenters, which are located randomly, is 5% of the number of backbone switches with a capacity of 4,000 VMs per datacenter.
- *Benign traffic demands:* We assume a gravity model of traffic demands between ingress-egress switch pairs [50]. The total volume is scaled linearly with the size of the network such that the average link load on the topology backbone is 24Gbps with a maximum bottleneck link load of 55Gbps. We use `iperf` and custom code to generate benign traffic.
- *Attack traffic:* We implemented custom modules to generate attack traffic: (1) **SYN flood** attack by sending only SYN packets with spoofed IP addresses at a high rate; (2) **DNS amplification** using `OpenDNS` server with `BIND` (version 9.8) and emulating an attacker sending DNS requests with spoofed source IPs; (3) We use `iperf` to create some fixed bandwidth traffic to generate **elephant flows**, and (4) **UDP flood** attacks. We randomly pick one edge PoP as the target and vary the target across runs. We ramp up the attack volume until it induces maximum reduction in throughput of benign flows to the target. On our testbed, we can ramp up the volume up to 10 Gbps. For larger attacks, we use simulations.

8.1 Bohatei scalability

Resource management: Table 2 compares the run time and optimality of the ILP-based algorithm and Bohatei (i.e., DSP and SSP) for 3 ISP topologies of various sizes. (We have results for several other topologies but do not show it for brevity.) The ILP approach takes from several tens of minutes to hours, whereas Bohatei takes only a few milliseconds enabling rapid response to changing traffic patterns. The optimality gap is $\leq$ 0.04%.

Control plane responsiveness: Figure 10 shows the per-flow setup latency comparing Bohatei to the SDN per-flow and reactive paradigm as the number of attack flows in a DNS amplification attack increases. (The results are consistent for other types of attacks and are not shown for brevity.) In both cases, we have a dedicated machine for the controller with 8 2.8GHz cores and 64

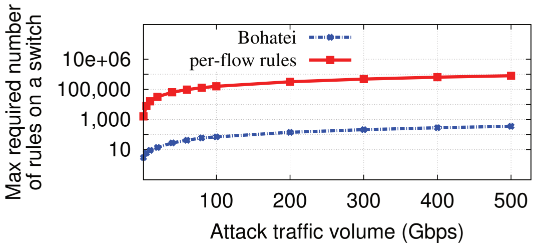


Figure 11: Number of switch forwarding rules in Bohatei vs. today's flow-based forwarding.

GB RAM. To put the number of flows in context, 200K flows roughly corresponds to a 1 Gbps attack. Note that a typical upper bound for switch flow set-up time is on the order of a few milliseconds [59]. We see that Bohatei incurs zero rule setup latency, while the reactive approach deteriorates rapidly as the attack volume increases.

Number of forwarding rules: Figure 11 shows the maximum number of rules required on a switch across different topologies for the SYN flood attack. Using today's flow-based forwarding, each new flow will require a rule. Using tag-based forwarding, the number of rules depends on the number of VM instances, which reduces the switch rule space by four orders of magnitude. For other attack types, we observed consistent results (not shown). To put this in context, the typical capacity of an SDN switch is 3K-4K rules (shared across various network management tasks). This means that per-flow rules will not suffice for attacks beyond 10Gbps. In contrast, Bohatei can handle hundreds of Gbps of attack traffic; e.g., a 1 Tbps attack will require $<$ 1K rules on a switch.

Benefit of scale-out load balancing: We measured the resources that would be consumed by a dedicated load balancing solution. Across different types of attacks with a fixed rate of 10Gbps, we observed that a dedicated load balancer design requires between 220–300 VMs for load balancing alone. By delegating the load balancing task to the VMs, our design obviates the need for these extra load balancers (not shown).

8.2 Bohatei end-to-end effectiveness

We evaluated the effectiveness of Bohatei under four different types of DDoS attacks. We launch the attack traffic of the corresponding type at 10th second; the attack is sustained for the duration of the experiment. In each scenario, we choose the attack volume such that it is capable of bringing the throughput of the benign traffic to zero. Figure 12 shows the impact of attack traffic on the throughput of benign traffic. The Y axis for each scenario shows the network-wide throughput for TCP traffic (a total of 10Gbps if there is no attack). The results shown in this figure are based on Cogent, the largest topology with 196 switches; the results for other topologies were consistent and are not shown. While we do see some small differences across attacks, the overall reaction time is short.

Attack type	# VMs needed	
	Monolithic	Fine-grained scaling
DNS Amplification	5,422	1,005
SYN Flood	3,167	856
Elephant flows	1,948	910
UDP flood	3,642	1,253

Table 3: Total hardware provisioning cost needed to handle a 100 Gbps attack for different attacks.

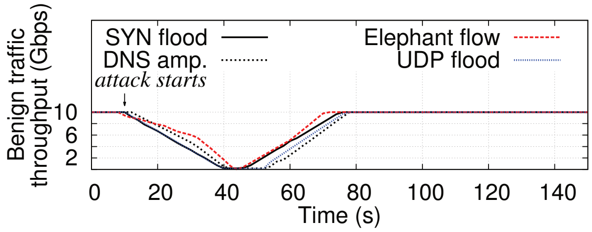


Figure 12: Bohatei enables rapid response and restores throughput of legitimate traffic.

The key takeaway is that Bohatei can help networks respond rapidly (within one minute) to diverse attacks and restore the performance of legitimate flows. We repeated the experiments with UDP as the benign traffic. In this case, the recovery time was even shorter, as the throughput does not suffer from the congestion control mechanism of TCP.

Hardware cost: We measure the total number of VMs needed to handle a given attack volume and compare two cases: (1) monolithic VMs embedding the entire defense logic for an attack, and (2) using Bohatei's fine-grained modular scaling. Table 3 shows the number of VMs required to handle different types of 100 Gbps attacks. Fine-grained scaling gives a 2.1–5.4× reduction in hardware cost vs. monolithic VMs. Assuming a commodity server costs $3,000 and can run 40VMs in Bohatei (as we did), we see that it takes a total hardware cost of less than about $32,000 to handle a 100 Gbps attack across Table 3. This is in contrast to the total server cost of about $160,000 for the same scenario if we use monolithic VMs. Moreover, since Bohatei is horizontally scalable by construction, dealing with larger attacks simply entails a linearly scale up of the number of VMs.

Routing efficiency: To quantify how Bohatei addresses the routing inefficiency of existing solutions (§2.2), we ran the following experiment. For each topology, we measured the end-to-end latency in two equivalently provisioned scenarios: (1) the location of the DDoS defense appliance is the node with the highest betweenness value[8], and (2) Bohatei. As a baseline, we consider

[8] Betweenness is a measure of a node's centrality, which is the fraction of the network's all-pairs shortest paths that pass through that node.

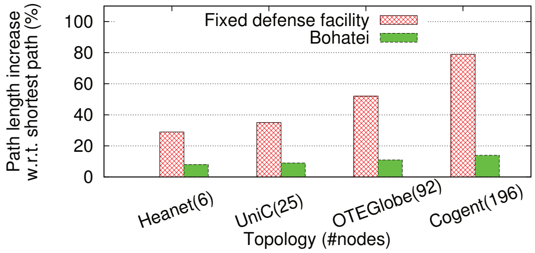
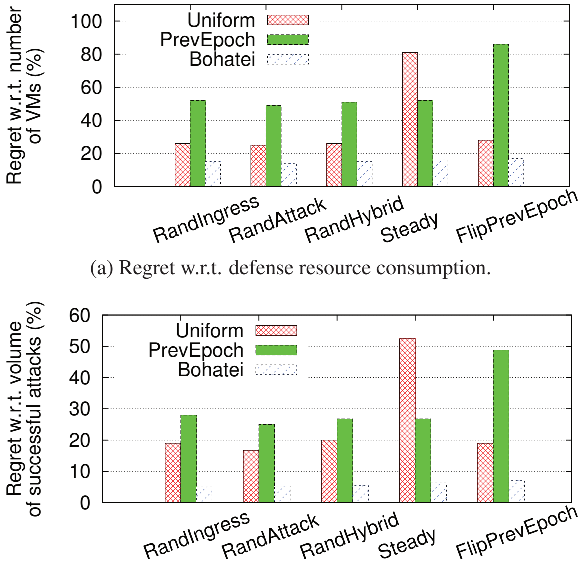


Figure 13: Routing efficiency in Bohatei.



(a) Regret w.r.t. defense resource consumption.

(b) Regret w.r.t. successful attacks.

Figure 14: Effect of different adaptation strategies (bars) vs. different attacker strategies (X axis).

shortest path routing without attacks. The main conclusion in Figure 13 is that Bohatei reduces traffic latency by 20% to 65% across different scenarios.

8.3 Dynamic DDoS attacks

We consider the following dynamic DDoS attack strategies: (1) *RandIngress*: In each epoch, pick a random subset of attack ingresses and distribute the attack budget evenly across attack types; (2) *RandAttack*: In each epoch, pick a random subset of attack types and distribute the budget evenly across all ingresses; (3) *RandHybrid*: In each epoch, pick a random subset of ingresses and attack types independently and distribute the attack budget evenly across selected pairs; (4) *Steady*: The adversary picks a random attack type and a subset of ingresses and sustains it during all epochs; and (5) *FlipPrevEpoch*: This is conceptually equivalent to conducting two *Steady* attacks $A1$ and $A2$ with each being active during odd and even epochs, respectively.

Given the typical DDoS attack duration (≈ 6 hours [12]), we consider an attack lasting for 5000 5-second epochs (i.e., ≈7 hours). Bohatei is initialized with a zero starting point of attack estimates. The metric of interest we report is the *normalized regret* with respect to the best static decision in hindsight; i.e., if we had to pick a single static strategy for the entire duration. Figure 14a and Figure 14b show the regret w.r.t. the two goals G1 (the number of VMs) and G2 (volume of successful attack) for a 24-node topology. The results are similar using other topologies and are not shown here. Overall, Bohatei's online adaptation achieves low regret across the adversarial strategies compared to two strawman solutions: (1) uniform estimates, and (2) estimates given the previous measurements.

9 Related Work

DDoS has a long history; we refer readers to surveys for a taxonomy of DDoS attacks and defenses (e.g., [41]). We have already discussed relevant SDN/NFV work in the previous sections. Here, we briefly review other related topics.

Attack detection: There are several algorithms for detecting and filtering DDoS attacks. These include time series detection techniques (e.g., [27]), use of backscatter analysis (e.g., [42]), exploiting attack-specific features (e.g., [35]), and network-wide analysis (e.g., [38]). These are orthogonal to the focus of this paper.

DDoS-resilient Internet architectures: These include the use of capabilities [58], better inter-domain routing (e.g., [60]), inter-AS collaboration (e.g., [39]), packet marking and unforgeable identifiers (e.g., [26]), and traceback (e.g., [51]). However, they do not provide an immediate deployment path or resolution for current networks. In contrast, Bohatei focuses on a more practical, single-ISP context, and is aligned with economic incentives for ISPs and their customers.

Overlay-based solutions: There are overlay-based solutions (e.g., [25,52]) that act as a "buffer zone" between attack sources and targets. The design contributions in Bohatei can be applied to these as well.

SDN/NFV-based security: There are few efforts in this space such as FRESCO [53] and AvantGuard [54]. As we saw earlier, these SDN solutions will introduce new DDoS avenues because of the per-flow and reactive model [54]. Solving this control bottleneck requires hardware modifications to SDN switches to add "stateful" components, which is unlikely to be supported by switch vendors soon [54]. In contrast, Bohatei chooses a proactive approach of setting up tag-based forwarding rules that is immune to these pitfalls.

10 Conclusions

Bohatei brings the flexibility and elasticity benefits of recent networking trends, such as SDN and NFV, to DDoS defense. We addressed practical challenges in the design of Bohatei's resource management algorithms and control/data plane mechanisms to ensure that these do not become bottlenecks for DDoS defense. We implemented a full-featured Bohatei prototype built on industry-standard SDN control platforms and commodity network appliances. Our evaluations on a real testbed show that Bohatei (1) is scalable and responds rapidly to attacks, (2) outperforms naive SDN implementations that do not address the control/data plane bottlenecks, and (3) enables resilient defenses against dynamic adversaries. Looking forward, we believe that these design principles can also be applied to other aspects of network security.

Acknowledgments

This work was supported in part by grant number N00014-13-1-0048 from the Office of Naval Research, and NSF awards 1409758, 1111699, 1440056, and 1440065. Seyed K. Fayaz was supported in part by the CMU Bertucci Fellowship. We thank Limin Jia, Min Suk Kang, the anonymous reviewers, and our shepherd Patrick Traynor for their helpful suggestions.

References

[1] Bohatei. https://github.com/ddos-defense/bohatei.

[2] Amazon EC2. http://aws.amazon.com/ec2/.

[3] Arbor Networks, worldwide infrastructure security report, volume IX, 2014. http://bit.ly/1R0NDRi.

[4] AT&T and Intel: Transforming the Network with NFV and SDN. https://www.youtube.com/watch?v=F55pHxTeJLc#t=76.

[5] AT&T Denial of Service Protection. http://soc.att.com/1IIlUec.

[6] AT&T Domain 2.0 Vision White Paper. http://soc.att.com/1kAw1Kp.

[7] Click Modular Router. http://www.read.cs.ucla.edu/click/click.

[8] CloudFlare. https://www.cloudflare.com/ddos.

[9] DDoS protection using Netfilter/iptables. http://bit.ly/1IImM2F.

[10] Dell PowerEdge Rack Servers. http://dell.to/1dtP5Jk.

[11] GSA Advantage. http://1.usa.gov/1ggEgFN.

[12] Incapsula Survey : What DDoS Attacks Really Cost Businesses, 2014. http://bit.ly/1CFZyIr.

[13] iptables. http://www.netfilter.org/projects/iptables/.

[14] NTP attacks: Welcome to the hockey stick era. http://bit.ly/1ROlwQe.

[15] ONS 2014 Keynote: John Donovan, Senior EVP, AT&T Technology & Network Operations. http://bit.ly/1RQFMko.

[16] Open vSwitch. http://openvswitch.org/.

[17] OpenDaylight project. http://www.opendaylight.org/.

[18] Packet processing on Intel architecture. http://intel.ly/1efIEu6.

[19] Prolexic. http://www.prolexic.com/.

[20] Radware. http://www.radware.com/Solutions/Security/.

[21] Time for an SDN Sequel? http://bit.ly/1BSpdma.

[22] Topology Zoo. www.topology-zoo.org.

[23] Verizon-Carrier Adoption of Software-defined Networking. https://www.youtube.com/watch?v=WVczl03edi4.

[24] ZScaler Cloud Security. http://www.zscaler.com.

[25] D. G. Andersen. Mayday: Distributed filtering for internet services. In *Proc. USITS*, 2003.

[26] D. G. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable internet protocol (AIP). In *Proc. SIGCOMM*, 2008.

[27] P. Barford, J. Kline, D. Plonka, and A. Ron. A signal analysis of network traffic anomalies. In *Proc. ACM SIGCOMM Workshop on Internet Measurement*, 2002.

[28] R. Cáceres, F. Douglis, A. Feldmann, G. Glass, and M. Rabinovich. Web proxy caching: The devil is in the details. *SIGMETRICS Perform. Eval. Rev.*, 26(3):11–15, Dec. 1998.

[29] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian. Fabric: A retrospective on evolving sdn. In *Proc. HotSDN*, 2012.

[30] J. Czyz, M. Kallitsis, M. Gharaibeh, C. Papadopoulos, M. Bailey, and M. Karir. Taming the 800 pound gorilla: The rise and decline of ntp ddos attacks. In *Proc. IMC*, 2014.

[31] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using FlowTags. In *Proc. NSDI*, 2014.

[32] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4D approach to network control and management. *ACM CCR*, 2005.

[33] V. Heorhiadi, S. K. Fayaz, M. Reiter, and V. Sekar. Frenetic: A network programming language. *Information Systems Security*, 2014.

[34] Jain et al. B4: Experience with a globally-deployed software defined wan. In *Proc. SIGCOMM*, 2013.

[35] C. Jin, H. Wang, and K. G. Shin. Hop-count filtering: An effective defense against spoofed ddos traffic. In *Proc. CCS*, 2003.

[36] A. Kalai and S. Vempala. Efficient algorithms for online decision problems. *J. Comput. Syst. Sci.*, 2005.

[37] M. S. Kang, S. B. Lee, and V. Gligor. The crossfire attack. In *Proc. IEEE Security and Privacy*, 2013.

[38] A. Lakhina, M. Crovella, and C. Diot. Mining Anomalies Using Traffic Feature Distributions. In *Proc. SIGCOMM*, 2005.

[39] R. Mahajan et al. Controlling high bandwidth aggregates in the network. *CCR*, 2001.

[40] N. McKeown et al. OpenFlow: enabling innovation in campus networks. *CCR*, March 2008.

[41] J. Mirkovic and P. Reiher. A taxonomy of ddos attack and ddos defense mechanisms. In *CCR*, 2004.

[42] D. Moore, C. Shannon, D. J. Brown, G. M. Voelker, and S. Savage. Inferring internet denial-of-service activity. *ACM Trans. Comput. Syst.*, 2006.

[43] Network functions virtualisation – introductory white paper. http://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[44] A. Networks. ATLAS Summary Report: Global Denial of Service. http://atlas.arbor.net/summary/dos.

[45] P. Patel et al. Ananta: cloud scale load balancing. In *Proc. ACM SIGCOMM*, 2013.

[46] V. Paxson. Bro: A system for detecting network intruders in real-time. In *Computer Networks*, 1999.

[47] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *Proc. OSDI*, 2014.

[48] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *LISA*, 1999.

[49] C. Rossow. Amplification hell: Revisiting network protocols for ddos abuse. In *Proc. USENIX Security*, 2014.

[50] M. Roughan. Simplifying the Synthesis of Internet Traffic Matrices. *ACM SIGCOMM CCR*, 2005.

[51] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical network support for ip traceback. In *Proc. SIGCOMM*, 2000.

[52] E. Shi, I. Stoica, D. Andersen, and A. Perrig. OverDoSe: A generic DDoS protection service using an overlay network. Technical Report CMU-CS-06-114, School of Computer Science, Carnegie Mellon University, 2006.

[53] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson. FRESCO: Modular composable security services for software-defined networks. In *Proc. NDSS*, 2013.

[54] S. Shin, V. Yegneswaran, P. Porras, and G. Gu. AVANT-GUARD: Scalable and vigilant switch flow management in software-defined networks. In *Proc. CCS*, 2013.

[55] A. Studer and A. Perrig. The coremelt attack. In *Proc. ESORICS*, 2009.

[56] T. Koponen et al. Network virtualization in multi-tenant datacenters. In *Proc. NSDI*, 2014.

[57] P. Verkaik, D. Pei, T. Schollf, A. Shaikh, A. C. Snoeren, and J. E. van der Merwe. Wresting Control from BGP: Scalable Fine-grained Route Control. In *Proc. USENIX ATC*, 2007.

[58] X. Yang, D. Wetherall, and T. Anderson. A dos-limiting network architecture. In *Proc. SIGCOMM*, 2005.

[59] S. Yeganeh, A. Tootoonchian, and Y. Ganjali. On scalability of software-defined networking. *Communications Magazine, IEEE*, 2013.

[60] X. Zhang, H.-C. Hsiao, G. Hasker, H. Chan, A. Perrig, and D. G. Andersen. Scion: Scalability, control, and isolation on next-generation networks. In *Proc. IEEE Security and Privacy*, 2011.

A ILP Formulation

The ILP formulation for an optimal resource management (mentioned in §4.2) is shown in Figure 15.

Vairables: In addition to the parameters and variables that we have defined earlier in §4, we define the binary variable $q_{d,a,i,vm,s,i',vm',s',l}$ as follows: if it is 1, VM *vm* of

$$\text{Minimize } \alpha \times \sum_e \sum_a \sum_d f_{e,a,d} \times T_{e,a} \times L_{e,d} + \sum_d dsc_d \tag{1}$$

s.t.

$$\forall e,a : \sum_d f_{e,a,d} = 1 \triangleright \text{ all suspicious traffic should be served} \tag{2}$$

$$\forall a,d : t_{a,d} = \sum_e f_{e,a,d} \times T_{e,a} \triangleright \text{ traffic of each type to each datacenter} \tag{3}$$

$$\forall d : \sum_a t_{a,d} \leq C_d^{link} \triangleright \text{ datacenter link capacity} \tag{4}$$

$$\forall d,a,i : \sum_{s \in S_d} n_{a,i}^{d,s} \geq t_{a,d} \times \frac{\sum_{i':(i',i)=e_{a,i' \to i}^{annotated}} W_{a,i' \to i}}{P_{a,i}} \triangleright \text{ provisioning sufficient VMs (} S_d \text{ is the set of } d\text{'s servers.)} \tag{5}$$

$$\forall d,s \in S_d : \sum_a \sum_i n_{a,i}^{d,s} \leq C_{d,s}^{compute} \triangleright \text{ server compute capacity} \tag{6}$$

$$\forall d : dsc_d = intraR_d \times IntraUnitCost + interR_d \times InterUnitCost \triangleright \text{ total cost within each datacenter} \tag{7}$$

$$\forall d : intraR_d = \sum_a \sum_{(i,i')=e_{a,i \to i'}^{annotated}} \sum_{(s,s') \in sameRack} \sum_{vm=1}^{MaxVM} \sum_{vm'=1}^{MaxVM} \sum_{l=1}^{MaxVol} q_{d,a,i,vm,s,i',vm',s',l} \triangleright \text{ intra-rack cost} \tag{8}$$

$$\forall d : interR_d = \sum_a \sum_{(i,i')=e_{a,i \to i'}^{annotated}} \sum_{(s,s') \notin sameRack} \sum_{vm=1}^{MaxVM} \sum_{vm'=1}^{MaxVM} \sum_{l=1}^{MaxVol} q_{d,a,i,vm,s,i',vm',s',l} \triangleright \text{ inter-rack cost} \tag{9}$$

$$\forall d,a,i',vm' : \sum_s \sum_{s'} \sum_{i:(i,i')=e_{a,i \to i'}^{annotated}} \sum_{vm=1}^{MaxVM} \sum_{l=1}^{MaxVol} q_{d,a,i,vm,s,i',vm',s',l} \leq P_{a,i'} \triangleright \text{ enforcing VMs capacities} \tag{10}$$

$$\forall d,s \in S_d,a,i' : n_{a,i'}^{d,s} \times P_{a,i'} \geq \sum_{vm=1}^{MaxVM} \sum_{vm'=1}^{MaxVM} \sum_{i:(i,i')=e_{a,i \to i'}^{annotated}} \sum_{s'} \sum_{l=1}^{MaxVol} q_{d,a,i,vm,s,i',vm',s',l} \triangleright \text{ bound traffic volumes} \tag{11}$$

$$\forall d,s \in S_d,a,i' : n_{a,i'}^{d,s} \times P_{a,i'} \leq \sum_{vm=1}^{MaxVM} \sum_{vm'=1}^{MaxVM} \sum_{i:(i,i')=e_{a,i \to i'}^{annotated}} \sum_{s'} \sum_{l=1}^{MaxVol} q_{d,a,i,vm,s,i',vm',s',l} + 1 \triangleright \text{ bound traffic volumes} \tag{12}$$

$$\triangleright \text{ flow conservation for VM } vm \text{ of type logical node } k \text{ that has both predecessor(s) and successor(s)}$$
$$\forall d,a,k,vm : \sum_{vm'=1}^{MaxVM} \sum_{g:(g,k)=e_{a,g \to k}^{annotated}} \sum_s \sum_{s'} \sum_{l=1}^{MaxVol} q_{d,a,g,vm',s',k,vm,s,l} = \sum_{vm'=1}^{MaxVM} \sum_{h:(k,h)=e_{a,k \to h}^{annotated}} \sum_s \sum_{s'} \sum_{l=1}^{MaxVol} q_{d,a,k,vm,s,h,vm',s',l} \tag{13}$$

$$\forall link \in ISP\ backbone : \sum_{link \in Path_{e \to d}} \sum_a f_{e,a,d} \times T_{e,a} \leq \beta \times MaxLinkCapacity \triangleright \text{ per-link traffic load control} \tag{14}$$

$$f_{e,a,d} \in [0,1], q_{d,a,i,vm,s,i',vm',s',l} \in \{0,1\}, n_{a,i}^d, n_{a,i}^{d,s} \in \{0,1,\ldots\}, t_{a,d}, interR_d, intraR_d, dsc_d \in \mathbb{R} \triangleright \text{ variables} \tag{15}$$

Figure 15: ILP formulation for an optimal resource management.

type $v_{a,i}$ runs on server s and sends 1 unit of traffic (e.g., 1 Gbps) to VM vm' of type $v_{a,i'}$ that runs on server s', where $e_{a,i \to i'}^{annotated} \in E_a^{annotated}$, and servers s and s' are located in datacenter d; otherwise, $q_{d,a,i,vm,s,i',vm',s',l} = 0$. Here l is an auxiliary subscript indicating that the one unit of traffic associated with q is the lth one out of $MaxVol$ possible units of traffic. The maximum required number of VMs of any type is denoted by $MaxVM$.

The ILP involves two key decision variables: *(1)* $f_{e,a,d}$ is the fraction of traffic $T_{e,a}$ to send to datacenter D_d, and *(2)* $n_{a,i}^{d,s}$ is the number of VMs of type $v_{a,i}$ on server s of datacenter d, hence physical graphs $DAG_a^{physical}$.

Objective function: The objective function (1) is composed of inter-datacenter and intra-datacenter costs, where constant $\alpha > 0$ reflects the relative importance of inter-datacenter cost to intra datacenter cost.

Constraints: Equation (2) ensures all suspicious traffic will be sent to data centers for processing. Equation (3) computes the amount of traffic of each attack type going to each datacenter, which is ensured to be within datacenters bandwidth capacity using (4). Equation (5) is intended to ensure sufficient numbers of VMs of the required types in each datacenter. Servers compute capacities are enforced using (6). Equation (7) sums up the cost associated with each datacenter, which is composed of two components: intra-rack cost, given by (8), and inter-rack component, given by (9). Equation (10) ensures the traffic processing capacity of each VM is not exceeded. Equations (11) and (12) tie the variables for number of VMs (i.e., $n_{a,i}^{d,s}$) and traffic (i.e., $q_{d,a,i,vm,s,i',vm',s',l}$) to each other. Flow conservation of nodes is guaranteed by (13). Inequality (14) ensures no ISP backbone link gets congested (i.e., by getting a traffic volume of more than a fixed fraction β of its maximum capacity), while $Path_{e \to d}$ is a path from a precomputed set of paths from e to d. The ILP decision variables are shown in (15).

B DSP and SSP Algorithms

As described in §4.3, due to the impractically long time needed to solve the ILP formulation, we design the DSP and SSP heuristics for resource management. The ISP global controller solves the DSP problem to assign suspicious incoming traffic to data centers. Then each local controller solves an SSP problem to assign servers to

VMs. Figure 16 and 17 show the detailed pseudocode for the DSP and SSP heuristics, respectively.

```
▷ Inputs: $L$, $T$, $DAG_a^{annotated}$, $C_d^{link}$, and $C_d^{compute}$
▷ Outputs: $DAG_{a,d}^{physical}$ and $f_{e,a,d}$ values

Build max-heap $T^{maxHeap}$ of attack volumes $T$
while $!Empty(T^{maxHeap})$
      do $t \leftarrow ExtractMax(T^{maxHeap})$
          $d \leftarrow$ datacenter with min. $L_{t.e,t.d}$ and cap.$> 0$
          ▷ enforcing datacenter link capacity
          $t_1 \leftarrow min(t, C_d^{link})$
          ▷ compute capacity of $d$ for traffic type $a$
          $t_2 \leftarrow \frac{C_d^{Compute}}{\sum_i \frac{\sum_{i'} W_{a,i' \rightarrow i}}{P_{a,i}}}$
          ▷ enforcing datacenter compute capacity
          $t_{assigned} \leftarrow min(t_1, t_2)$
          $f_{e,a,d} \leftarrow \frac{t_{assigned}}{T_{t.e,t.a}}$
          for each module type $i$
               do ▷ update $n_{a,i}^d$ given new assignment
                    $n_{a,i}^d = n_{a,i}^d + t_{assigned}^d \frac{\sum_{i'} W_{a,i' \rightarrow i}}{P_{a,i}}$
          $C_d^{link} \leftarrow C_d^{link} - t_{assigned}$
          $C_d^{compute} \leftarrow C_d^{compute} - t_{assigned} \sum_i \frac{\sum_{i'} W_{a,i' \rightarrow i}}{P_{a,i}}$
          ▷ leftover traffic
          $t_{unassigned} = t - t_{assigned}$
          if ($t_{unassigned} > 0$)
             then $Insert(T^{maxHeap}, t_{unassigned})$
for each datacenter $d$ and attack type $a$
      do Given $n_{a,i}^d$ and $DAG_a^{annotated}$, compute $DAG_{a,d}^{physical}$
```

Figure 16: Heuristic for datacenter selection problem (DSP).

```
▷ Inputs: $DAG_{a,d}^{physical}$, $IntraUnitCost$, $InterUnitCost$,
and $C_{d,s}^{compute}$ values
▷ Outputs: $n_{a,i}^{d,s}$ values

while entire $DAG_{a,d}^{physical}$ is not assigned to $d$'s servers
      do $N \leftarrow v_{a,i}^{annotated}$ whose all predecessors are assigned
          if ($N == NIL$)
             then $N \leftarrow v_a^{annotated}$ with max $P_{a,i}$
          $localize$(nodes of $DAG_{a,d}^{physical}$ corresponding to $N$)

▷ function $localize$ tries to assign all of its
input physical nodes to the same server or rack
$localize$(inNodes){
assign all inNodes to emptiest server
if failed
   then assign all inNodes to emptiest rack
          if failed
             then split inNodes $V_a^{physical}$ across racks
update $n_{a,i}^{d,s}$ values
}
```

Figure 17: Heuristic for server selection problem (SSP) at datacenter d.

Boxed Out: Blocking Cellular Interconnect Bypass Fraud at the Network Edge

Bradley Reaves
University of Florida
reaves@ufl.edu

Ethan Shernan
Georgia Institute of Technology
eshernan3@mail.gatech.edu

Adam Bates
University of Florida
adammbates@ufl.edu

Henry Carter
Georgia Institute of Technology
carterh@gatech.edu

Patrick Traynor
University of Florida
traynor@cise.ufl.edu

Abstract

The high price of incoming international calls is a common method of subsidizing telephony infrastructure in the developing world. Accordingly, international telephone system interconnects are regulated to ensure call quality and accurate billing. High call tariffs create a strong incentive to evade such interconnects and deliver costly international calls illicitly. Specifically, adversaries use VoIP-GSM gateways informally known as "simboxes" to receive incoming calls over wired data connections and deliver them into a cellular voice network through a local call that appears to originate from a customer's phone. This practice is not only extremely profitable for simboxers, but also dramatically degrades network experience for legitimate customers, violates telecommunications laws in many countries, and results in significant revenue loss. In this paper, we present a passive detection technique for combating simboxes at a cellular base station. Our system relies on the raw voice data received by the tower during a call to distinguish errors in GSM transmission from the distinct audio artifacts caused by delivering the call over a VoIP link. Our experiments demonstrate that this approach is highly effective, and can detect 87% of real simbox calls in only 30 seconds of audio with no false positives. Moreover, we demonstrate that evading our detection across multiple calls is only possible with a small probability. In so doing, we demonstrate that fraud that degrades network quality and costs telecommunications billions of dollars annually can easily be detected and counteracted in real time.

1 Introduction

Cellular networks provide digital communications for more than five billion people around the globe. As such, they represent one of the largest, most integral pieces of critical infrastructure in the modern world. Deploying these networks requires billions of dollars in capital by providers, and often necessitates government subsidies in poorer nations where such investments may not produce returns for many decades. As a means of maintaining these systems, international calls destined for such networks are often charged a significant tariff, which distributes the costs of critical but expensive cellular infrastructure to callers from around the world.

Many individuals seek to avoid such tariffs by any means necessary through a class of attacks known as *interconnect bypass fraud.* Specifically, by avoiding the regulated network interconnects and instead finding unintended entrances to the provider network, a caller can be connected while dramatically lowering his or her costs. Such fraud constitutes a "free rider" problem, a term from economics in which some participants enjoy the benefits of expensive infrastructure without paying to support it. The most common implementation of interconnect bypass fraud is known as simboxing. Enabled by VoIP GSM gateways (i.e., "simboxes"), simboxing connects incoming VoIP calls to local cellular voice network via a collection of SIM cards and cellular radios. Such calls appear to originate from a customer phone to the network provider and are delivered at the subsidized domestic rate, free of international call tariffs. Interconnection bypass fraud negatively impacts availability, reliability and quality for legitimate consumers by creating network hotspots through the injection of huge volumes of tunneled calls into underprovisioned cells, and costs operators over $2 Billion annually [28].

In this paper, we present Ammit[1], a system for detecting simboxing designed to be deployed in a cellular network. Our solution relies on the fact that audio transmitted over the Internet before being delivered to the GSM network will be degraded in measurable, distinctive ways. We develop novel techniques and build

[1] Ammit was an Egyptian funerary deity who was believed to separate pure and impure souls, preventing the latter from achieving immortality in the afterlife.

on mechanisms from the Pindr0p call fingerprinting system [25] to measure these degradations by applying a number of light-weight signal processing methods to the received call audio and examining the results for distinguishing characteristics. These techniques rapidly and automatically identify simboxed calls and the SIMs used to make such connections, thereby allowing us to quickly shut down these rogue accounts. In so doing, our approach makes these attacks far less likely to be successful and stable, thereby largely closing these illegal entrances to provider networks.

We make the following contributions:

- **Identify audio characteristics useful for detecting simboxes:** We identify features in simboxed call audio that make it easily differentiable from traditional GSM cellular calls and argue why such features are difficult for adversaries to avoid.

- **Develop rapid detection architecture for the network edge:** We design and implement Ammit, a detection tool that uses signal processing techniques for identifying illicitly tunneled VoIP audio in a GSM network, and demonstrate that our techniques can easily execute in real time. Such performance means that our solution can be practically deployed at the cellular network edge.

- **Demonstrate high detection rate for SIM cards used in simboxes:** Through experimental analysis on a real simbox, we show that Ammit can quickly profile and terminate *87% of simboxed calls with no false positives*. Such a high detection rate arguably makes interconnect bypass fraud uneconomical.

We note that our techniques differ significantly from related work, which requires either large-scale post hoc analysis [42] or serendipitous test calls to network probes [10, 13, 15, 16, 18]. Our approach is intended to be used in real time, allowing for rapid detection and elimination of simboxes. We specifically characterize these techniques in Section 8.

It should be noted that the authors are not attempting to combat the spread of inexpensive VoIP calls in this paper. Traditional VoIP calls, which connect users through IP or a licensed VoIP-PSTN (Public Switched Telephone Network) gateway, are not considered a problem in countries that combat simboxes. Instead, we seek to prevent the creation of unauthorized entry points into private cellular networks that degrade performance for legitimate users and cost providers and governments two billion dollars annually. This is analogous to the problem of rogue Wi-Fi access points; simboxing prevents network administrators from controlling access to the network and can degrade service for other users. Moreover, similar to other economic free rider problems, failure to combat such behavior can lead to both underprovisioning and the overuse of such networks, making quality and stability difficult to achieve [49]. *Failure to combat simbox fraud may ultimately lead to raising prices and lower reliability for subsidized domestic calls in developing nations, where the majority of citizens can rarely afford such cost increases.*

The remainder of this paper is organized as follows: Section 2 provides background information on cellular networks; Section 3 describes simbox operation and their consequences; Section 4 presents our detection methodology; Section 6 describes our experimental methodology; Section 7 discusses our results; Section 8 offers an overview of important related work; and Section 9 presents our final remarks.

2 Background

2.1 Cellular Networks

The Global System for Mobile Communications (GSM) is a suite of standards used to implement cellular communications. It is used by the majority of carriers in the US and throughout Europe, Africa, and Asia. GSM is a "second generation" (2G) cellular network and has evolved into UMTS (3G) and LTE (4G) standards. We focus on GSM because it is the most available for direct experimentation. Note that the methods we present in the paper can easily be ported to other cellular standards.

GSM manages user access to the network by issuing users a small smartcard called a Subscriber Identity Module (SIM card) that contains identity and cryptographic materials. A carrier SIM card can be placed in any device authorized to operate on a carrier's network. Because GSM networks cryptographically authenticate almost every network transaction, cellular network activity can always be attributed to a specific SIM card. In the past, the ability to clone a SIM card negated this guarantee; however, modern SIM cards now have hardware protections that prevent practical key recovery and card cloning.

In addition to describing network functionality, the GSM standards also specify a method for encoding audio known as the GSM Full Rate (GSM-FR) codec [23]. Although designed for mobile networks, it is also used as a general purpose audio codec and is frequently implemented in VoIP software. To avoid ambiguity, we use "GSM" or "air transmission" to mean GSM cellular networks and "GSM-FR" to indicate the audio codec.

2.2 VoIP

Voice over Internet Protocol (VoIP) is a technology that implements telephony over IP networks such as the Internet. Two clients can complete a VoIP call using exclusively the Internet, or calls may also be routed from a VoIP client to a PSTN line (or vice-versa) through a VoIP Gateway. Providers including Vonage, Skype, and Google Voice provide both IP-only and IP-PSTN calls. The majority of VoIP calls are set up using a text-based protocol called the Session Initiation Protocol (SIP). One of the jobs of SIP is to establish which audio codec will be used for the call. Once a call has been established, audio flows between callers using the Realtime Transport Protocol (RTP), which is typically carried over UDP.

VoIP call quality is affected by packet loss and jitter. Absent packets, whether they are the result of actual loss or jitter, cause gaps in audio. Such gaps are filled in with silence by default. Some VoIP clients attempt to improve over this standard behavior and implement Packet Loss Concealment (PLC) algorithms to fill in missing packets with repeated or generated audio. Specifically, PLC algorithms take advantage of the fact that speech waveforms are more or less stationary for short time periods, so clients can generate a plausible section of audio from previous packets. Many codecs have mandatory PLCs, although some are optional (as in the case of the G.711 audio codec) or are not implemented (as is frequently the case when GSM-FR is used outside of cellular networks). Some VoIP software (including Asterisk) implements their own PLC algorithms, but do not activate them unless configured by an administrator.

3 What is a Simbox?

A simbox is a device that connects VoIP calls to a GSM voice (*not data*) network. A simple mental model for a simbox is a VoIP client whose audio inputs and outputs are connected to a mobile phone. The term "simbox" derives from the fact that the device requires one or more SIM cards to wirelessly connect to a GSM network.

There is a strong legitimate market for these devices in private enterprise telephone networks. GSM-VoIP gateways are sold to enterprises to allow them to use a cellular calling plan to terminate[2] calls originating in an office VoIP network to mobile devices. This is typically a cost saving measure because the cost of maintaining a mobile calling plan is often lower that the cost of paying termination fees to deliver the VoIP call through a VoIP PSTN provider (as well as the cost to the receiving party). *Such a setup is done with the permission of a licensed telecommunications provider and is only done for domestic calls.* This is in direct opposition to simboxers, who purchase subsidized SIM cards to deliver traffic onto a local network without paying the legally mandated tariffs.

Because there is a high demand for GSM-VoIP gateways, they span a wide range of features and number of concurrent calls supported. Some gateways support limited functionality and only a single SIM card, while others hold hundreds of cards and support many audio codecs. Some simboxes used in simbox fraud rings are actually distributed, with one device holding hundreds of cards in a "SIM server" while one or more radio interfaces connect calls using the "virtual SIM cards" from the server. This allows for simple provisioning of SIM cards, as well as the ability to rotate the cards to prevent high-use or location-based fraud detection.

3.1 How Simbox Fraud Works

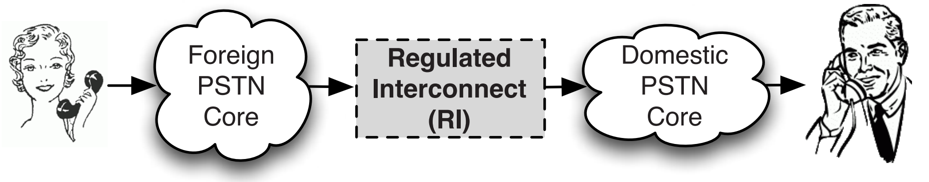
Simboxing is a lucrative attack. Because simboxers can terminate calls at local calling rates, they can significantly undercut the official rate for international calls while still making a handsome profit. In doing so, simboxers are effectively acting as an unlicensed and unregulated telecommunications carrier. Simboxers' principal costs include simbox equipment (which can represent an investment up to $200,000 US in some cases), SIM cards for local cellular networks, airtime, and an Internet connection. Successfully combating this type of fraud can be accomplished by making any of these costs prohibitively high.

Figure 1 demonstrates in greater detail how simboxing compares to typical legitimate international call termination. Figure 1 shows two international call paths: a typical path (Figure 1a) and one simbox path (Figure 1b).

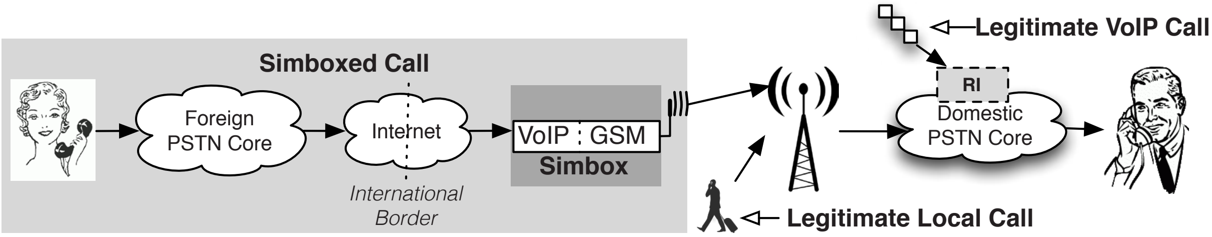
In the typical case, when Alice calls Bob, her call is routed through the telephone network in her country (labeled "Foreign PSTN Core") to an interconnect between her network and Bob's network. The call is passed through the interconnect, routed through Bob's domestic telephone network ("Domestic PSTN Core") to Bob's phone. If Alice and Bob are not in neighboring countries, there may be several interconnects and intermediate networks between Alice and Bob. The process essentially remains the same if Alice or Bob are using mobile phones. The interconnect in this scenario is crucial — interconnects are heavily regulated and monitored to ensure both call quality and billing accuracy (especially for tariffs).

In the simbox case, Alice's call is routed through her domestic telephone network, but rather than passing through a regulated interconnect, her call is routed over IP to a simbox in the destination country. The simbox then places a separate call on the cellular network in the

[2] In cellular and telephone networks, "terminating a call" has the counterintuitive meaning of "establishing a complete circuit from the caller to the callee."



(a) A typical international call is routed through a regulated interconnect. Note that VoIP calls from services such as Skype that terminate on a mobile phone also pass through this regulated interconnect and are not the target of this research.



(b) A simboxed international call (gray box) avoids the regulated interconnect by routing the call to a simbox that completes the call using the local cellular network.

Figure 1: Typical and Simboxed Calls

destination country, then routes the audio from the IP call into the cellular call, which is routed to Bob through the domestic telephone network.

In practice, simboxers execute this attack and profit in one of two ways. The most common method is for the simboxer to present themselves as a legitimate telecommunications company that offers call termination as a service to other telecom companies. As a call is routed through these intermediate networks, neither of the end users is aware that the call is being routed through a simbox. This agreement is analogous to a contract between two ISPs who have agreed to route traffic between their networks. While the end user has no knowledge of how his traffic is routed, the intermediate network owners profit from reduced prices for routed traffic.

The second method simboxers use to profit is to offer discounted call rates directly to end consumers, primarily through the sale of international calling cards. Such cards have a number that the user must dial before she can dial the recipient's number; this number will route to a number provided by a VoIP provider that points to the simbox in the recipient's country. When the user calls the number on her calling card, the simbox will answer, prompt her to dial the recipient's number, then the simbox will connect the call.

3.2 Consequences of Simbox Operation

The consequences of simboxing are significant to users who place simbox calls, users who share the cellular network with simboxers, and to cellular carriers and national governments.

As for the effects on users, Alice is likely unaware of the details of her call routing. However, Alice and Bob may both notice a degradation in quality, and Bob may notice that the Caller ID for Alice does not show her correct number. Bob may blame his local carrier for poor call quality, and so the carrier unfairly suffers in reputation.

Other users in the same cell as the simbox also suffer negative consequences. Cellular networks are provisioned to meet an expected demand of *mobile* users who only use the network a fraction of the time, and accordingly may only be able to support a few dozen simultaneous calls. When a simboxer sets up an unauthorized carrier and routes dozens of calls through a cell provisioned to support only a handful of simultaneous calls, the availability of that cell to service legitimate calls is significantly impaired. Connectivity within the cell may be further impaired by the dramatic increase in control traffic [50].

4 Methodology

Legitimate VoIP calls and other international calls enter a cellular network through a regulated interconnect or network border gateway. To halt simboxed calls, we only need to monitor incoming calls from devices containing a SIM card. Figure 1b shows the path of legitimate and simboxed audio, respectively, from the calling source to the final destination. In both cases, the tower believes it is servicing a voice call from a mobile phone. However, the audio received by the tower from a simboxed call will contain losses, indicating that the audio signal has traveled over an Internet connection, while the audio from a legitimate call will not contain these losses, having been recorded directly on the transmitting mobile phone. As discussed in Section 2, jitter and loss in Internet telephony manifest as unconcealed and concealed gaps of audio to the receiving client (the simbox, in this case). These features are *inherent* to VoIP transmission, and the only variant is the frequency of these events. All simboxed calls will have some amount of packet loss and jitter, so we design Ammit to detect these audio degradations. Because the audio transmitted *to* the mobile device could have originated from a variety of connection types, Ammit only analyzes audio received *from* mobile devices. If the mobile device is a simbox, the characteristics of this audio will exhibit the loss patters consistent with a VoIP connection, making the call distinguishable from audio recorded and sent by a mobile phone.

4.1 Inputs to Ammit

The most common codec supported by simboxes is G.711 [3] (see the Appendix for details). The G.711 codec is computationally simple, royalty-free, and serves as a least common denominator in VoIP systems. It was originally developed in 1972 for digital trunking of audio in the PSTN, and it is still the digital encoding used in PSTN core networks. The original standard indicated that G.711 should insert silence when packets are delayed or lost, so we examine G.711 using this setting.

Simboxers will have a clear incentive to configure their simboxes to evade detection, and an obvious evasion strategy is to ensure that audio is as close as possible to legitimate audio by using the GSM-FR codec for the VoIP link. Therefore, we show how Ammit accounts for this difficult case where GSM-FR is used with and without PLC. We discuss how Ammit addresses other evasion techniques in Section 5.

In summary, Ammit must detect the two audio phenomena characteristic of VoIP transmission: concealed and unconcealed packet losses. The following subsections detail how Ammit detects these phenomena, but first we briefly describe the data that Ammit receives from the tower before detecting audio features.

In GSM, audio encoded with the GSM-FR codec is transmitted between a mobile station (MS, i.e., a phone) and a base transceiver station (BTS, i.e., a cell tower) using a dedicated traffic channel. The encoding used by GSM-FR causes certain bits in a frame to be of greater importance than others. When an audio frame is transmitted, frame bits are separated by their importance. "Class 1" bits containing the most important parameters are protected by a parity check and error correcting codes, while "Class 2" bits are transmitted with no protections because bit error in these bits has only a small effect on the quality of the audio. The approach of only protecting some bits is a compromise between audio quality and the cost of the error correcting code. When Class 1 bit errors cannot be corrected, the receiver erases (i.e., drops) the entire frame. When Class 2 bits are modified, the audio is modified, but the receiver has no mechanism to detect or correct these modifications. This is termed "bit error." It should be noted that bit error and frame erasure are distinct concerns in GSM.

The receiving device (MS or BTS) may use PLC to conceal this frame erasure. When a BTS erases a frame, it conceals the loss before forwarding the audio into the core network. Visibility into frame erasures motivates our choice to place Ammit at the tower. However, there are additional benefits to locating Ammit at a tower. Specifically, this allows for scalable detection of simboxes because a single Ammit instance is only responsible for the dozens of calls that pass through the tower instead of the thousands of concurrent calls in a region or nation. Finally, if Ammit has a high confidence that a call is simboxed (as defined by a network administrator policy), ending a call at the tower is simpler than in other parts of the network. This policy would further frustrate the efforts of simboxers. It is also possible to deploy Ammit closer to the network core, perhaps at BSC or MSC nodes, but GSM loss information would need to be forwarded.

Ammit takes two inputs: a stream of GSM-encoded audio frames and a vector indicating which audio frames were erased (both of which can be collected by the BTS connecting the call). Ammit uses the frame erasure vector to ignore the effects of the air interface on the call audio. Ignoring erased frames ensures that losses on the air interfaces are not misinterpreted as losses caused by VoIP.

4.2 Detecting Unconcealed Losses

Ammit must detect two degradation types: unconcealed packet loss and concealed packet loss. To detect unconcealed loss, Ammit looks for portions of audio where the energy of the audio drops to a minimum value then

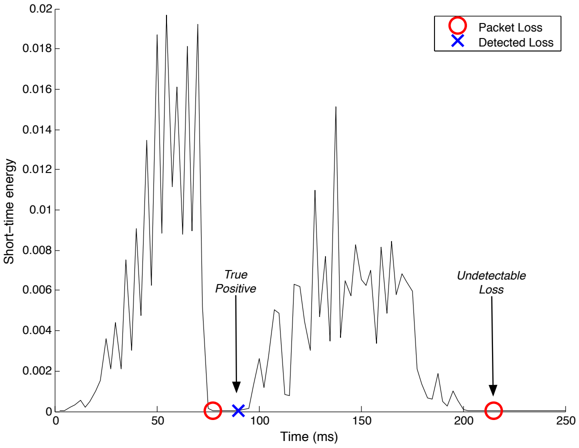


Figure 2: The short-term energy of speech during audio can reveal silence insertion. Packet loss that falls in naturally silent sections of audio is undetectable.

quickly rises again. This technique is also used in the Pindr0p system. The following discussion describes the Pindr0p approach to detecting unconcealed losses, with additional implementation insight and details.

Figure 2 demonstrates unconcealed packet loss in a clip of audio at 78 ms and 215 ms. At 78 ms, a packet is lost and silence begins. A short time later, at 90 ms, the energy rises again, indicating that a new packet has arrived containing speech. Because the time between the energy fall and rise is less than typical in speech, Ammit marks that section of audio as containing a lost packet.

While the intuition is simple, there are several challenges to using this technique to detect losses from simboxed audio. The first challenge is that many packet losses will occur during naturally silent audio — meaning that there will be no significant change in energy. This fact merely limits the amount of detectable loss events. The second challenge is that speech regularly has short pauses (causing false positives). A third challenge is that because there is no guarantee that VoIP frames are fully contained within a single GSM frame, a VoIP loss could begin in the middle of a GSM packet. Finally, uncorrected packet losses will have very low but non-zero energy because the pure silence is altered by bit errors in air transmission or by degradations within the simbox.

The first step of detecting unconcealed packet loss is to compute the energy of the audio signal. Ammit uses Short Time Energy (STE) as its measure of signal energy. Short time energy is a frequently used metric in speech analysis [38]. STE is computed by taking small windows of data and summing the squared values of the signal in the window. More formally, STE can be written as

$$E_n = \sum_{i=n-N+1}^{n} ((x(i))w(n-i))^2$$

where x is the audio signal, w is the window function, n is the frame number and N is the frame size.

Ammit computes STE using a 10ms audio frame, not the 20ms frames used in GSM-FR and many other codecs, because 10ms is the minimum frame size used by a VoIP codec, as standardized in RFC 3551 [47]. We use the standard practice of using a Hamming window half the length of the frame with a 50% overlap. Therefore, each STE measurement covers 5ms of audio and overlaps with 2.5ms of audio with the last window. This fine-grained measurement of energy ensures that Ammit can detect packet loss that begins in the middle of a GSM air frame.

With STE computed, Ammit then computes the lower envelope of the energy. In the presence of noise, the "silence" inserted in the VoIP audio will have non-zero energy. We define the lower envelope as the mean of the minimum energy found in the 10 ms frames. We also determine a tolerance around the minimum energy consisting of 50% of the lower envelope mean (this was determined experimentally).

Once Ammit has determined the lower envelope, it looks for energies that fall within the minimum envelope tolerance but then rise after a short number of energy samples. We experimentally chose 40ms as the maximum value for a sudden drop in packet energy, and our experimental results reflect the fact that this period is lower than the minimum for pauses in standard speech (which is around 50–60ms).

Because this method simply looks for silence, it is effective for both codecs we study, and it is fundamentally suited for all codecs that insert silence in the place of lost packets.

4.3 Detecting Concealed Losses in GSM-FR

Before we describe how Ammit detects GSM-FR packet loss concealment, we first describe GSM-FR PLC [24] at a high level. On the first frame erasure, the erased frame is replaced entirely by the last good frame. On each consecutive frame erasure, the previous frame is attenuated before replacing the erased frame. After 320ms (16 frames) of consecutive frame erasures, silence is inserted. Attenuation of repeated frames is motivated by the fact that while speech is stationary in the short term, longer-term prediction of audio has a high error that users perceive as unnatural.

Repeating frames wholesale has the frequency domain effect of introducing harmonics every $\frac{1}{20ms} = 50Hz$ [43]. Thus, there will be a spike in the cepstrum[3] at the 20ms

[3] A "Cepstrum" is a signal representation defined as the inverse Fourier transform of the logarithm of the Fourier transform. A rough mental model is to think of the "cepstrum" as the "Fourier transform

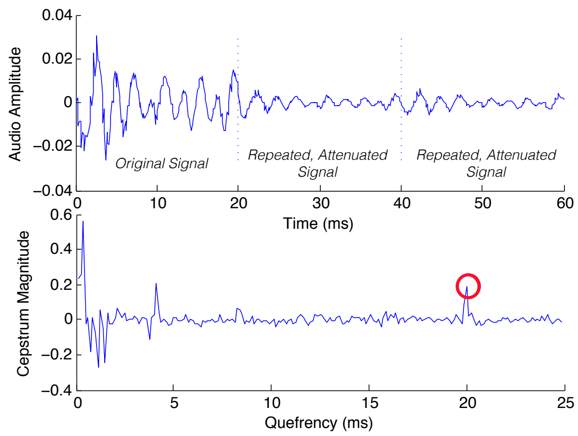


Figure 3: GSM-FR repeats and attenuates the last good frame to conceal packet loss. This results in a clear peak at 20ms in the cepstrum of the audio that can be used to detect a simboxed call.

quefrency. Because 50Hz is well below human pitch, this is a distinctive indicator of GSM packet loss. Figure 3 shows a clip of audio that has had GSM-FR packet loss concealment applied and the corresponding cepstrum. Note that the audio repeats (but is attenuated) every 20ms resulting in a peak at the cepstrum at 20ms. To detect GSM-FR PLC, Ammit computes the cepstrum of a window of three frames of audio and looks for a coefficient amplitude in the 20ms quefrency bin that is double the standard deviation of amplitudes of the other cepstral coefficients and not located in a silent frame.

4.4 Simbox Decision and SIM Detection

While concealed and unconcealed packet loss are measurable indicators of simboxing, there is a small false positive rate caused by the imperfection of our signal processing techniques. Accordingly, a single instance of a detected loss or concealed loss is not sufficient to consider a call to be originated from a simbox. Instead, we normalize the counts of loss events by the number of total frames in a call and consider a call as simboxed if the loss event percentage is much higher than the average loss event percentage for legitimate audio. We show in the following section that this approach is effective for all but the highest quality VoIP links, which provide few loss events to detect.

Even with this thresholding, some legitimate calls will occasionally be marked as a simbox. To ensure detection of simboxes with even improbably low loss rates, and to reduce the impact of false positives, we propose that the network should keep track of the number of times a call placed from a SIM is marked as a simboxed call. We term this technique “SIM detection” and show in the following sections that by using this technique we can further discriminate the legitimate subscribers from simboxers.

4.5 Efficiency of Ammit

Ammit is designed to analyze call audio in real time as it is received by the cellular tower. So, the system must be designed to function efficiently using minimal computation and network resources. To accomplish this, we avoid using costly analysis associated with machine learning or complex signal feature analysis, and instead apply simple threshold checks to processed audio signals. For each time window collected by Ammit, we apply two iterations of the Fast Fourier Transform (FFT) and a comparison operation to the distinguishing criteria noted above. The FFT is a well-known algorithm that can be run with $O(n \log n)$ complexity, and is used to analyze audio in real time for applications such as audio visualizers. We further verify empirically that these operations can be executed in real time in Section 7.

In addition, any added load on the network will cause a minimal impact on the overall throughput. While Traynor et al. [50] demonstrated that added signaling within the cellular network can cause a DDoS effect, Ammit sends only a single message to the HLR for any call flagged as a simboxed call. For this added messaging to cause an effect on the internal cellular network, a cell containing a simbox would have to simultaneously send significantly more messages than there are channels to handle cellular calls, which is not possible.

5 Threat Model and Evasion

To evade Ammit, simboxers must either compromise Ammit’s measurement abilities or successfully prevent or hide VoIP losses. While simboxers will take every economically rational action to preserve their profitability, attempting to evade Ammit will be difficult and likely expensive. This will hold true even if simboxers are aware of Ammit’s existence and detection techniques, and even if simboxers are able to place arbitrary numbers of calls to test evasion techniques. In this section, we outline basic assumptions about our adversary. We then provide details about how Ammit can be expanded to address stronger adversaries that could defeat the prototype described in this paper.

of the Fourier Transform of a signal. ”The domain of the function is termed “quefrency” and has the units of seconds

5.1 Security Assumptions

The effectiveness of Ammit relies on four reasonable assumptions to ensure that Ammit cannot be trivially evaded by simboxers. First, we assume that the Ammit system (hardware and software) is no more accessible to the attacker than any other core network system (including routing and billing mechanisms). Second, we assume that Ammit will be used to analyze all call audio so that simboxers cannot evade a known evaluation period. We show in Section 7.4 that Ammit can efficiently analyze calls. Third, we assume that Ammit will report measurements to a single location (like the HLR) so that simboxers cannot evade Ammit by frequently changing towers. Finally, we recommend that Ammit be widely deployed throughout a carrier's infrastructure because a wider deployment will provide fewer places for simboxes to operate.

5.2 Evasion

If the simboxer cannot avoid Ammit analysis, he must hide or prevent VoIP packet loss and jitter. Hiding packet loss and jitter was the very goal of over two decades of intense academic and industrial research that has so far only provided good *but algorithmically detectable* solutions, including jitter buffering and loss concealment.

Extreme jitter buffering VoIP clients (including simboxes) routinely use short audio buffers to prevent low levels of jitter from causing delays in playback. Simboxers could set the jitter buffer to a large value (say, several seconds of audio) to prevent jitter from causing noticeable audio artifacts. However, this would be intrusive to users, and Ammit could still detect true losses as well as the added false starts and double talk. While we leave the testing of this approach to future work, we briefly describe how high jitter buffers could be detected by measuring the incidence of double talk. Double talk is the phenomenon where, after a lull in conversation, two users begin to talk (apparently) simultaneously. Because double talk increases with audio latency, increased double talk will be indicative of increased latency. Because an increased jitter buffer (combined with the already high call latency from an international call) will lead to higher than "normal" latency, detecting anomalous double talk will help in detecting simboxing. Detecting double talk is an important task in equipment quality testing, and ITU-T standard P.502 provides an off-the-shelf method for measuring it. Feasibility and appropriate thresholds can be determined using call data through simboxes and from legitimate subscribers. While such data is unavailable to outside researchers (including the authors of this paper), it is available to the carriers who would be fielding such a system.

Alternative PLC approaches Ammit looks for brief silences as one signal of VoIP loss, so simboxers could replace silence with noise or other audio. This is a well known form of Packet Loss Concealment. In general, PLC algorithms (like the GSM-FR PLC) fall into three categories: insertion, interpolation, and regeneration [44]. Although there are a number of algorithms in each category, the majority are published (and those that are not are often similar to those that are). All will have some artifacts that can lead to detection, and because the Pindr0p project has developed techniques to identify other codecs[25], we leave detecting other PLCs as future engineering work not essential to confirming our hypothesis that audio features can identify simboxes.

Improved link quality In addition to jitter and loss concealment, simboxers could reduce losses and jitter with high-quality network links or a redundant transmission scheme, but there are several barriers to this. First, finding a reliable provider may not be possible given the low connectivity conditions in simboxing nations. If a provider is available, the costs will likely be prohibitive. For example, in Kenya one can expect to pay $200,000 US *per month* for a high-quality 1 Gbps link[40]. This connection also guarantees little beyond the first routing hop. Beyond the costs, having a better quality connection than many universities and businesses may raise undesirable scrutiny and attention to the simboxers. Even if a high-quality link is available, it would not remove degradations from the call that occur before the call arrives at the entry point to the simbox.

Garbled frame transmission Finally, Simboxers could evade Ammit detection by failing to transmit valid GSM air frames when an IP frame is lost. In effect, Ammit would believe that all VoIP losses were air losses and would not detect VoIP losses. Ammit could detect this evasion by noting anomalous air loss patterns.

Currently, conducting a simboxing operation requires the technical sophistication of systems administrator. This evasion technique will require significant engineering resources (with expertise in embedded system design, implementation, and production) because GSM modems are typically sold as packages that accept an audio stream and high-level control commands (e.g. "place a call" or "send an SMS"). These tightly integrated chips are not capable of sending damaged packets on command. While the Osmocom baseband project [20] could provide a start for a custom radio, Osmocom targets inexpensive (though relatively rare) feature phone variants and would not be a turnkey GSM baseband for a custom simbox[4]. Finally, even if the simboxers develop such a modem, they would have to conceal all detectable artifacts from both the final VoIP step as well as any intermediate networks (like a caller's mobile network). For these reasons, this strategy would only be effective for the most motivated and very well-funded simboxers.

However, in the event that simboxers do pursue this strategy, we propose the following methodology to detect such an attack. Given the considerable difficulty in developing the attack as well as constructing a suitable test environment, we leave testing this detection methodology to future work. We hypothesize that this a garbled packet evasion strategy can be detected from anomalous air interface loss patterns because simboxed calls will see the "typical" amount of loss *plus* the loss created by the simboxer. Loss patterns may be anomalous for improbable amounts of loss, or for improbably bursty sequences of lost frames. These anomalies could be determined on a tower-by-tower basis to take into account local transmission conditions (like a tunnel affecting signal quality). Because mobile stations (i.e. phones) do not know which frames are erased when they arrive at the tower, simboxers will not be able to tune their loss rate to be within the bounds used by this strategy.

6 Experimental Setup

In this section, we describe how we characterize Ammit through the use of simulation and test its effectiveness against a real simbox.

We simulate simboxed calls by taking a corpus of recorded audio and passing them first through a VoIP simulator then through a GSM air simulator (again, we use the term "air" to distinguish GSM cellular transmission). The GSM air simulator provides Ammit with both audio and a vector of GSM frame errors. To simulate legitimate calls, we pass the audio corpus through the air simulator only. We motivate the use of simulation in Section 6.6.

We test Ammit against three simbox codec choices: G.711 with no packet loss concealment and GSM-FR with and without packet loss concealment (we discussed this choice in Section 4. We evaluate single simbox call detection and SIM detection at 1%, 2%, and 5% loss rates (we justify this choice later in this section).

6.1 Speech corpus

The source of voice data for our experiments was the TIMIT Acoustic-Phonetic Continuous Speech corpus [33]. This is a *de facto* standard dataset for call audio testing. The TIMIT corpus consists of recordings of audio of 630 English speakers from 8 distinct regions each reading 10 "phonetically rich" standard sentences [5]. The recordings are 16kHz 16-bit signed Pulse Code Modulation (PCM), which are downsampled to 8kHz to conform to telephone quality. For the single call detection tests, we concatenate the 10 sentences for each of the 462 speakers into 1 call per speaker, creating a dataset of 462 calls [6]. Each call is approximately 30 seconds in length. The SIM detection test requires a larger call corpus, so for 98 randomly selected speakers we generate 20 calls for each speaker using permutations of the 10 sentences for each speaker (for a total of 1960 calls). Calls consist of only one speaker because Ammit analyzes each direction of the call separately.

6.2 VoIP Degradation and Loss

VoIP simulation takes TIMIT call audio as input and outputs audio that has been degraded by VoIP transmission. The simulator must convert the input audio from its original format (PCM) to the VoIP codec simulated (GSM or G.711), simulate loss, implement packet loss concealment in the case of GSM-FR, and output the final degraded audio. We examine these steps in greater detail in this subsection.

Audio conversions: The input audio files, encoded using PCM, must either be converted to G.711 or GSM-FR. We use the widely-used open source utility `sox` [8] for all codec transitions throughout the Ammit testing infrastructure. Note that these codec transitions are standard practice throughout PSTN and VoIP networks.

Packet Loss Modeling: We model Internet losses with the widely-used [39] Gilbert-Elliot packet loss model [34]. The Gilbert-Elliot model is a 2-state Markov model that models packet losses with bursty tendencies. A given channel can be in either a "good" state or a "bad" state. If the channel is in the "bad" state, packets are dropped. The Gilbert-Elliot model can be described with two parameters: p, the likelihood that the channel enters the "bad" state, and r, the likelihood that the channel leaves the bad state. p controls the frequency of loss events while r controls how long bursts last. We parameterize the model such that p is the target loss rate (for these experiments, 1%,2%, and 5%) and $r = 1 - p$. This means that the higher the loss rate, the greater the tendency of losses to be bursty.

Although jitter is a source of audio artifacts, we do not model jitter explicitly. Instead, because the audio symp-

[4] We pursued this line of research ourselves before finally purchasing a commercial simbox

[5] N.B. We use a subset of 462 male and female speakers from all 8 regions

[6] We set aside 12 of these calls as a training set to develop and verify our algorithms and set detection thresholds. These calls were not used for testing.

toms of jitter and packet loss are the same (i.e., audio is not present when needed), we simply consider jitter as a special case of packet loss, as is done by Jiang and Schulzrinne [39].

Loss Rate Justification: The reader may note that we are modeling loss rates that are considered high for Internet loss. Our model is justified for several reasons. The first consideration is that the typical Internet connection conditions in simboxing countries are of much lower quality than what most of Europe, East Asia, or even North America experiences [40, 51], with loss rates *often exceeding 10%*. Second, because conditions can vary from hour to hour or even moment to moment, examining performance at higher loss rates than typical is justified [39].

G.711 processing: To implement VoIP loss in G.711 audio, we use a packet loss simulation tool from the G.711 reference implementation available in the ITU Software Tools Library [7]. This tool implements concealed and unconcealed loss on 16-bit 8kHz PCM audio. We use `sox` to encode our input files to G.711 and back to 16-bit PCM before processing by the tool. This step is required because G.711 is a lossy codec, and the act of encoding and decoding irreversibly changes the audio. The tool takes a frame error vector as input, allowing us to use the Gilbert-Elliot Model described above.

GSM-FR processing: We developed our own GSM-FR VoIP loss simulator in Matlab. All audio processing in this tool is done on GSM-FR encoded audio. The tool implements the previously discussed packet loss model, the GSM-FR PLC as defined in 3GPP Standard 46.011 [24], and unconcealed packet loss by inserting GSM-FR silent frames.

6.3 GSM Air Loss

As we discussed in Section 6.6, we simulate simbox calls out of necessity. To simulate GSM cellular transmission (i.e., "air loss") we modify a GSM Traffic Channel simulation model for Simulink [41]. This model takes frames of GSM-encoded audio and encodes them as transmission frames for transmission over a GSM traffic channel as specified in 3GPP Standard 45.003 [21]. The transmission encoding includes interleaving as well as the error correcting codes and parity checks applied to Class 1 bits (as discussed in Section 4).

The model then simulates the modulation and transmission of the encoded frame using GMSK (Gaussian Minimum Shift Keying) in the presence of Gaussian white noise in the RF channel. This white noise is the source of random transmission errors in the model.

The model then demodulates the transmitted channel frame, evaluates the error correcting codes, and computes the parity check to determine if the frame is erased or not. Finally, the model outputs the received audio and a vector indicating which frames were erased.

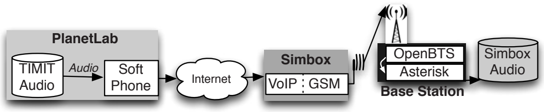


Figure 4: Our detection mechanisms are run against a real simbox deployment (Hybertone GoIP-1) communicating to a modified Range Networks OpenBTS base station.

The channel model signal-to-noise ratio is tuned to produce a frame erasure rate (FER) of 3% at the receiver, which is considered nominal according to 3GPP Standard 45.005 [22].

6.4 Simboxing SIM Detection Test

Our SIM detection mechanism is tailored to reduce the effect of a single false positive or false negative call judgment by examining multiple calls.

To measure the effectiveness of this mechanism, we use 20 audio files from 98 unique speakers (for a total of 1960 calls) to simulate legitimate and simboxed calls using our GSM and VoIP simulators. We examine legitimate calls as well as simboxes covering all three codecs (GSM-FR, GSM-FR with PLC, and G.711) at 1%, 2%, and 5% loss rates. We model individual SIM cards as groups of 20 calls. For legitimate SIM cards, all calls from a particular speaker are assigned to a single SIM card, while simbox SIM cards consist of groups of randomly selected calls. This models the fact that simbox SIMs will rarely be used to provide service for the same user twice.

We analyzed all legitimate and simboxed calls with Ammit, then computed the percentage of calls in each SIM card group that were marked as simboxed. We consider a SIM to be used in a simbox if at least 25% of the calls it makes are marked as simboxed by Ammit call analysis.

6.5 Real Simbox Tests

We collect audio traces from calls made through a real simbox to validate our simulation experiments.

Figure 4 shows a schematic diagram of our experimental setup. We use 100 randomly selected audio files from the single call detection corpus (discussed in Section 6.1) to model the original call source. The call path begins at a PJSIP soft phone at a PlanetLab node located in Thai-

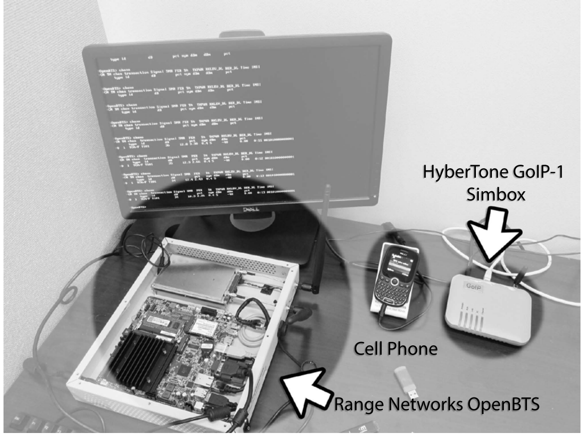


Figure 5: Our simbox experimental apparatus, including our OpenBTS GSM base station, mobile phone to model legitimate calls, and our GoIP-1 simbox.

land, a country with major simboxing problems [46][7]. This step emulates the arrival of a call to a simboxer.

The call originates from a soft phone and is routed through an Asterisk PBX[8] (not shown in the figure) to our Hybertone GoIP-1 simbox in the United States. Hybertone simboxes offer useful features to simboxing, including the ability to automatically change the IMEI number broadcast to evade filtering and detection systems like those presented in prior work [42]. Hybertone products have been advertised for sale specifically for simboxing [2], and entrepreneurs even sell value-added management consoles specifically for simboxers [1]. While the GoIP-1 supports several incoming codecs, it does not disclose which PLC algorithm it uses. We have determined experimentally that it is using a variant of the GSM-FR PLC.

The simbox delivers the call to a cellular base station under our control. Our base station is a Range Networks Professional Development Kit running the OpenBTS 5.0 open-source base station software and Asterisk 11.7. This base station is a low-power research femtocell and allows us to record call audio digitally as the base station receives it – including frame erasure information. To determine false positives, we create control calls by playing the same 100 randomly-selected audio files into a BLU Samba Jr. Plus feature phone and capturing the call audio at the base station. Figure 4 shows our base station and simbox experimental apparatus.

6.6 Technical Considerations

Our experimental setup uses both simulation and real simbox data we collect ourselves for several reasons. First, simulations provide the best way to examine the effects of codec choice, packet loss concealment, and loss rates reproducibly and accurately. Second, they allow us to build generic models of simboxes so that our detection mechanism is not tied to any particular simbox model. Third, because we use tools and models that are extensively studied, verified, and frequently used throughout the literature [25, 52, 34, 39, 7, 41], we can have confidence that our results are correct. We supplement our simulations with data collected through a commonly used simbox to support and confirm our simulation results.

The reader will note that our real simbox calls were originated in a simboxing country, not terminated there. While simboxing is a global problem [42], we wanted to focus on areas where the problem is endemic and has a substantial impact. However, logistical, economic, and legal considerations prevented us from placing our simbox and research base station abroad. Instead, we capture the exact loss and jitter characteristics of the Internet connections in a simboxing country by originating the call there while terminating the call in our lab.

Legal and privacy concerns prevent us from receiving simbox audio from mobile operators (since the audio would be from callers who could not give their consent for such use). However, we note that there are no additional privacy concerns created when an operator deploys Ammit in a real network. Operators regularly use automated techniques to monitor call quality of ongoing conversations, and Ammit does no analysis that could be used to identify either the speakers or the semantic content of the call.

Finally, we note that the use of TIMIT audio is extremely conservative; it presumes pristine audio quality before the call transits an IP link. In fact, there will be detectable degradations from the PSTN even before the VoIP transmission. Chief among these will be GSM-FR PLC applied if Alice calls from a mobile phone. Because mobile phones regularly see high loss rates[9], simboxers carrying mobile-originated traffic will be even more vulnerable to detection by Ammit than this methodology reflects.

7 Detection Results

This section demonstrates how Ammit detects simbox fraud. We first discussed Ammit's effectiveness at identifying a real simbox, followed by a discussion of the

[7] Note that Thailand is the only major simboxing country with functional PlanetLab node at the time of writing

[8] A Private Branch Exchange (PBX) is a telephony switch analogous to an intelligent router in the Internet

[9] Recall from 3GPP standard 45.005 [22] that 3% loss is considered nominal

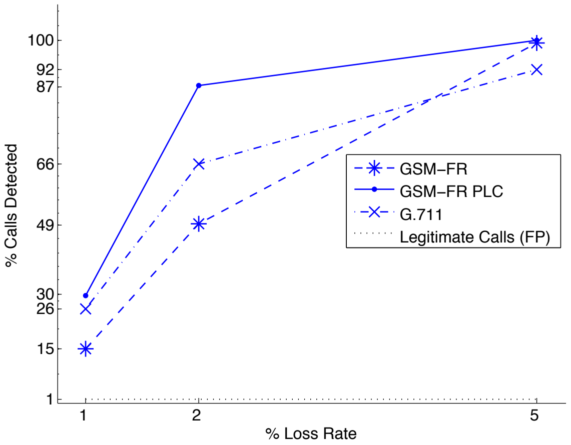


Figure 6: Ammit detection depends on the loss rate and Simbox codec used. For a 2% loss rate, Ammit detects over 55% of simboxed calls with less than a 1% false positive rate. This performance makes SIM detection (shown in Fig. 7) very reliable.

results of detecting simulated simboxed calls. We then examine how Ammit can be used to identify SIM cards used in simboxing fraud. Finally, we show that Ammit is fast enough to be effective in real networks.

7.1 Simulated Call Analysis

In this subsection, we evaluate Ammit's ability to detect individual simboxed calls and SIM cards used in simboxing.

Figure 6 presents the percentage of simboxed calls detected for three simbox types at three different loss rates. At the still plausible 5% loss rate, Ammit detects from 87% to 100% of simboxed calls. Lower detection rates for low loss rates are simply a result of fewer loss events for Ammit to detect. However, in the case of no packet loss concealment, Ammit still detects from 15–66% of the simboxed calls for 1 and 2% loss. As discussed in the previous section, these loss rates include the effect of jitter, so loss rates as low as 1% and 2% are unlikely to be encountered often in practice [40, 51].

Third, the lowest dotted line in Figure 6 shows the low (but non-zero) detection rate for the control group of simulated legitimate calls — less than 1% (0.87% to be exact).

Figure 7 shows the percentage of simbox SIM cards that can be automatically disabled at the threshold of 25% of calls. For a 5% loss rate, our policy can identify *100%* of SIM cards used in simboxes. For calls using GSM-FR with packet loss concealment our policy can also detect 100% of SIM cards. As the loss rates decrease, we identify fewer SIM cards for codecs without packet concealment. In the case of 2% loss, we identify 96% and 100% of SIMs used in GSM-FR and G.711 simboxes, respectively. In the case of 1% loss, we still identify 43% of G.711 SIMs and 28% of GSM-FR SIMs. Our threshold results in a false positive rate of 1% and was determined experimentally from a ROC curve (omitted for space reasons). To counter the effects of false positives, the operator could implement a simple policy step allowing users to reactivate canceled SIM after some verification. One possibility is requiring flagged users to verify the National ID numbers used to register the SIM card over the phone or in person at a sales agent.

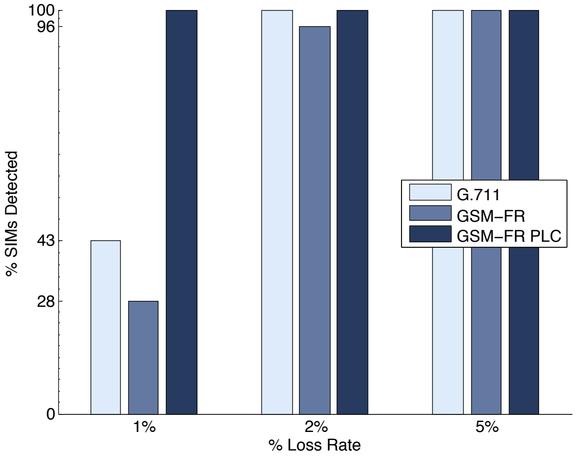


Figure 7: Even with unusually high-quality network connections, Ammit can be used to identify SIM cards used for simboxing.

7.2 Detection of Real Simboxes

We begin with the most important result that Ammit is effective at detecting real simboxes. We find Ammit can detect 87% of real simboxed calls with *zero* false positives on the call set. These figures are the result of running our GSM-FR packet loss concealment after tuning on simulated individual call data; improved detection may be possible at a cost of a low false positive rate. While simulations produce useful insights about Ammit's performance in a wide range of conditions, these results confirm our hypothesis that call audio can be used to effectively combat simbox fraud.

7.3 Discussion

We make three observations from the individual call simulations. First, the results show a clear relationship between the loss rate of a call and Ammit's ability to detect a call. Second, Figure 6 shows the counterintuitive result that using GSM-FR packet loss concealment makes calls

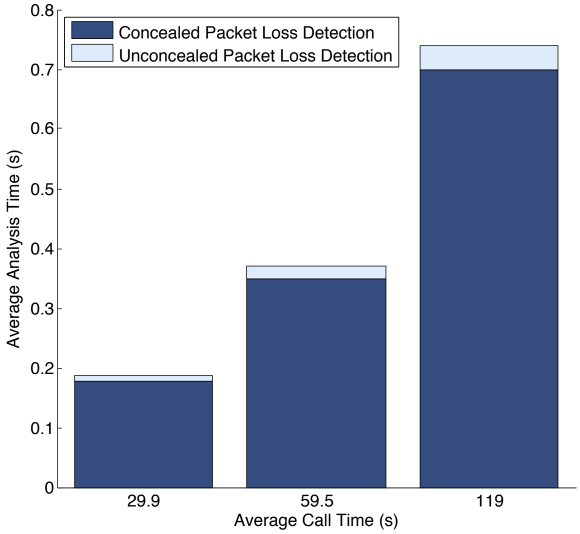


Figure 8: Ammit analyzes audio much faster than real time and is efficient enough to deploy in cell towers.

easier to detect. Even at a 1% loss rate, Ammit detects 30% of simboxed calls using GSM-FR PLC. Ammit is so effective at detecting concealed packet loss events because the GSM-FR PLC cepstral peak is distinctive and rare in speech. The corollary to this finding is that simboxers will have an incentive to disable packet loss concealment. This will noticeably impair call quality and user acceptability. Third, the non-zero false positive rate means that discretion will be required when Ammit indicates a positive simbox call.

Our SIM detection results show that Ammit can be used not only to detect single calls but as a larger initiative against simboxing. At 2% and 5% loss, we can detect and disable a single SIM card after at most 20 calls. Even at 1% loss, we can still detect and disable many SIM cards. Given that SIM cards come at a nontrivial cost (either at a legitimate point of sale or on a black market), by reducing the lifetime of a SIM card we make simboxers unable to operate.

Finally, we make two observations from the real simbox results. First, we note that our simulations were effective for tuning Ammit before applying real data. This validates our methodological strategy. Second, our simulation false positive rates were conservatively high; while we saw 1% false positives on our simulated data, we saw no false positives on our actual data.

7.4 Ammit Performance

To show that Ammit is scalable and performant, we examine the amount of time Ammit requires to analyze a call for concealed and unconcealed packet losses. Although in the previous subsections we analyzed Ammit's performance for 30 second calls, we hypothesize that longer analyses would lead to even better results, especially for lower loss rate calls. We tested Ammit's performance on a set of 10 calls of approximately 30s, 60s, and 120s; we present the averages of 10 analyses of each call in Figure 8.

We test Ammit on a late 2011 iMac with a quadcore 3.4 GHz Intel Core i7, 16GB RAM, and a 1TB solid state disk running OS X 10.9. Although this is capable hardware, the detection is done entirely with Matlab in a single thread, and the detection code is correct but far from optimal. Optimizing the Matlab code for efficiency would likely reduce analysis time. Beyond that, implementing Ammit in a more performant language like C could reduce analysis time further. For a commercial implementation, code customized for a digital signal processor could further improve performance. Ammit may be deployed directly as a BTS or BSC software update or as inexpensive standalone hardware.

As Figure 8 shows, the majority of analysis time is spent detecting concealed packet loss. Nevertheless, calls can be analyzed 150 times faster than real time, indicating that a single thread of execution could analyze approximately 150 calls per unit time. Even our unoptimized code would be able to analyze all traffic at a tower in real time.

8 Related work

Although this work is concerned with detecting simbox fraud, the techniques used belong to the long tradition of non-intrusive call quality measurement. Non-intrusive measurements are taken passively and without a reference audio; this is in opposition to intrusive measurements [4, 6] which measure the degradation of a known reference signal. Traditional call quality metrics measure listener experience, and imperceptible degradations do not significantly affect these scores. These scores have been shown to vary widely based on random conditions, language choice [48] or VoIP client [26]. The most widely used non-intrusive measurement standard is ITU specification P.563 [5], but other metrics have been developed for holistic quality measurements [32, 37, 30] and for individual artifacts like robotization [43] and temporal clipping [36]. Because call quality metrics like P.563 are only concerned with perceptible degradation and vary widely in results, they are unsuitable for detection of simbox fraud.

Telephony fraud detection is a well-studied problem, and efforts to fight telecommunications fraud have primarily depended on call records. Machine learning and data mining have been used extensively to detect fraudulent activity using call records [27, 29, 35, 45].

Given the importance of the simboxing problem in affected countries, there are a number of commercial simbox detection products, as well as two published research papers [31, 42]. Most simbox detection systems use one of two techniques: test call generation and call record analysis. A few products use hybrid techniques [14, 17]. Test call generation approaches [10, 13, 15, 16, 18] use probes widely deployed in many networks to verify that the CLI (i.e. Caller-ID) records on calls are correct — if a simbox is used, the CLI record would indicate the MSISDN (i.e. the phone number) of the SIM card routing the call and not the originating probe. Test call methods only work for certain kinds of simboxing (when a simboxer sells services to another telecom, not through the common case of selling calling cards to consumers). By contrast, call record analysis detect all types of simboxing. Those approaches rely on the fact that SIMs used in simboxes have usage patterns distinct from legitimate customers [31, 9, 11, 12, 19]. These techniques are prone to false positives and active evasion by simboxers. In recent work, Murynets et al. published a call record analysis approach that used machine learning to identify IMEIs (device identifiers) used by simboxes [42]. The authors' published accuracy rates measure identifying individual calls (not simbox devices) only after simboxes are identified, and thus are not directly comparable to the accuracy figures for Ammit. Additionally, that work identifies IMEIs (which are an asserted — and thus spoofable — identifier) of devices only after a simbox makes dozens or hundreds of calls with a single SIM card; even if the work described in that paper is deployed, simboxing will continue to be profitable. Our work is an improvement over the state of the art because we can reliably detect simboxed calls using features inherent to simboxing *at the time of the call*, thus making simboxing unprofitable.

While Ammit is the first system to combat simboxing using call audio, our system is a refinement of the ideas used in the Pindr0p system developed by Balasubramanian et al˙ [25].

The Pindr0p system combats telephony fraud by identifying callers using audio "fingerprints." These fingerprints consist of noise characteristics and indicators of different codecs used by the different PSTN and VoIP networks that route a call. For Pindr0p, capturing characteristics of end-to-end call path is essential to identify repeat callers. For Ammit, it is sufficient to hear audio that has been degraded by any prior network.

Ammit's techniques are tailored to better combat simboxing in several important ways.

First, for simbox detection, Pindr0p's greatest weakness is that it is designed to identify codec transitions. Accordingly, Pindr0p would fail to detect a simbox call that uses a single codec (i.e. the GSM codec) end to end. If Pindr0p were employed to combat simboxing, simbox operators would simply migrate to single-codec solutions. Accordingly, Ammit will detect simboxed calls that Pindr0p would fail to detect. In fact, by using loss information from the tower (and not endpoint audio) Ammit excels at this case better than any other tested.

Second, we showed that Pindr0p techniques used on individual calls may result in unacceptable false positives. This was especially true for the silence insertion detection methods proposed in that work. For Pindr0p, a classification error may prompt a call center worker to request additional authentication. For Ammit, a classification error will prompt a dropped call or suspended account — a much higher burden to the user. Ammit enhances Pindr0p by developing and verifying a SIM detection method that reduces false positives and increases confidence in classification.

Third, Pindr0p was designed to quickly fingerprint audio based on a short segment using a large number of features fed to a machine learning classifier. While we were unable to obtain access to Pindr0p for a direct comparison, we *do* show in this work that Ammit's techniques will enable real-time processing of all call audio at mobile network base stations.

9 Conclusions

Cellular networks in developing nations rely on tariffs collected at regulated interconnects in order to subsidize the cost of their deployment and operation. These charges can result in significant expense to foreign callers and create incentive for such callers to find less expensive, albeit unlawful, means of terminating their calls. Simboxes enable such interconnect bypass fraud by tunneling traffic from a VoIP connection into a provider network without proper authorization. In this paper, we develop the Ammit tool, which allows us to detect simboxes based on measurable differences between true GSM and tunneled VoIP audio. Ammit uses fast signal processing techniques to identify whether individual calls are likely made by a simbox and then to develop profiles of SIM cards. This approach allows a provider to deactivate the associated SIMs rapidly, and virtually eliminates the economic incentive to conduct such fraud. In so doing, we demonstrate that the subsidized rates that allow much of the developing world to be connected can be protected against the impact of this fraud.

10 Acknowledgments

The authors are grateful for the help of our shepherd, Srdjan Capkun, and for comments from our anonymous reviewers. Michael Good was instrumental in early work on this project that was not published in this paper, and conversations with Charles Lever were especially helpful to work out the kinks.

This work was supported in part by the US National Science Foundation under grant numbers CNS-1464088 and CNS-1318167. This work was also supported in part by the Harris Corporation. 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 or the Harris Corporation.

References

[1] GoAntifraud.com.

[2] Goip For Grey Route SIM Box. `http://www.alibaba.com/product-detail/16-ports-gsm-gateway-goip-for_862885942.html`.

[3] ITU-T recommendation G.711.

[4] ITU standard P.800 methods for subjective determination of transmission quality.

[5] ITU standard P.563:single-ended method for objective speech quality assessment.

[6] ITU standard P.862:perceptual evaluation of speech quality (PESQ).

[7] *ITU Software Tool Library Manual*. ITU, Geneva, 2009.

[8] sox. `http://sox.sourceforge.net/Main/HomePage`, 2013.

[9] Agilis international simbox detection. `http://www.agilisinternational.com/solutions/customer-analytics/risk-and-fraud-management/`, 2014.

[10] Araxxe SIM box detection. `http://www.araxxe.com/SIM-box-detection.html`, 2014.

[11] CxB solutions SIM box detection. `http://www.cxbsolutions.com/html/sim_box_detection.html`, 2014.

[12] FraudBuster SIMBuster. `http://www.fraudbuster.mobi/new-simbuster-and-trafficchecker-deployment-in-africa/`, 2014.

[13] Meucci solutions SIM box detection. `http://www.meucci-solutions.com/solutions/fraud-and-revenue/sim-box-detection/`, 2014.

[14] Mobius fraud mangement. `http://www.mobiusws.com/solutions/fraud-management/`, 2014.

[15] Mocean SIM box detector. `http://www.mocean.com.my/SIM_box_detector_solution.php`, 2014.

[16] Roamware SIM box detector. `http://www.roamware.com/predictive_intelligence_sim_box_detector.php`, 2014.

[17] ROC fraud management. `http://www.subex.com/pdf/bypass-fraud.pdf`, 2014.

[18] Telenor simbox detection. `http://www.telenorglobal.com/wp-content/uploads/sites/4/2013/09/Global-SIM-Box-Detection1.pdf`, 2014.

[19] XINTEC SIM box detector. `http://www.xintec.com/fraud-management/sim-box-detector/`, 2014.

[20] OsmocomBB GSM baseband. `http://bb.osmocom.org/trac/`, 2015.

[21] 3rd Generation Partnership Project. 3GPP TS 45.003 v12.0.0. Tech. Rep. Channel coding.

[22] 3rd Generation Partnership Project. 3GPP TS 45.005 v12.1.0. Tech. Rep. Radio transmission and reception.

[23] 3rd Generation Partnership Project. 3GPP TS 46.010 v11.1.0. Tech. Rep. Full rate speech; Transcoding.

[24] 3rd Generation Partnership Project. 3GPP TS 46.011 v11.1.0. Tech. Rep. Full rate speech; Transcoding.

[25] Balasubramaniyan, V. A., Poonawalla, A., Ahamad, M., Hunter, M. T., and Traynor, P. PinDr0p: using single-ended audio features to determine call provenance. In *Proceedings of the 17th ACM conference on Computer and communications security* (New York, NY, USA, 2010), CCS 10, ACM, p. 109120.

[26] Broom, S. VoIP quality assessment: Taking account of the edge-device. *IEEE Transactions on Audio, Speech, and Language Processing 14*, 6 (2006).

[27] Burge, P., and Shawe-Taylor, J. An unsupervised neural network approach to profiling the behavior of mobile phone users for use in fraud detection. *Journal of Parallel and Distributed Computing 61*, 7 (July 2001), 915–925.

[28] Communications Fraud Control Association (CFCA). 2013 Global Fraud Loss Survey. `http://www.cvidya.com/media/62059/global-fraud_loss_survey2013.pdf`, 2013.

[29] Cox, K. C., Eick, S. G., Wills, G. J., and Brachman, R. J. Visual data mining: Recognizing telephone calling fraud. *Data Mining and Knowledge Discovery 1*, 2 (June 1997), 225–231.

[30] Ding, L., Lin, Z., Radwan, A., El-Hennawey, M. S., and Goubran, R. A. Non-intrusive single-ended speech quality assessment in VoIP. *Speech Communication 49*, 6 (June 2007), 477–489.

[31] Elmi, A. H., Ibrahim, S., and Sallehuddin, R. Detecting SIM box fraud using neural network. In *IT Convergence and Security 2012*, K. J. Kim and K.-Y. Chung, Eds., no. 215 in Lecture Notes in Electrical Engineering. Springer Netherlands, Jan. 2013, pp. 575–582.

[32] Falk, T., and Chan, W.-Y. Single-ended speech quality measurement using machine learning methods. *IEEE Transactions on Audio, Speech, and Language Processing 14*, 6 (2006), 1935–1947.

[33] Garofolo, J. S., Lamel, L. F., Fisher, W. M., Fiscus, J. G., Pallett, D. S., Dahlgren, N. L., and Zue, V. *TIMIT Acoustic-Phonetic Continuous Speech Corpus*. Linguistic Data Consortium, Philadelphia, 1993.

[34] Hasslinger, G., and Hohlfeld, O. The Gilbert-Elliott model for packet loss in real time services on the Internet. In *Measuring, Modelling and Evaluation of Computer and Communication Systems (MMB), 2008 14th GI/ITG Conference* (March 2008), pp. 1–15.

[35] Hilas, C. S., and Mastorocostas, P. A. An application of supervised and unsupervised learning approaches to telecommunications fraud detection. *Knowledge-Based Systems 21*, 7 (Oct. 2008), 721–726.

[36] Hines, A., Skoglund, J., Kokaram, A., and Harte, N. Monitoring the effects of temporal clipping on VoIP speech quality. In *14th Annual Conference of the International Speech Communication Association* (2013), ISCA.

[37] Hoene, C., Karl, H., and Wolisz, A. A perceptual quality model intended for adaptive VoIP applications: Research articles. *Int. J. Commun. Syst. 19*, 3 (Apr. 2006), 299316.

[38] Jalil, M., Butt, F., and Malik, A. Short-time energy, magnitude, zero crossing rate and autocorrelation measurement for discriminating voiced and unvoiced segments of speech signals. In *Technological Advances in Electrical, Electronics and Computer Engineering (TAEECE), 2013 International Conference on* (May 2013), pp. 208–212.

[39] Jiang, W., and Schulzrinne, H. Comparison and optimization of packet loss repair methods on VoIP perceived quality under bursty loss. In *Proceedings of the 12th International Workshop on Network and Operating Systems Support for Digital Audio and Video* (New York, NY, USA, 2002), NOSSDAV '02, ACM, pp. 73–81.

[40] Les Cottrell, R. Pinging Africa - a decade long quest aims to pinpoint the Internet bottlenecks holding Africa back. *Spectrum, IEEE 50*, 2 (Feb 2013), 54–59.

[41] Molina, J. A. S. GSM traffic channel simulator. `http://www.mathworks.com/matlabcentral/fileexchange/11078-gsm-traffic-channel-simulator`, 2006.

[42] Murynets, I., Zabarankin, M., Jover, R., and Panagia, A. Analysis and detection of SIMbox fraud in mobility networks. In *2014 Proceedings IEEE INFOCOM* (Apr. 2014), pp. 1519–1526.

[43] Paping, M., and Fahnle, T. Automatic detection of disturbing robot voice and ping-pong effects in GSM transmitted speech. In *EUROSPEECH* (1997).

[44] Perkins, C., Hodson, O., and Hardman, V. A survey of packet loss recovery techniques for streaming audio. *IEEE Network 12*, 5 (1998), 40–48.

[45] Qayyum, S., Mansoor, S., Khalid, A., Khushbakht, K., Halim, Z., and Baig, A. Fraudulent call detection for mobile networks. In *2010 International Conference on Information and Emerging Technologies (ICIET)* (2010), pp. 1–5.

[46] Ratana, U. Telcos lose money to SIM fraud. *Phnom Penh Post* (Feb. 2014).

[47] Schulzrinne and Casner. RFC 3551. Tech. Rep. RTP Profile for Audio and Video Conferences with Minimal Control.

[48] Takahashi, A., Kurashima, A., and Yoshino, H. Objective assessment methodology for estimating conversational quality in VoIP. *IEEE Transactions on Audio, Speech, and Language Processing 14*, 6 (2006), 1984–1993.

[49] Traynor, P. Characterizing the Security Implications of Third-Party EAS Over Cellular Text Messaging Services. *IEEE Transactions on Mobile Computing (TMC) 11*, 6 (2012), 983–994.

[50] Traynor, P., Lin, M., Ongoing, M., Rao, V., Jaeger, T., McDaniel, P., and La Porta, T. On cellular botnets: Measuring the impact of malicious devices on a cellular network core. In *Proceedings of the 16th ACM conference on Computer and communications security* (2009).

[51] Wang, Y., Huang, C., Li, J., and Ross, K. Queen: Estimating packet loss rate between arbitrary internet hosts. In *Passive and Active Network Measurement*, S. Moon, R. Teixeira, and S. Uhlig, Eds., vol. 5448 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2009, pp. 57–66.

[52] White, A. M., Matthews, A. R., Snow, K. Z., and Monrose, F. Phonotactic reconstruction of encrypted voip conversations: Hookt on fon-iks. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2011), SP '11, IEEE Computer Society, pp. 3–18.

A Appendix

Table 1: Commercial VoIP GSM Gateway Survey

Brand	Selected Codecs
2N VoiceBlue Next	G.711, G.729ab
2N StarGate	G.711, G.729a
OpenVox VoxStack	G.711, G.729, GSM
Dinstar DWG2000B	G.711, G.729ab
ElGato K32	G.711, G.729
Gempro GP-708	G.711, GSM
iQsim M400	G.711, G.729(a&ab)
Nicherons SpoGSM-G4	G.711, GSM, G729
PORTech MV-378	G.711, G.729(a&ab)
SunComm SC-024-S	G.711, G.729ab
Hybertone GoIP-1	G.711, G.729ab, GSM
Yeastar NeoGate TG800	G.711, G.729a

GSMem: Data Exfiltration from Air-Gapped Computers over GSM Frequencies

Mordechai Guri, Assaf Kachlon, Ofer Hasson, Gabi Kedma, Yisroel Mirsky[1], Yuval Elovici[1]
{gurim, assafka, hassonof, gabik, yisroel, elovici}@post.bgu.ac.il
Ben-Gurion University of the Negev, Beer-Sheva, Israel
[1] Telekom Innovation Laboratories at Ben-Gurion University, Beer-Sheva, Israel

Abstract

Air-gapped networks are isolated, separated both logically and physically from public networks. Although the feasibility of invading such systems has been demonstrated in recent years, exfiltration of data from air-gapped networks is still a challenging task. In this paper we present *GSMem*, a malware that can exfiltrate data through an air-gap over cellular frequencies. Rogue software on an infected target computer modulates and transmits electromagnetic signals at cellular frequencies by invoking specific memory-related instructions and utilizing the multi-channel memory architecture to amplify the transmission. Furthermore, we show that the transmitted signals can be received and demodulated by a rootkit placed in the baseband firmware of a nearby cellular phone. We present crucial design issues such as signal generation and reception, data modulation, and transmission detection. We implement a prototype of GSMem consisting of a transmitter and a receiver and evaluate its performance and limitations. Our current results demonstrate its efficacy and feasibility, achieving an effective transmission distance of 1 - 5.5 meters with a standard mobile phone. When using a dedicated, yet affordable hardware receiver, the effective distance reached over 30 meters.

1. Introduction

Security-aware organizations take various steps to prevent possible theft or leakage of sensitive information. The computers responsible for storing and processing sensitive information often operate on air-gapped networks. These networks are physically disconnected from non-essential networks, primarily those in the public domain. With the growing awareness of negligent or malicious insiders compromising air-gapped networks, as evidenced in several incidents [1] [2], some organizations have begun to restrict USB access, to prevent malware infection or data leakage via USB thumb-drives [3].

Acknowledging the security risks of mobile phones equipped with cameras, Wi-Fi, or Bluetooth, some organizations has restricted their use, forbidding them in classified areas. For instance, an Intel Corporation best-practices document [4] asserts: "*Currently, manufacturing employees can use only basic corporate-owned cell phones with voice and text messaging features. These phones have no camera, video, or Wi-Fi.*" In another case, visitors at one of Lockheed-Martin’s facilities [5] are instructed as follows: "*Because ATL is a secure facility, the following items are not allowed to our floor of the building: cameras (film, video, digital), imaging equipment, tape recorders, sound recording devices. Cell phones are allowed, but camera/recording features may not be used*." Similar regulations are likely to be found in many other security-aware organizations. Clearly, the issue of information leakage associated with basic cellular phones or a phone without a camera, Wi-Fi and the like, has been overlooked in cases in which such phones are allowed in the vicinity of air-gapped computers. However, modern computers are electronic devices and are bound to emit some electromagnetic radiation (EMR) at various wavelengths and strengths. Furthermore, cellular phones are agile receivers of EMR signals. Combined, these two factors create an invitation for attackers seeking to exfiltrate data over a covert channel.

In this paper, we present an adversarial attack model in which any basic desktop computer can covertly transmit data to a nearby mobile phone. Transmission is accomplished by invoking specific memory-related CPU instructions that produce baseband compliant EMR at GSM, UMTS, and LTE frequencies. By using the functionality of multi-channel memory architecture, the signals are amplified and transmitted with increased power. These signals are received and decoded by a rootkit installed at the baseband of a standard mobile phone. To demonstrate the feasibility of the attack model, we developed *GSMem*, a bifurcated malware that consists of a transmitter that operates on a desktop computer and a receiver that runs on a GSM mobile phone. We implemented communication protocols for data modulation and channel reliability and provide extensive experimental results.

As will be explained later, the proposed method is applicable with GSM, UMTS, and LTE basebands. In this paper we focus on a prototype using a GSM mobile phone as receiver, hence the codename, *GSMem*.

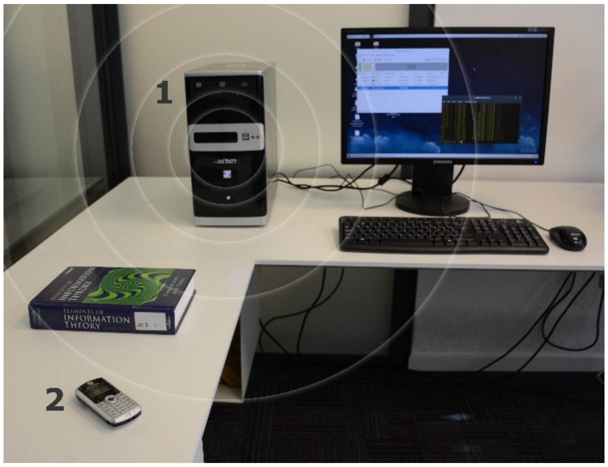


Figure 1: Demonstration of the covert channel in a working environment. Signals at GSM frequencies are emitted from the workstation and received by the nearby compromised mobile phone.

Figure 1 demonstrates the covert channel in a typical real-life scenario, in which rogue software on a computer (1) modulates sensitive information and transmits it over GSM cellular frequencies. The transmissions take place while the computer is at work, without affecting the user experience. A baseband level rootkit on the cellular phone (2) receives the signals and demodulates them, converting them into meaningful information. Note that the components exploited by the proposed model are present on virtually all computers and cellular devices, even on low-end cellular devices which are often allowed into classified environments.

1.1. The Closed Nature of the Baseband Industry

The baseband chip of a cellular device manages the low-level Radio Frequency (RF) connection with the cellular network, thereby making it an indispensable component. The baseband processor runs a real time operating system (RTOS), stored in its firmware. The code is closed to the public, and only the device manufacturer can access the baseband chip's functionality through a limited interface [6]. The RTOS source code, along with the protocol stack and other implementation details, are well-guarded trade secrets, kept off-limits by the protective baseband industry, which is led by a handful of high-ranking players that dominate the market [7]. Lacking access to this information, including documentation and implementation details, independent software vendors cannot intelligently develop new products and interfacing technologies for baseband chips.

It can be argued that the current state of affairs promotes "security through obscurity" by masking the internal workings of the baseband systems. However, this policy has only limited effectiveness. Skilled hackers working on behalf of advanced persistent attackers eventually manage to exploit baseband systems—obscure and isolated though they may be. Baseband exploitation and attacks are thoroughly discussed by Weinmann [8] [9] [10]. Welte and Markgraf [6] also point out several security problems associated with current commercial baseband technology and practices.

1.2. Paper Contributions

While emission security (EMSEC) in itself is not a new concept [11], this paper offers the following original contributions: (1) a novel method for transmitting signals at cellular frequency bands from an ordinary desktop computer, using multi-channel memory related CPU instructions without any special or additional hardware, and (2) a novel method for receiving and demodulating EMR signals using a rootkit in the baseband firmware of a mobile phone, thus turning virtually any mobile phone into an effective EMR eavesdropping device without the use of specialized equipment. We believe the proposed adversarial attack model constitutes a new security threat that security experts should be aware of.

While the bulk of this paper focuses on the mobile phone as a receiver, we also evaluate an alternative communication method in which the transmitter uses memory-related CPU instructions to emit EMR, and the receiver uses software defined radio (SDR) with dedicated, yet affordable hardware. This allows us to study the capabilities and boundaries of the transmission method on a wider scale.

The remainder of this paper is organized as follows: In Section 2 we present assorted related works, along with a concise review of our contributions. Next, in Section 3, we present the adversarial attack model. In Section 4 we present the essential technical background. Section 5 provides a detailed description of the transmitter, followed by Section 6 which describes the receiver. In Section 7 we evaluate GSMem and present the results. Next, in Section 8, we discuss possible defensive countermeasures. Finally, we conclude in Section 9.

2. Related Work

EMSEC, reviewed by Anderson [11], addresses attacks which use compromised emanations of either conducted or radiated electromagnetic signals. Concern about this issue dates back to World War I, but for decades it was relegated solely to governmental and military agencies [12]. However in 1985, van Eck [13] showed how the so-called TEMPEST exploits can be conducted using affordable equipment. He managed to reconstruct an image from electromagnetic signals produced by a video card at a considerable distance, using a modified TV set. Around 2000, Kuhn and Anderson released several publications related to TEMPEST [14] [15], demonstrating that EMR emissions originating from a desktop computer can be manipulated by appropriate software, in either a defensive or offensive manner. Public interest in EMSEC and TEMPEST was amplified by web publications, offering a glimpse into classified TEMPEST-related official standards [16], or providing 'do it yourself' tutorials related to TEMPEST exploits. Thiele [17] provides an open source program dubbed "TEMPEST for Eliza", utilizing the computer CRT monitor to modulate and transmit radio signals at AM frequencies.

Note that side-channels have a variety of possible uses, beyond intentional exfiltration of information as described in this paper. Side-channels may be used for eavesdropping, attacking sophisticated encryption methods, defensive detection of hidden malicious activities, and other uses. Furthermore, side-channels are not limited to electromagnetic radiation (EMR). Clark, Ransford et al [18] refer to power consumption as a side-channel that can reveal hidden information or activities. They present 'WattsUpDoc', a system that detects the presence of malware on medical embedded devices by measuring their power consumption. Rührmair et al [19] discuss the use of power and timing side-channels to attack physical unclonable functions (PUFs). Other researchers investigating side-channels go beyond EMR emanations. Halevy and Saxena [20], explore acoustical eavesdropping, focusing on keyboard acoustical emanations. Hanspach and Goetz [21] present so-called "covert acoustical networks". Their method is based on near-ultrasonic waves, transmitted by the speaker of one laptop computer and received by the microphone of a nearby laptop computer. Callan et al [22] provide a method for measuring the so-called "signal available to the attacker" (SAVAT), with a side-channel based on instruction-level events. Their method is based on the EMR emitted by rather generic CPU/memory operations. The receiver, however, comprises expensive dedicated equipment, and the range of explored distances is quite limited. Guri et al [23] present AirHopper, a bifurcated malware in which the transmitter exploits the EMR emanated by the VGA cable. The receiver is an FM-enabled standard cellular phone.

2.1 Comparison of Relevant Covert Channels

Current state-of-the-art covert channels methods that could be used to exfiltrate data from air-gapped networks involve various physical effects, such as FM transmissions from a display cable [23], ultrasonic acoustic emissions from a speaker [21] [24], EMR emitted by generic CPU operations [22], and thermal emission [25]. Our method, GSMem, uses emissions produced by multi-channel memory data bus. Table 1 provides a brief comparison between GSMem and other current models.

Method	Transmitter	Receiver	Distance (m)	Rate (bit/s)
AirHopper **[23]** (78MHz -108MHz)	Display cable	Cellular FM receiver	7	104-480
Ultrasonic **[21] [24]**	Speaker	Microphone	19.7	20
SAVAT **[22]** (~80KHz)	CPU/memory (laptops)	Dedicated equipment	1.0	N/A
BitWhisper [25]	Computer CPU/GPU	Computer Heat Sensors	0.4	8 bit/hour
GSMem (cellular frequencies)	RAM bus (multi-channel)	Baseband	5.5	1-2
		Dedicated equipment	30+	100-1000

Table 1: Comparison of current covert channels for air-gapped networks

As can be seen, all five methods utilize basic computer equipment as the transmitter. However, whereas a display cable or a speaker may not be present on every conceivable computer configuration [26], the CPU and memory, utilized by GSMem and SAVAT, are always present. On the receiver's end, a microphone may not be present on every computer, particularly within a classified zone [26]. A cellular FM receiver (as used by AirHopper) may not be present on every mobile phone, while the baseband processor (used by GSMem) is an integral part of any mobile phone.

In terms of bandwidth, with the dedicated hardware receiver we achieved bit rates of 100 to 1000 bit/s. However, when using a mobile phone as the receiver, the bit rate was much slower (2 bit/s) – making this equipment suitable for leaking small amount of data. It is important to note that our concept was demonstrated on a nine year old low-end phone, the only available alternative with open source firmware, given the protective nature of the baseband industry. Demonstrating the same concept on newer basebands will likely yield better results, and is left as a future research direction.

3. The Adversarial Attack Model

GSMem, viewed as a concept, contributes to the general domain of covert channels. However, we describe a particular attack model which might utilize this covert channel for the purpose of data exfiltration. The adversarial attack model is bifurcated since it requires both a contaminated computer to serve as a *transmitter* and a contaminated mobile phone to serve as a *receiver*. Infecting a computer within an air-gapped network can be accomplished, as demonstrated by the attacks involving Stuxnet [27] [28], Agent.Btz [2] and others [1] [29] [30] [31]. Compromising a mobile phone can occur via social engineering, malicious apps, USB interface, or physical access [32] [33] [34]. Once a compromised mobile phone is in the vicinity of an infected computer, it can detect, receive and decode any transmitted signals and store the relevant acquired information. Later, the phone can transmit the data to the attacker via mobile-data, SMS, or Wi-Fi (in the case of smartphones). Although this attack model is somewhat complicated, attackers have grown more sophisticated, and complex attack patterns have increasingly been proven feasible during the last few years [35] [36] [37] [38].

4. Technical Background

The exfiltration channel is based on the emission of electromagnetic signals, in the frequencies allocated to cellular bands. These signals can be picked up by a malicious component located at the baseband level of a nearby mobile phone. In this section, we provide an overview and some helpful technical background information about cellular networks and frequency bands, along with the basics of baseband components in mobile phones.

4.1. Cellular Networks

2G, and the newer 3G and 4G networks are three 'generations' of mobile networks. Each generation has its own set of standards, network architecture, infrastructure, and protocol. 2G, 3G, and 4G networks are commonly referred to as GSM, UMTS, and LTE respectively, generally reflecting, the implementation of these standards. In this paper, we use the terms GSM, UMTS, and LTE to denote the three generations.

4.1.1. Cellular Network Bands

Wireless communication between mobile-handsets (i.e., mobile phones) and the cellular network takes place through a base transceiver station (BTS), which handles the radio link protocols with the handsets. Communication with the BTS takes place over 'frequency bands' allocated for the cellular network. Various standards define the radio frequencies allocated to each band. In practice, the standard in use depends on the country, region, and support of the cellular provider. Modern mobile phones support all common frequency bands for GSM, UMTS, and LTE, although some phones are region specific. Table 2 shows the main frequency bands supported by modern mobile phones. Each band encompasses frequencies within a range surrounding (above and below) the main frequency. For example, GSM-850 has a frequency range between 824.2MHz and 894.2MHz. Lists of bands and their allocated frequencies are specified by the standards [39].

Standard	Frequency band (MHz)
GSM	850 / 900 / 1800 / 1900
UMTS	850 / 900 / 1900 / 2100
LTE	800 / 850 / 900 / 1800 / 1900 / 2100 / 2600

Table 2: The main frequency bands for GSM, UMTS and LTE cellular networks.

4.1.2. ARFCN

The communication (transmission and reception) between the mobile phone and the BTS occurs over a subset of frequencies within the entire frequency band. The absolute radio-frequency channel number (ARFCN) specifies a pair of radio carriers used for transmission (uplink) and reception (downlink) in GSM networks. For example, the GSM-850 band consists of 123 ARFCN codes (ARFCN 128 to ARFCN 251), in which the ARFCN 128 code represents the uplink frequency of 824.2MHz and the downlink frequency of 869.2MHz. In UMTS and LTE, the ARFCN are replaced with UARFCN and EARFCN respectively. The mapping of each ARFCN on the corresponding carrier frequency is given in [40].

4.2. Baseband in Mobile Phones

Modern mobile phones consist of at least two separate processors [9] [41]. The application processor runs the main operating system (e.g., Android or iOS) and is responsible for handling the graphical user interface, memory management and process scheduling. The baseband processor runs a dedicated RTOS which manages the radio communication and maintains the protocol stack. The application processor and the baseband processor work independently from one another and have separate memory space. However, it is necessary to exchange data between the two processors on a routine basis, for example, when the dialer application initiates a call (application processor to baseband processor) or when an SMS notification is received (baseband processor to application processor). Communication between the processors is commonly handled through a shared-memory segment or a dedicated serial interface [9] [41]. Unlike modern smartphones, low-end mobile phones, also referred to as feature-phones, employ a single processor to manage both user-interface and cellular communication. On feature-phones, this single processor is also referred to as a baseband processor.

4.2.1. Baseband Chip Architecture

The baseband processor is an integral part of the baseband chip. The chip consists of: (1) the RF frontend, (2) the analog baseband, (3) the digital baseband, and (4) the baseband processor [6] [41].

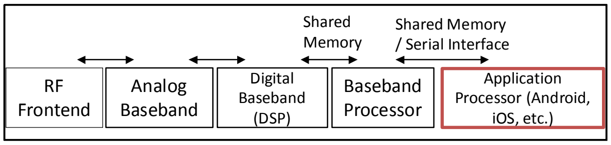


Figure 2: The baseband components and application processor in modern mobile phones. In low-end phones, an application processor doesn't exist.

The **RF frontend** handles received and transmitted signals on the physical level. This component consists of items such as: an antenna, a low-noise amplifier (LNA), and a mixer. The **analog baseband** contains, among other components, an analog to digital converter (ADC) and a digital to analog converter (DAC) to mediate between the digital baseband and the RF frontend. The **digital baseband** includes the digital signal processor (DSP) which is responsible for the lowest parts of the protocol stack (i.e., modulation/demodulation and error-correction). The **baseband processor** is responsible for handling the higher and more complex layers of the protocol stack. Communication between the DSP and the baseband processors takes place through a shared-memory interface (Figure 2).

5. The Transmitter

The physical effect underlying our transmission method is electromagnetic radiation (EMR), a form of energy emitted by certain electromagnetic processes. The emitted waves propagate through space in a radiant manner. Electromagnetic waves have two defining properties: the frequency f measured in Hertz (Hz) and the amplitude (i.e., strength) measured in decibel-milliwatts (dBm). In many cases, electronics (such as wiring, computer monitors, video cards, and communication cables) emit EMR in the radio frequency spectrum. Their frequencies and amplitudes depend on their internal currents and voltage. An exploitation of intentional and unintentional emissions from computer components has been addressed in previous research [14] [23] [13] [42].

We propose that a computer's memory bus can be exploited to act as an antenna capable of transmitting information wirelessly to a remote location. When data is exchanged between the CPU and the RAM, radio waves are emitted from the bus's long parallel circuits. The emission frequency is loosely wraps around the frequency of the RAM's I/O bus clock with a marginal span of +/-200MHz. The casual use of a computer does not generate these radio waves at significant amplitude, since it requires a major buildup of voltage in the circuitry. Therefore, we have found that by generating a continuous stream of data over the multi-channel memory buses, it is possible to raise the amplitude of the emitted radio waves. Using this observation, we are able to modulate binary data over these carrier waves by deterministically starting and stopping multi-channel transfers using special CPU instructions.

In the remainder of this section, we describe the design and implementation of the transmitter from the bottom up. First, we discuss the carrier wave (channel frequency) of the emitted radio waves. Next, we discuss a method for modulating binary data over a bus. Last, we propose a simple bit framing protocol to help the receiver demodulate the received signal. It is important to note that since the focus of this paper is the feasibility of the proposed covert channel, we do not exhaustively explore all possible signal modulations or bit framing protocols. Improvements to the communication protocol are a subject of future research.

5.1. EMR Emissions

Multi-channel memory architecture is a technology that increases the data transfer rate between the memory modules and the memory controller by adding additional buses in between them. The address space in multi-channel memory is spread across the physical memory banks, consequentially enabling data to be simultaneously transferred via multiple (two, three, or four) data buses. In this way, more data can be transferred in each read/write operation. For example, motherboards with dual-channel support have 2x64 bit data channels. Some computers support triple-channel memory and modern systems even have quadruple-channel support. Multi-channel architecture is implemented in all modern Intel and AMD motherboards.

In Figure 3, the radio emissions from an ordinary desktop workstation with dual channel memory are plotted on the frequency plane, comparing emissions from casual activity to those associated with intentional actions. When all channels are used, the radio emissions from the buses increase (red) in comparison to the emissions from casual activity (blue). We observed an increase of at least 0.1 - 0.15 dB across the frequency band 750-1000MHz, where some specific sub-bands showed an increase of about 1 - 2.1dB. A full summary of the radio emissions of different motherboards and memory technologies can be found in Table 3.

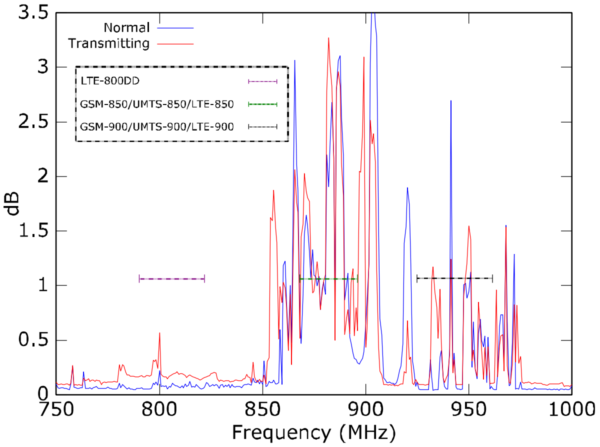


Figure 3: A plot of the amplitude of the radio waves emitted from a motherboard with an 800MHz I/O bus using DDR3-1600 RAM. Blue: casual use of the computer. Red: our transmission algorithm while using the dual channel data paths.

Based on our experiments, we have found that the use of three or four channels increases amplitude emissions across nearly the entire band depicted in Figure 3. This means that as the memory architectures mature, the quality of the proposed covert channel will increase. Note that these radio emissions fall within the frequency bands of GSM, UMTS and LTE, making them detectable by all modern basebands.

Standard Name	I/O bus clock (f_c)	EMR Range
DDR3-1600	800MHz	600MHz-1100MHz
DDR3-1866	933MHz	750MHz-1150MHz
DDR4-2133	1066MHz	750MHz-943MHz (fragmented) 1.04GHz-1.066GHz

Table 3: Summary of radio emissions from different memory buses.

5.2. Signal Modulation

In communications, modulation is the process where analog waveforms are varied to carry information over some medium. Typically, a carrier wave (for wireless a radio wave at the frequency f_c) is selected as the channel frequency, where most of the energy from the modulation can be found in the band around f_c.

There are many techniques for modulating a carrier wave to carry binary data. For simplicity and as a show of feasibility, we use a variant of the two level amplitude shift keying (B-ASK) modulation; to send a ‘1’ or ‘0’ the transmitter raises or lowers the amplitude of f_c accordingly over set time intervals T (in seconds) [43]. In other words, the time domain is partitioned into intervals of length T, and the symbol (i.e., signal amplitude) that corresponds to the current bit is transmitted over that entire interval. Our variation of B-ASK is that ‘0’ is not represented by a near zero amplitude, but rather by the average level of the casual emissions. It is assumed that the receiver can differentiate between average and high emission levels (described in detail later in Section 6). The motherboard bus’s radio emissions can be modulated to carry a B-ASK signal in the following way: to transmit a ‘1’ all memory channels are utilized for T seconds, and to transmit a ‘0’ nothing special is done (casual emissions are emitted). In this case, f_c is the motherboard’s memory clock.

5.3. Modulation Algorithm

In order to transmit a ‘1’, it is necessary to consistently utilize multiple memory channels for T seconds. To do this we generate a long random data transfer from the CPU to the main memory using the single instruction multiple data (SIMD) instruction set. SIMD utilizes special CPU registers of 64-bits and 128-bits in order to process wider chunks of data in a single instruction. SIMD instructions are usually used for vectorized calculations such as 2D/3D graphics processing, and includes instructions to load/store data between the main memory and special registers.

Algorithm 1 transmit32 (data)

```
buffer ← ALIGNED_ALLOCATE(16,4096)
tx_time ← 500000
for bit_index ← 0 to 32 do
    if (data[bit_index] = 1) then
        start_time ← CURRENT_TIME()
        while (tx_time > CURRENT_TIME() - start_time) do
            buffer_ptr ← buffer
            for i ← 0 to buffer_size do
                SIMDNTMOV(buffer_ptr, 128bit_register)
                buffer_ptr ← buffer_ptr + 16
            end for
        end while
    else
        SLEEP(tx_time)
    end if
end for
```

We implemented the B-ASK modulation algorithm using the Streaming SIMD Extension (SSE) instruction set found in Intel and AMD CPUs. The SSE specifies a set of 128-bit (quadword) registers numbered xmm0-xmm16, and includes a group of instructions for moving data between these xmm registers and the main memory [44] [45]. Using these instructions it is possible to instruct the CPU to utilize the multi-channel data paths, thereby amplifying the radio emissions.

One of the challenges we had to overcome resulted from the use of the CPU caching mechanisms. When the processor employs a cache hierarchy, transferring data between xmm registers and the main memory does not guarantee any immediate activity over the bus. This inconsistency presents an issue regarding the use of the proposed B-ASK modulation, since the symbols must start and stop precisely within the symbol interval (T).

Beginning with SSE version 2, there is a set of instructions that enable read/write operations directly to/from the main memory, while bypassing all cache levels (non-temporal). Specifically, we use the Move Double Quadword Non-Temporal instruction, MOVNTDQ *m128, xmm*. The intent of this instruction is for copying double quadwords from the xmm register to the 128-bit memory address, while minimizing pollution in the cache hierarchy.

Our implementation of the B-ASK modulation (Algorithm 1) works in the following way. The *transmit32()* method receives the outbound binary as an array of 32 bits. A temporary buffer of 4096 bytes (32x128) is allocated on the heap (lines 1-2) as a destination for the MOVNTDQ memory operations. Note that the allocated memory has to be 16-bytes aligned, as required for SSE memory operands. Next, on line 2, we set T to 500ms. Although a shorter T would provide a faster bit transmission rate, doing so directly increases the error rate. For the tested Motorola C123 phone with the Calypso baseband, a value of 500ms appears to provide satisfying results. Basebands of modern smartphones are probably capable of higher sampling quality, and therefore might require a shorter T. With specialized receiver hardware, setting T to 1-10ms provided good reception quality (Section 6).

The outer loop (line 3) iterates over the 32-bit array and performs the memory operations to generate the radio emissions. When the current bit is a '1' a loop repeatedly uses the MOVNTDQ instruction to copy data from xmm registers to the heap, until T seconds have elapsed. Conversely, when the current bit is a '0' the algorithm sleeps for T seconds.

5.4. Bit Framing

As mentioned earlier, when our variant of B-ASK modulates a '0' the amplitude of the transmitted signal is that of the bus's average casual emissions, and anything significantly higher than that (by some threshold) is considered a '1'. This incurs two issues: (1) the receiver has no prior information as to what the optimum threshold should be making it difficult for the receiver to detect activity in its area, and (2) the strength of amplitudes surrounding f_c is dependent on the distance between the transmitting desktop and the receiver; this means that if the mobile phone is moving during a transmission or other interference exists, a '1' and '0' can be decoded incorrectly.

Therefore, in order to assist the receiver in dynamically synchronizing with the transmitter, we place the data into frames. The binary stream is partitioned into sequential payloads of 12 bits, and the payloads are transmitted with a header consisting of the preamble sequence '1010' (Table 4). The preamble is used by the receiver to determine when a frame is being transmitted and to determine the amplitude levels of a '1' and a '0'. This process is discussed in depth in Section 6. The framing process takes place before data transmission. Once the frame has been built, it is passed to Algorithm 1 as the outbound data.

Preamble	*Payload*	*Preamble*	*Payload*
1010	12 bits	1010	12 bits

Table 4: The basic frame format used to send segments of a bit stream, using the *transmit32()* function.

5.5. Transmitter Stealth and Compatibility

The transmitting program has a small memory and CPU footprint, making the activities of the transmitter easier to hide. In terms of memory consumption, the program consumes merely 4K of memory allocated on the heap. In terms of CPU intake, the transmitter runs on a single, independent thread. At the OS level, the transmitting process can be executed with no elevated privileges (e.g., root or admin). Finally, the code consists of bare CPU instructions, avoiding API calls to escape certain malware scanners. In short, the transmission code evades common security mechanisms such as API monitoring and resource tracing, making it hard to detect.

As for compatibility, since 2004 SIMD instructions have been available for x86-64 Intel and AMD processors [46] [47], making the transmission method is applicable to most modern workstations and servers. Similar instructions on IBM's Power architecture have been in place since Power ISA v.2.03 was initiated [48]. The proposed transmitter has been implemented and successfully tested on several operating systems, including Microsoft Windows platform (Windows 7, 64bit), Linux Fedora 20 and 21 (64bit), and Ubuntu 12.1 (64bit).

6. The Receiver

In this section we describe how a mobile phone in close proximity to a transmitting computer can successfully receive and decode emitted signals. We implement the GSMem receiver component by modifying the firmware of a mobile phone's baseband. We present the receiver architecture and implementation, along with the modulation and decoding mechanisms. Interestingly, we found that under certain circumstances, the GSMem signals can be indirectly received by an application running on a modern Android smartphone with a non-modified baseband. This optional implementation yields rather limited effective distance of 10cm, and provides a conceptual rather than an immediate practical contribution. Therefore, to stay in line with the core of this paper, the description of this implementation is deferred to Appendix A.

6.1. Receiver Implementation

Reception of the transmitted data is accomplished in the following manner: (1) sample the amplitude of the carrier wave, (2) performs noise mitigation, (3) search for bit frame header (preamble detection), and (4) demodulate the frame's payload. We will describe each of these steps in this order after discussing the implementation framework.

6.1.1. Baseband Firmware

As discussed in Section 1, the baseband industry is highly protective, keeping information about baseband architecture, the RTOS, and the protocol stack, guarded from the public [9] [10] [49]. The secrecy and complexity of the baseband technology makes it extremely difficult to make modifications at the binary level, particularly without the availability of information such as source code [10] [49]. However, there have clearly been cases where attackers have used explicit access to the device firmware in order to perform malicious activities [29] [31] [33] [50]. Our implementation of the GSMem receiver is based on '*OsmocomBB,'* an open source GSM baseband software implementation [51].

The open source project, launched in 2010, is the only way to freely examine the implementation of a mobile's GSM baseband software. OsmocomBB provides source code for the GSM protocol stack, along with device drivers for digital and analog basebands chips. The project currently supports about 13 models of mobile phones. Most of the supported phones are OEM by Motorola and works with Calypso baseband chipsets made by Texas Instruments. For our experiments, we selected the Motorola C123 model [52] that supports 2G bands but has no GPRS, Wi-Fi, or mobile data traffic capabilities. The Motorola C123 is a limited mobile phone, supporting our attack scenario described in Section 3. It is worthwhile to note that the baseband components of modern smartphones are more advanced in terms of RF reception, sampling rate and processing power due to the improved hardware and the support in new technologies such as the LTE [6] [53]. That means that implementation of the GSMem receiver on modern device may yield better results in terms of reception quality and transfer-rates.

The GSM protocol stack at the baseband consists of three main layers [49]. Layer 1 is the most relevant layer in term of GSMem implementation. It handles the RF interface which modulates the data over the air. In OsmocomBB, the lower part of the layer 1 is handled by the DSP, while the baseband processor handles the upper layers. Layer 1 includes, among other functionalities, the power management, which is responsible for acquiring the raw signal power measurements (in dBm) of specific frequencies (ARFCNs). Note that measuring RF power levels is a basic functionality of any baseband chip [39]. The interaction between the baseband processor and the DSP is depicted in Figure 4.

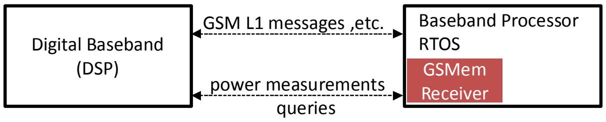


Figure 4: Interaction between the baseband processor and the DSP.

6.1.2. Firmware Modification

The receiver is implemented by patching the main event handler in the baseband RTOS. Figure 5 shows the outline of the OsmocomBB initialization and main loop. After initialization (lines 1-2), the baseband processor enters the event loop (line 3). The event loop continuously processes a sequence of event handlers, including the keypad handler, timer updates, and layers 2 and 3 handlers, interrupts from the DSP, power measurements, etc.

```
1: // baseband initialization
2: // specific receiver initialization
3: while true do
4:     // keypad handlers
5:     // layer 2/3 handlers
6:     // power management handler
7:     // ...
8:     ReceiverHandler()
9: end while
```

Figure 5: Calypso RTOS code outline

In order to implement the functionalities of the receiver, we added a routine of our own called *ReceiverHandler()* (line 8). Since it is placed in the main loop, the routine is run continuously at every iteration.

The *ReceiverHandler()* has three possible states: (1) scan for best frequency (2) search for bit frame header (preamble), and (3) B-ASK signal demodulation. Scan state is the initial state of the routine. The pseudo code for *ReceiverHandler()* is presented in Algorithm 2.

Algorithm 2 ReceiverHandler

```
1:  $dBm \leftarrow$ MEASURE($f_c$)
2:  $filtered\_signal \leftarrow$ UPDATEMOVINGAVERAGE($dBm$)
3:  if ($state = SCAN$) then
4:      $f_c \leftarrow$ SCANFREQ()
5:      SETSTATE($PREAMBLE$)
6:  end if
7:  if ($state = PREAMBLE$) then
8:      if (IDENTIFYPREAMBLE($filtered\_signal$) $= true$) then
9:          SETSTATE($RECEIVE$)
10:     end if
11: end if
12: if ($state = RECEIVE$) then
13:     $b \leftarrow$ DEMODULATEBIT($filtered\_signal$)
14:     $bitSequence$.add($b$)
15:     if ($bitSequence$.size%16 $= 0$ or SIGNALLOST($filtered\_signal$)) then
16:         SETSTATE($PREAMBLE$)
17:     end if
18: end if
```

6.1.3. Signal Sampling

The first step in detecting a GSMem transmission is to sample the amplitude of the carrier wave f_c. Note that this step takes place only after f_c has been determined in a initial scanning phase. Each time the main loop runs *ReceiverHandler(),* Algorithm 2 causes the DSP module to sample the power level (amplitude) of f_c (line 1) and stores it in a buffer (line 2). This data is used later in the demodulation routines. The function *Measure()* invokes an amplitude measurement request on the DSP using a function called *l1a_l23_rx()*. The DSP measurements are performed in bands of 0.2MHz. Our tests show that the tested Calypso baseband was able to sample power measurements at a rate of 1.8kHz, hence 1.8kbps is the fastest bit rate that this device can demodulate at. This is a much faster bit rate than we achieved due to the limited processing capabilities of the device. However, the power measurements rate is an important consideration to take into account when implementing an improved GSMem receiver on a more advanced device in the future.

6.1.4. Noise Mitigation

After the power measurement, a noise mitigation function is applied to the current sample by averaging it with the last W original samples. This operation is essentially a moving average filter, an effective technique for mitigating high frequency noise. In our experiments with the Motorola C123, we tried a W of 50-750 samples and found that the size of W directly affects the bit rate. A larger W provided better noise mitigation, while a smaller one produced a faster bit rate.

6.1.5. Detecting the Best Carrier Wave

In the *SCAN* state, the receiver searches for the best f_c to use for demodulating GSMem transmissions. Note that since the radio emissions of the transmitter fallout across the GSM-850/GSM-900 bands (Figure 3, Section 5.1), the f_c can be set in advance to any frequency in those bands. However, we observed that some frequencies have more interference than others (e.g., the channels actively used by nearby cellular base stations). Therefore, during the scanning state, the better f_c is determined as the frequency that provides the best carrier to interference ratio (CIR). This frequency is found by scanning the range of the entire GSM-850 range and selecting the frequency with the minimum average amplitude (in dBm). The assumption is that the minimum average amplitude indicates a low level of interferences, making it easier to detect a '1' using our variant of B-ASK. In our implementation, the scanning takes place after the device boots, and after every 30 seconds of noisy or lost signals. After the f_c value is set, the algorithm moves to the *PREAMBLE* state.

6.1.6. Preamble Detection and Demodulation

If state is set to *PREAMBLE*, the receiver searches for a preamble sequence (lines 7-11 of Algorithm 2). If the sequence '1010' is detected, then it is assumed to be the start of a frame, and state is changed to *RECEIVE* to complete the B-ASK demodulation process (lines 12-18). The preamble sequence allows the GSMem receiver to: (1) synchronize with the GSMem transmitter (2) identify '1' and '0' amplitude levels δ and (3) determine the signals' duration t, if unknown. Dynamically setting δ for every frame is necessary for demodulating signals while the mobile is moving. For example, a frame may be received at close proximity to the transmitter where f_c is much stronger thereby setting amplitude levels to be high. The subsequent frame may be sent while the mobile phone is farther away – where smaller amplitude would be more appropriate. Once a preamble has been detected, the payload is demodulated in a similar manner using the updated parameters.

6.1.7. Signal Loss

On line 15 in Algorithm 2, the state of the receiver returns to *PREAMBLE* if the whole payload has received, or if the signal has been lost. The function *SignalLost()* returns true if during the data reception, the measured signal power is weaker than the amplitude of the ‘0’s from the preamble for three seconds straight. In this case, any partially received data discarded or marked appropriately.

7. Evaluation

In this section we evaluate GSMem’s performance as a communication channel. We present in detail the evaluation using a tampered cellular baseband receiver. We also examine the signal reception using a dedicated hardware receiver programmed via software defined radio (SDR).

7.1. Experiment Setup

We used the Motorola C123 with the modified firmware as the receiver for all experiments in this section. As for the transmitters, we used three different models of desktop workstations (WS), each with a different configuration and different case. The details of these computers and their tested settings can be found in Table 5. Note that WS3 is a much stronger transmitter than the others since its RAM has a quad channel operation mode, which employs wider data paths. In all the experiments, the transmitter used the 4kb allocation method described in Section 5, with a T of 1.8 seconds. The receiver listened to the carrier frequency (f_c) ARFCN 25 downlink (940MHz), unless otherwise mentioned.

	WS1	**WS2**	**WS3**
OS	Linux Fedora 20		
Chassis (metal)	infinity chassis	GIGABYTE Setto 1020 GZ-AX2CBS	Silverstone RL04B
CPU	Intel i7-4790	Intel i7-3770	Intel i7-5820K
Motherboard	GIGABYTE GA-h87M-D3H	GIGABYTE H77-D3H	GIGABYTE GA-X99-UD4
RAM Type	2 x 4GB 1600MHz		4 x 4GB 2133MHz
RAM Frequencies Tested	1333/1600 MHz		1833/2133 MHz
RAM Operation Modes Tested	Single / Dual		Dual / Quad

Table 5: Configuration of the transmitting workstations.

There are several major factors that affect the quality of a wireless communication channel. Typically, the quality of a channel is measured by taking the signal to noise ratio (SNR), where $SNR \equiv 10\log(P_{signal}/P_{noise}) = P_{signal}dB - P_{noise}dB$ and P is the power level (a larger SNR is better than a smaller one). The noise power P_{noise} can originate from naturally occurring noise and from other interferences such as the emissions from nearby computers in the same office space. Therefore, in order to match our attack scenario from Section 3, the experiments in this section all take place in a regular work space with several active desktop workstations within a 10m radius.

There are many factors which can decrease the SNR of a wireless channel when the location of the receiver is changed. Because we are dealing with a low power transmission, we do not consider properties such as multipath propagation (fading). Instead, we focus on how different receiver distances and positions affect the channel’s SNR.

7.2. Channel Signal to Noise Ratio (SNR)

The first set of experiments tests the SNR of the WSs from different distances. Figure 6, Figure 7 and Figure 8 show the receiver’s maximum measured amplitudes at different distances from WSs 1, 2, and 3 respectively. Here, WSs 1 and 2 have their RAM set to dual mode at 1600MHz, and WS3 has its RAM set dual / quad mode at 1833 / 2133MHz. As illustrated by Figure 9, the SNR remains positive (more signal power than noise) even up to a distance of 160cm. This gives a good indication of the proposed covert channel’s effective distance. Given these observations, we assume that a distance of 160cm from a workstation is within the normal range where a mobile device is expected to be held while working on the workstation.

Note that WS3 in dual mode has a significant advantage in range over WSs 2 and 3. This is due to the fact that WS3 uses a higher RAM frequency than all other WSs in the workplace scenario. This means that it is subject to less interference, thereby improving its SNR. When quad channel mode is used, the range increases further, demonstrating that a higher number of active memory channels increases the signal’s amplitude.

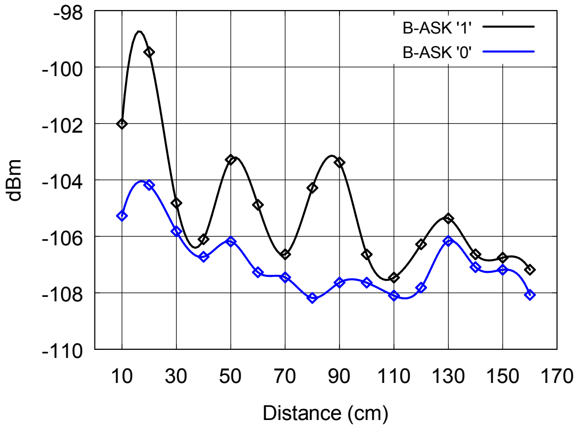


Figure 6: Signal strength received from WS2 (1600MHz, Dual) at various distances from the backside of the chassis. The blue line can also be viewed as the casual emissions (noise).

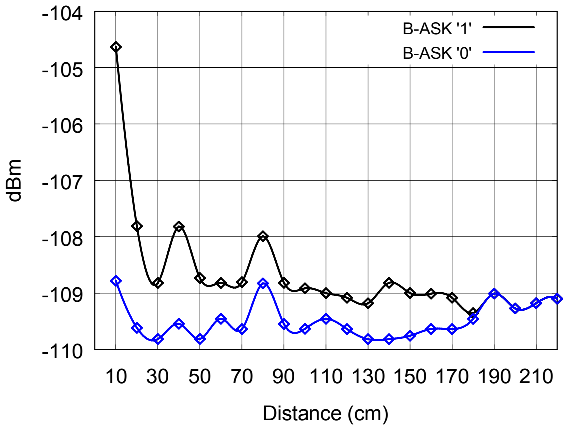


Figure 7: Signal strength received from WS1 (1600MHz, Dual) at various distances from the backside of the chassis.

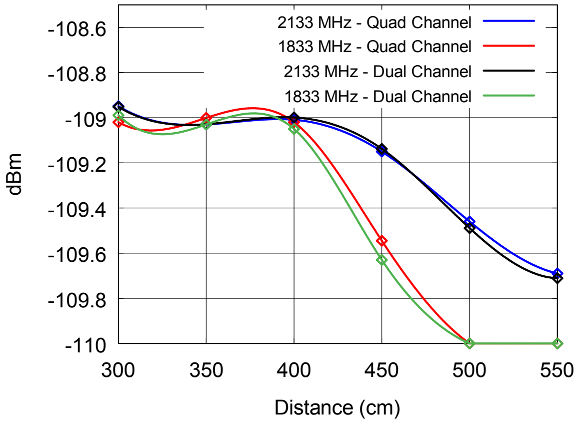


Figure 8: Signal strength received from WS3 (1833/2133MHz, dual/quad channels) at various distances from the front side of the chassis.

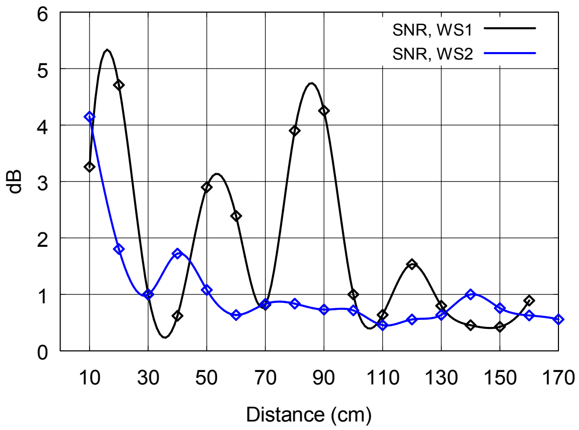


Figure 9: Receiver SNR from WS1 and WS2 (1600MHz, Dual) at various distances from the backside of the chassis.

During the experiments, we observed that the position of the receiver with respect to the transmitter has a significant impact on the SNR. For instance, using WS2, an SNR of 0.5 is achieved at a farther distance from the front of the chassis as opposed to the back. Furthermore, the best position for WS1 (using 1600MHz) is from the front, while the best position for WS2 is from the back. These differences make sense considering that each case has variations in shape and metal content. In all cases, we observed that the optimum position for the receiver to be is in front of the chassis. This may have to do with the fact that the front of an ATX case is mainly made of plastic (blocking less of the signal).

Figure 10 and Figure 11 depict the distance at which an SNR of 0.5dB can be achieved at different positions around the WSs.

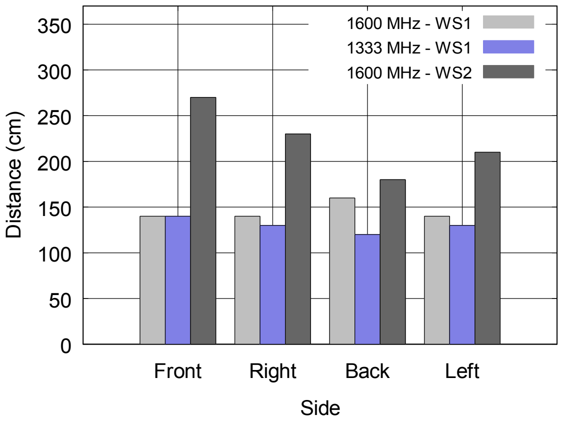


Figure 10: The distance at which an SNR of 0.5dB is achieved at various positions around the transmitters WS1 and WS2 using dual mode and different clock speeds

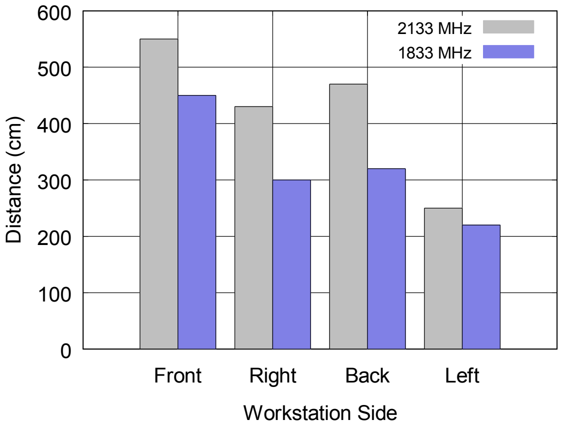


Figure 11: The distance at which at least 0.5dB of SNR is achieved at various positions around the transmitter WS3 using quad mode and different clock speeds.

7.3. Bit Rates

The GSMem receiver implemented using OsmocomBB on the nine year old mobile phone significantly limit the channel's quality. Although this device provides the advantage of GSM baseband programmability, it has limited real-time processing power and inadequate access to the DSP's full capabilities. Due to these limitations, we preferred using simple ASK type modulations over other more sophisticated options. Using the proposed B-ASK modulation with this device, we were able to receive binary data from the GSMem transmitter at a bit rate of 1 to 2 bit/s. This allows exfiltration of small amounts of information such as identifiers, passwords, and encryption keys, within several minutes. We examined the bit error rate (BER) by transmitting a set of 256-bit encryption keys from a workstation. Figure 12 depicts the BER over varying distances between the transmitting workstation and a nearby mobile phone.

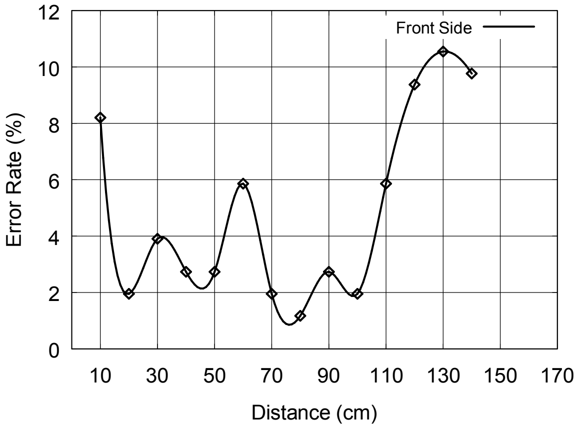


Figure 12: The Motorola C123's BER plot from a B-ASK transmission using WS1 as the transmitter.

7.4. Software Defined Radio (SDR)

Much higher bit rates - at even further distances - are achievable when more modern equipment is used and the full capabilities of the baseband component are accessible. To demonstrate this fact, we implemented a GSMem receiver using GNU-Radio software on an affordable SDR kit; the Ettus Research Universal Software Radio Peripheral (USRP) B210 [54], which is capable of capturing data at velocities up to 32 million samples per second. The USRP was connected to Lenovo ThinkPad T530 (through the USB 3.0 interface), with dedicated software suitable for capturing signals from the USRP, i.e. GNU-Radio v3.7.5.1. The OS is Linux Ubuntu 14.10 (64 bit).

Since we had full access to the DSP's capabilities, we implemented the receiver using the frequency shift keying modulation scheme (FSK) where a '1' and '0' were modulated by using two distinct frequencies. Creating two carrier waves was accomplished by adding a slight delay inside the memory transfer operation loop. Since this version of the GSMem transmitter was not implemented on a cellular device, we omit the rest of its details from the body of this paper. Using this hardware, we were able to improve the signal quality and the reception distance significantly. At a distance of 2.6m and where $T = 0.001$, we achieved a bit rate of 1000 bit/s, with a BER of approximately 0.087%. Table 6 summarizes the time needed to transfer certain pieces of sensitive information at the rates of T=0.5 (using Motorola C123) and T=0.001 (using USRP).

Data	Length (bit)	Rx Time Motorola C123	Rx Time USRP
MAC Address	48	30 sec	48 ms
Plain Password	64	40 sec	64 ms
MD5	128	1.3 sec	128 ms
GPS Coordinate	128	1.3 sec	128 ms
SHA1 Hash	160	1.6 min	160 ms
Disk Encryption Key	256	2.6 min	256 ms
RSA Private Key	2048	21.3 min	2.04 sec
Fingerprint Template	2800	29.1 min	2.8 sec

Table 6: Transmission times

In order to increase the effective distance, we used a directed printed circuit board (PCB) log periodic antenna [55], optimized for capturing signals at the range of 400 MHz – 1000 MHz. The antenna connected to the USRP via its standard connectors.

We measured the signal levels of '1' and '0' emitted from a transmitting WS3 over varying distances. The transmitter resides in a regular work space with several active desktop workstations situated within a 10m radius. As can be seen in Figure 13, the signals were received in 30 meters and beyond. This is a significant improvement when compared to the mobile phone receiver. Furthermore, these results were obtained with a rather affordable hardware receiver, using commonly available components.

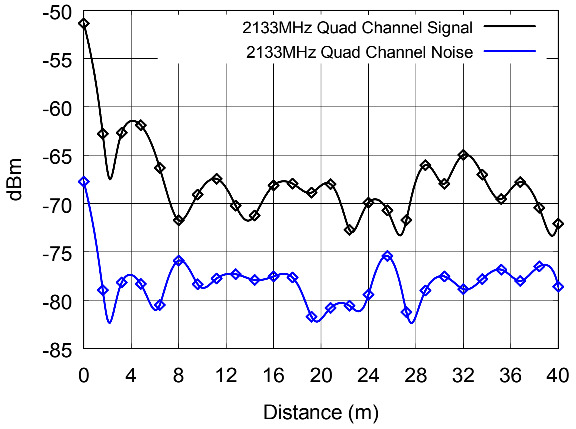


Figure 13: Signal strength received on f_{c1} as transmitted from WS3 at distances of 0-40 meters from the front side of the chassis.

8. Countermeasures

Official governmental and military standards concerning EMSEC countermeasures are mainly classified, despite some occasional leaks [16], [56]. With the exfiltration method described in this paper, the "zones" approach may be used as a countermeasure, defining spatial regions where mobile phones, including simple devices, are prohibited. As discussed earlier, however, the signal reception distance may grow when dedicated hardware receiver is being used. In this context, the insulation of partition walls may help. Structural building elements, such as reinforced concrete floors, seem to provide insulation by acting as a Faraday cage. However, enclosing each computer within a Faraday cage seems impractical. Shielding the transmitting component within the computer, i.e., the multi-channel memory bus is a challenging task, particularly when compared to shielding other emanation sources, such as monitor cables. Another defensive strategy may involve behavioral (dynamic) analysis and anomaly detection, trying to detect GSMem activities at runtime on the process level [9] [57]. However, when the baseband firmware is utilized as the GSMem receiver, it is particularly hard to detect because of the separation of the baseband component from the main operating system [49]. In this case, a meticulous forensic analysis of the device may be required.

9. Conclusion

In this paper we present GSMem, a method for exfiltrating data from air-gapped networks. Our major contributions include a unique covert channel, consisting of a feasible transmitting method, and a ubiquitous receiver that doesn’t arouse suspicion. The covert channel is based on electromagnetic waves emitted at frequency bands of GSM, UMTS and LTE cellular networks. The transmitting software exploits specific memory-related CPU instructions, utilizing the multi-channel memory bus to amplify the transmission power. Subsequently, the transmitted signals are received and demodulated by a rootkit residing at the baseband level of a cellular phone. Note that, unlike some other recent work in this field, GSMem exploits components that are virtually guaranteed to be present on any desktop/server computer and cellular phone. Furthermore, elementary cellular phones, those without Wi-Fi, camera, or other nonessential instrumentation, are often allowed into classified facilities, even in security-aware organizations. We provide essential technical background information about cellular networks and an overview of baseband components in mobile phones. Next, we discuss the design considerations of the transmitter and the receiver, regarding signal generation, data modulation, transmission detection, noise mitigation, and handling a moving receiver. Our GSMem transmission software - implemented on Windows and Linux - has a small computational footprint, which makes it hard to detect. The GSMem receiver is implemented on a mobile phone, by modifying the baseband firmware of a low-end device. We present its architecture and discuss its capabilities and limitations. We go on to evaluate the method’s using extensive configurations, settings, and various parameters. Our current results demonstrate the overall feasibility of the method, at a distance of 1-5.5 meters when using a standard cellular baseband receiver. We also evaluated the wider boundaries of GSMem using a dedicated yet affordable hardware receiver. The associated experiments yielded an effective distance of 30 meters and beyond. We believe that exposing this new covert channel will serve to raise professional awareness and academic interest.

References

[1] GReAT team, "A Fanny Equation: "I am your father, Stuxnet"," Kaspersky Labs' Global Research & Analysis Team, 17 2 2015. [Online]. Available: https://securelist.com/blog/research/68787/a-fanny-equation-i-am-your-father-stuxnet/.

[2] A. Gostev, "Agent.btz: a Source of Inspiration?," SecureList, 12 3 2014. [Online]. Available: http://securelist.com/blog/virus-watch/58551/agent-btz-a-source-of-inspiration/.

[3] N. Shachtman, "Under Worm Assault, Military Bans Disks, USB Drives," Wired, 19 11 2008. [Online]. Available: http://www.wired.com/2008/11/army-bans-usb-d/.

[4] IT@Intel White Paper, IT Best Practices, *Enabling Smart Phones in Intel's Factory,* Intel IT, 2011.

[5] L. Martin, "Important Information," Lockheed Martin, [Online]. Available: http://www.lockheedmartin.com/us/atl/maps/cherryhill/information.html. [Accessed 17 2 2015].

[6] H. Welte, "Anatomy of contemporary GSM cellphone hardware," Unpublished paper, c, 2010.

[7] M. Degrasse, "Broadcom looks to sell baseband unit," RCRWirelessNews, 2 6 2014. [Online]. Available: http://www.rcrwireless.com/20140602/wireless/broadcom-exploring-sale-baseband-unit.

[8] R. P. Weinmann, "All your baseband are belong to us," hack. lu., 2010.

[9] R. P. Weinmann, "Baseband Attacks: Remote Exploitation of Memory Corruptions in Cellular Protocol Stacks," in *WOOT*, 2012.

[10] R. P. Weinmann, "Baseband exploitation in 2013: Hexagon challenges," in *Pacsec 2013*, Tokyo, Japan, 2013.

[11] R. J. Anderson, "Emission security," in *Security Engineering, 2nd Edition*, Wiley Publishing, Inc., 2008, pp. 523-546.

[12] R. J. Aldrich, "Shootdowns, Cyphers and Spending," in *GCHQ – The Uncensored Story of Britains Most Secret Intelligence Agency*, Harper Press, 2010, pp. 201-226.

[13] W. van Eck, "Electromagnetic Radiation from Video Display Units: An Eavesdropping Risk?," *Computers and Security 4,* pp. 269-286, 1985.

[14] M. G. Kuhn and R. J. Anderson, "Soft tempest: Hidden data transmission using electromagnetic emanations," in *Information Hiding*, 1998, pp. 124--142.

[15] M. G. Kuhn, "Compromising emanations: Eavesdropping risks of computer displays," University of Cambridge, Computer Laboratory, 2003.

[16] J. McNamara, "The Complete, Unofficial TEMPEST Information Page," 1999. [Online]. Available: http://www.jammed.com/~jwa/tempest.html. [Accessed 4 10 2013].

[17] E. Thiele, "Tempest for Eliza," 2001. [Online]. Available: http://www.erikyyy.de/tempest/. [Accessed 4 10 2013].

[18] S. S. Clark, B. Ransford, A. Rahmati, S. Guineau, J. Sorber, K. Fu and W. Xu, "WattsUpDoc: Power side channels to nonintrusively discover untargeted malware on embedded medical devices," in *USENIX Workshop on Health Information Technologies (Vol. 2013)*, 2013.

[19] U. Rührmair, X. Xu, J. Sölter, A. Mahmoud, M. Majzoobi, F. Koushanfar and W. Burleson, Efficient power and timing side channels for physical unclonable functions, Springer Berlin Heidelberg, 2014.

[20] T. Halevi and N. Saxena, "A closer look at keyboard acoustic emanations: random passwords, typing styles and decoding techniques," in *ACM Symposium on Information, Computer and Communications Security*, 2012.

[21] M. Hanspach and M. Goetz, "On Covert Acoustical Mesh Networks in Air.," *Journal of Communications,* vol. 8, 2013.

[22] R. Callan, A. Zajic and M. Prvulovic, "A Practical Methodology for Measuring the Side-Channel Signal Available to the Attacker for Instruction-Level Events," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, IEEE, 2014, pp. 242-254.

[23] G. Mordechai, G. Kedma, A. Kachlon and Y. Elovici, "AirHopper: Bridging the air-gap between isolated networks and mobile phones using radio frequencies," in *Malicious and Unwanted Software: The Americas (MALWARE), 2014 9th International Conference on*, IEEE, 2014, pp. 58-67.

[24] M. Hanspach and M. Goetz, "Recent Developments in Covert Acoustical Communications.," in *Sicherheit*, 2014, pp. 243-254.

[25] M. Guri, M. Monitz, Y. Mirski and Y. Elovici, "BitWhisper: Covert Signaling Channel between Air-Gapped Computers using Thermal Manipulations," in *arXiv preprint arXiv:1503.07919*, 2015.

[26] P. John-Paul, "Mind the gap: Are air-gapped systems safe from breaches?," Symantec, 5 May 2014. [Online]. Available: http://www.symantec.com/connect/blogs/mind-gap-are-air-gapped-systems-safe-breaches.

[27] C. A., Q. Zhu, P. R. and B. T., "An impact-aware defense against Stuxnet," in *American Control*, 2013.

[28] J. Larimer, "An inside look at Stuxnet," IBM X-Force, 2010.

[29] D. Goodin, "Meet "badBIOS," the mysterious Mac and PC malware that jumps airgaps," ars technica, 31 10 2013. [Online]. Available: http://arstechnica.com/security/2013/10/meet-badbios-the-mysterious-mac-and-pc-malware-that-jumps-airgaps/.

[30] J. Zaddach, A. Kurmus, D. Balzarotti, E.-O. Blass, A. Francillon, T. Goodspeed, M. Gupta and I. Koltsidas, "Implementation and implications of a stealth hard-drive backdoor," *Proceedings of the 29th Annual Computer Security Applications Conference,* pp. 279-

288, 2013.

[31] D. Goodin and K. E. Group, "How “omnipotent” hackers tied to NSA hid for 14 years—and were found at last," ars technica, 2015.

[32] J. Linden, "DeathRing: Pre-loaded malware hits smartphones for the second time in 2014," Lookout, 4 December 2014. [Online]. Available: https://blog.lookout.com/blog/2014/12/04/deathring/.

[33] M. Kelly, "MouaBad: When your phone comes pre-loaded with malware," Lookout, 11 April 2014. [Online]. Available: https://blog.lookout.com/blog/2014/04/11/mouabad/.

[34] M. Guri, G. Kedma, A. Kachlon and Y. Elovici, "AirHopper: Bridging the air-gap between isolated networks and mobile phones using radio frequencies," in *Malicious and Unwanted Software: The Americas (MALWARE), 2014 9th International Conference on*, IEEE, 2014, pp. 58-67.

[35] RSA Research Labs, "Anatomy of an Attack," 1 4 2011. [Online]. Available: https://blogs.rsa.com/anatomy-of-an-attack/.

[36] J. Scahill and J. Begley, "THE GREAT SIM HEIST: HOW SPIES STOLE THE KEYS TO THE ENCRYPTION CASTLE," TheIntercept, 19 February 2015. [Online]. Available: https://firstlook.org/theintercept/2015/02/19/great-sim-heist/.

[37] F. Obermaier, H. Moltke, L. Poitras and J. Strozyk, "Snowden-Leaks: How Vodafone-Subsidiary Cable & Wireless Aided GCHQ’s Spying Efforts," sueddeutsche, 25 November 2014. [Online]. Available: http://international.sueddeutsche.de/post/103543418200/snowden-leaks-how-vodafone-subsidiary-cable.

[38] D. Sanger and N. Perlroth, "Bank Hackers Steal Millions via Malware," NY times, 14 February 2015. [Online]. Available: http://www.nytimes.com/2015/02/15/world/bank-hackers-steal-millions-via-malware.html?_r=0.

[39] "Digital cellular telecommunications system (Phase 2+), Radio transmission and reception 12.4.0 12," ETSI 3GPP, January 2015. [Online]. Available: http://www.etsi.org/deliver/etsi_ts/145000_145099/145005/12.04.00_60/ts_145005v120400p.pdf.

[40] Cellmapper, "Frequency Calculator," Cellmapper, [Online]. Available: https://www.cellmapper.net/arfcn.

[41] M. Shiraz, M. Whaiduzzaman and A. Gani, "A study on anatomy of smartphone," *Computer Communication \& Collaboration,* vol. 1, pp. 24-31, 2013.

[42] S. S. a. M. H. a. R. B. a. S. J. a. F. K. a. X. W. Clark, "Current events: Identifying webpages by tapping the electrical outlet," in *Computer Security--ESORICS 2013*, Springer, 2013, pp. 700-717.

[43] G. Patents, "Frequency shift keying". Patent US Patent 2,461,456, 8 February 1949.

[44] Intel, "Intel® Streaming SIMD Extensions 2 Store Intrinsics," Intel, [Online]. Available: https://software.intel.com/sites/products/documentation/doclib/iss/2013/compiler/cpp-lin/GUID-19F086CA-B0AE-4FC0-B5B5-A99AD5D62CFE.htm.

[45] AMD, "AMD64 Architecture Programmer’s Manual 128-Bit and 256-Bit XOP and FMA4 Instructions," 11 2009. [Online]. Available: http://support.amd.com/TechDocs/43479.pdf.

[46] "Intel Instruction Set Architecture Extensions - Advanced Vector Extensions," Intel, [Online]. Available: https://software.intel.com/en-us/intel-isa-extensions#pid-16007-1495.

[47] AMD, "All SIMD All the Time," AMD, September 2007. [Online]. Available: http://developer.amd.com/community/blog/2007/09/10/all-simd-all-the-time/.

[48] IBM, "Power ISA™," 3 May 2013. [Online]. Available: https://www.power.org/wp-content/uploads/2013/05/PowerISA_V2.07_PUBLIC.pdf.

[49] H. Welte, "Anatomy of contemporary GSM cellphone hardware," unpublished paper, c, 2010.

[50] SRlabs , "Turning USB peripherals into BadUSB," [Online]. Available: https://srlabs.de/badusb/.

[51] OsmocomBB, "OsmocomBB," [Online]. Available: http://bb.osmocom.org/trac/. [Accessed 13 1 2015].

[52] "Motorola C123," GSM arena, [Online]. Available: http://www.gsmarena.com/motorola_c123-2101.php.

[53] C. Turner, "New CortexTM - R Processors for LTE and 4G Mobile Baseband," ARM, 22 February 2011. [Online]. Available: http://arm.com/files/pdf/new_cortex-r_processors_for_lte_and_4g_mobile_baseband.pdf.

[54] "Ettus Research," [Online]. Available: http://www.ettus.com/.

[55] "LP0410 Antenna," Ettus Research, [Online]. Available: http://www.ettus.com/product/details/LP0410.

[56] USAF, "AFSSI 7700: Communications and information emission security," Secretary of the Air Force, 2007.

[57] M. Guri, G. Kedma, B. Zadov and Y. Elovici, "Trusted Detection of Sensitive Activities on Mobile Phones

Using Power Consumption Measurements," *Intelligence and Security Informatics Conference (JISIC), 2014 IEEE Joint,* pp. 145-151, 2014.

[58] ETSI, "Digital cellular telecommunications system: Radio subsystem link control," ETSI, July 1996. [Online]. Available: http://www.etsi.org/deliver/etsi_gts/05/0508/05.01.00_60/gsmts_0508v050100p.pdf.

Appendix A

Receiver Implementation (*Android Application Level*)

In this appendix we show how, under certain circumstances, the GSMem signals can be received by an application run on a modern Android smartphone with an untampered baseband. This technique is limited to close proximity to the transmitter (10cm).

Android Radio Interface Layer (RIL)

Android, as part of its open source framework, defines the upper software layers with respect to its hardware peripherals. In Android, the Radio Interface Layer (RIL) component interfaces between high level telephony services (*android.telephony*) and the baseband hardware. Each vendor supplies its own implementation for the RIL interface. The vendor RIL is closed source and shipped with the stock Android firmware as a shared object (.so) binary file.

Signal Demodulation

We developed a reception method which we refer as *'neighbor cell jamming'*. According to the GSM standard, mobile equipment must periodically listen to the broadcasted pilot channels of neighboring cells in order to provide service reliability [58]. Generally, the mobile must always be registered to a cell preferably the one with the best reception. These broadcasts are sent over logical channels called broadcast control channels (BCCH), which carry information such as that cell's ID and configuration. The GSM baseband component maintains a list of best neighboring cells along with their received power level (in dBm or equivalent units) and other information. Since GSMem operates at the same frequency as the neighboring BTSs, it is possible for a GSMem transmitter to affect a drop in the reception of a station that is rather far away. This jamming effect can be used as a side channel to detect the B-ASK modulation such a sudden drop in reception quality represent a '1' and otherwise a '0'.

Implementation

Android allow obtaining the neighboring cells' information from the baseband. E.g., by invoking the method *telephonyManager.getNeighboringCellInfo().*It includes the received signal strength indication (RSSI) of each neighboring cell. Our Android application repeated an algorithm similar to Algorithm 2 (Section 6) with a few modifications. It continuously sampled and stored the signal strength of the weakest cell out of the neighboring cells (the cell which our transmission will likely override). The modulation is inversed: low RSSI represents '1' (transmission occurred) and high RSSI represents '0' (no transmission). Figure 14 shows the reception of a single bit, as received by our application on the Samsung Galaxy S5 smartphone. The phone was located 10cm away from a transmitting workstation. The 'jammed' cell had signal strength of 23asu (equal to -67dBm) before it was jammed. At second 6, the GSMem at the workstation transmit '1', causing a drop in the RSSI measurement for that cell. The transmission stops at second 8.

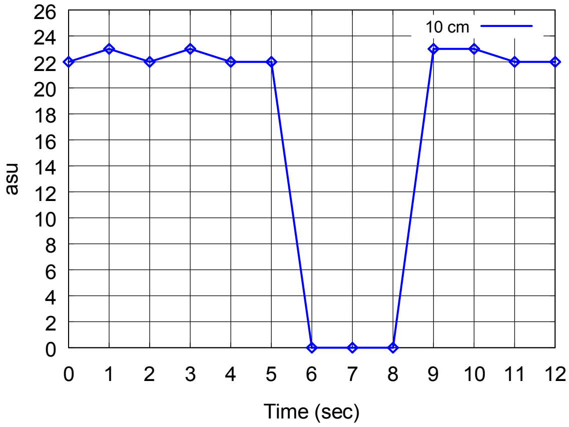


Figure 14: Neighbor cell reception level during transmission of a single bit.

Thermal Covert Channels on Multi-core Platforms

Ramya Jayaram Masti*, Devendra Rai†, Aanjhan Ranganathan*, Christian Müller†
Lothar Thiele†, Srdjan Capkun*
**Institute of Information Security, ETH Zurich*
{rmasti, raanjhan, capkuns}@inf.ethz.ch
†Computer Engineering and Networks Laboratory, ETH Zurich
{raid, thiele}@tik.ee.ethz.ch, chrismu@student.ethz.ch

Abstract

Side channels remain a challenge to information flow control and security in modern computing platforms. Resource partitioning techniques that minimise the number of shared resources among processes are often used to address this challenge. In this work, we focus on multi-core platforms and we demonstrate that even seemingly strong isolation techniques based on dedicated cores can be circumvented through the use of thermal channels. Specifically, we show that the processor core temperature can be used both as a side channel as well as a covert communication channel even when the system implements strong spatial and temporal partitioning. Our experiments on an Intel Xeon server platform demonstrate covert thermal channels that achieve up to 12.5 bps and weak thermal side channels that can detect processes executed on neighbouring cores. This work therefore shows a limitation in the isolation that can be achieved on existing multi-core systems.

1 Introduction

Covert and side channels have for a long time remained an open threat to information flow control and isolation techniques in a variety of contexts including cloud and mobile computing [50, 71, 76]. Such channels can be used for data exfiltration from a victim [17] and be exploited by colluding applications to covertly exchange information [46].

A common technique to mitigate covert and side channel attacks that leverage co-location is to dedicate resources (e.g. processor cores, memory) to individual processes for the duration of their execution. Although seemingly inefficient, such a technique is becoming viable with the appearance of multi and many-core systems which contain hundreds of cores [12]. However, it has already been shown that, due to their architecture and use, multi-core systems do not trivially protect against all types of information leakage — covert channels have been demonstrated by exploiting shared caches [35], memory bus [34], network stacks [54], virtual memory [72], I/O devices [64], etc. These covert channels, however, still exploit the resources that particular multi-core platforms, for performance and other reasons, share among the processes. The threats arising from covert and side channel attacks led to the development of mitigation techniques such as partitioning of the shared resources when possible, for example, partitioning caches [68] and bus bandwidth [31].

In this paper, we show that even strong isolation techniques based on dedicated cores and memory can be circumvented in multi-core systems through the use of a *thermal channel*. For this, we leverage the temperature information that is exposed to processes for performance reasons on multi-core platforms and two aspects of the thermal behaviour of these systems. First, the thermal capacitance and resistance of computing platforms result in remnant heat from computations, i.e. the heat is observable even after that computation has stopped. As a result, information about one process may leak to another that follows it in the execution schedule. Second, the effects of heat resulting from processes running on one core can be observed on other cores across the chip. This leaks information about a process to its peers running on other cores in a processor chip. We demonstrate our attacks on commodity multi-core systems. So far, thermal (heat) channels have not been studied on these systems. There is a trend towards exposing thermal data to users and allowing them to make thermal management decisions based on it [21]. For example, temperature information is accessible from user-space on modern Linux systems [1]. This paper highlights the tension between building thermally-efficient systems which requires exposing high-quality temperature data to applications and securing them.

In summary, we make the following contributions: *(i)* We demonstrate the feasibility of using thermal covert channels for communication between colluding applications. We measure the throughput of such a channel on an Intel Xeon server with two processors containing 8 cores each. The challenges in building such a channel include the system's thermal capacitance, effect of cooling on multi-core systems and resolution limitations of the thermal sensors available on these platforms. Although thermal covert channels have a low throughput, sensitive data such as credit-cards (16 digits) can be transmitted within 5 seconds to 4 minutes even on systems that use resource partitioning. *(ii)* We explore the factors that influence the throughput of this covert channel — processor frequency and relative locations of the colluding applications (processes) and show the throughput varies between 0.33 bps and 12.5 bps. *(iii)* We demonstrate the existence of limited thermal side channel leakage from processes running on adjacent cores that allow identification of applications based on their thermal traces. On existing systems, heat-based leakage is non-trivial to avoid without a performance penalty; we discuss possible countermeasures to eliminate or limit the impact of such attacks.

The rest of this paper is organised as follows. In Section 2, we discuss the background and motivation for our work. Section 3 discusses the thermal behaviour of x86 platforms and Section 4 describes how these properties can be exploited to create thermal channels. In Section 5, we demonstrate the feasibility of using thermal channels for covert communication even in systems with isolation based on resource partitioning. Section 6 demonstrates that limited side channel leakage can occur through thermal channels which can be exploited for unauthorised application profiling. Section 7 and Section 8 summarise countermeasures against thermal channels and related work respectively. Finally, we conclude in Section 9.

2 Background and Motivation

In this section, we summarise the use of thermal information in modern processors and resource partitioning-based isolation techniques, as well as provide a motivation for our study.

2.1 Thermal Management

Thermal management is key to the safe and reliable operation of modern computing systems. Today, thermal sensors are incorporated into a number of system components including hard-drives, DRAM, GPU, motherboards and the processor chip itself [9]. In this work, we focus on the information available from thermal sensors that are embedded in processor chips.

Ensuring the thermal stability of a processor is becoming increasingly challenging given the rising power-density in modern processor chips. As a result, major processor vendors (e.g. Intel, AMD, VIA) incorporate thermal sensors to enable real-time monitoring of processor temperature. ARM-based processors also include thermal sensors inside the system-on-chip for power and temperature management.

Initially, thermal management was done statically in hardware and included mechanisms to power-off the processor to prevent melt-downs. This later evolved to more sophisticated dynamic frequency and voltage scaling techniques that change processor frequency to lower its temperature [13, 39]. Hybrid software- and hardware-approaches to thermal monitoring have become popular over time; operating systems today poll temperature sensors and use this to manage cooling mechanisms such as processor frequency-scaling and fan-speed [2]. More recently, there is a trend towards user-centric thermal management that exposes thermal data to users and allows them to implement customised thermal management policies. For example, Linux-based systems today enable users to configure thermal policies [14, 21].

The number and topology of thermal sensors depend on the processor vendor and family. For example, while Intel and VIA processors expose temperature data for individual cores using on-die sensors, some AMD (e.g. Opteron K10 series) processors only allow monitoring the overall temperature of the entire chip using a sensor in the processor socket [1]. Optimising the number and placement of thermal sensors on processors is an active research topic [47, 53, 55].

2.2 Resource Partitioning-based Isolation

Isolation techniques for multi-core platforms that are based on resource partitioning offer a number of benefits. First, resource management approaches that rely on partitioning reduce the size of the software Trusted Computing Base (TCB). In fact, resource partitioning is gaining popularity as a means to create multiple, isolated execution environments without the need for a software TCB in servers [6] and networked embedded systems [57]. Second, the simplicity of partitioning-based resource management eases formal verification and this is leveraged by separation kernels like Muen [23]. Third, modern processors rely on partitioning techniques to build Trusted Execution Environments (TEEs). TEE technologies such as Intel Trusted Execution Technology (TXT) [40], Intel Software Guard Extensions (SGX) [52] and ARM TrustZone [15] protect the execution of security-sensitive software from a compromised operating system. Intel TXT relies on temporal partitioning of resources such as CPU and memory between trusted and untrusted software. Intel SGX and ARM TrustZone use temporal partitioning only for the CPU and implement spatial partitioning for memory resources in the system.

Resource partitioning has already been proposed as a countermeasure against covert channels [30] and side channels [68]. This is because most covert and side channels exploit shared resources. For example, a process can modify shared resources (e.g. cache, file) to communicate covertly with another colluding process. An attacker can also exfiltrate sensitive information from a victim process by tracking the state of a shared resource such as cache. Therefore, there have been proposals for minimising shared resources to reduce the number and effectiveness of covert and side channels. Examples include partitioning of caches [68] and bus-bandwidth [31].

2.3 Motivation

The main motivation behind this work is our observation that despite its security advantages, resource partitioning on multi-core systems might not be able to completely eliminate some types of inference or communication across partitions. More specifically, we want to investigate if the exposure of core temperature information could be used to build both side channels and covert communication channels between processes that execute on different cores within a multi-core system. Our goal is to study these channels primarily in terms of their feasibility and throughput.

Thermal channels are particularly interesting in the context of multi-core systems for two main reasons: *(i)* today, these platforms expose the information from thermal sensors to users and *(ii)* thermal channels can be tested for their effectiveness under the resource partitioning-based isolation mechanisms that multi-core systems can support. To build thermal channels, it is necessary to understand the type and quality of temperature data available on systems today. One must also account for the nature of temperature variations on such systems and the factors that affect them. We focus on Intel x86 platforms for our study given their wide spread use.

3 Thermal Behaviour of x86 Platforms

In this section, we present theoretical and empirical aspects of the thermal behaviour of x86 systems. More specifically, we first discuss recent attempts to analyse, model, and simulate the thermal behaviour of the state-of-the-art processors. Then, we discuss the on-die thermal sensors available on Intel processors and the thermal behaviour of these platforms under a CPU-intensive load.

3.1 Models of Thermal Behaviour

The most common abstraction of the thermal behaviour of processors is the resistor-capacitor mesh network model. This model is based on the well-known duality between thermal and electrical phenomenon [36]. Each physical layer of the processor is modelled separately as a resistor-capacitor mesh. The heat flow between the layers itself and eventually to the environment is represented by connecting the various meshes using additional resistors and capacitors. Such an approach [28] assumes that it is possible to approximate the thermal properties of a processor using a linear model. It also captures factors such as high thermal resistivity of silicon, heat-sinks and fan cooling that affect overall processor temperature.

Alternative empirical approaches that approximate thermal behaviour using measurements from on-die thermal sensors and machine learning techniques have also been explored (e.g. [60]). The advantage of such an approach over using traditional models is that it does not need information such as detailed design parameters (e.g. floorplan of the processor) which are usually not readily available.

Given the complexity of modern commodity processors and the lack of public information required to accurately model them, in this paper, we focus on a more empirical approach. We use measurements from the on-die thermal sensors to understand the thermal behaviour of commodity systems.

3.2 Temperature Sensors in Intel Processors

Intel labels each of its processors with a maximum *junction temperature* which is the highest temperature that is safe for the processor's operation. If the processor's temperature exceeds this level, permanent silicon damage may occur. To avoid such processor melt-down, Intel facilitates processor temperature monitoring by incorporating one digital thermal sensor (DTS) into each of the cores in a processor. The layout of the cores within a processor chip can be identified using `lstopo` [7] on a Linux machine. For example, on the Xeon server used in our experiments, the cores are arranged along a line (as shown in Section 5.1). Each DTS reports the difference between the core's current temperature and the maximum junction temperature [3]. The accuracy of the DTS varies across different generations of Intel processors. They typically have a resolution of $\pm 1^\circ$C.

The absolute value of a core's temperature in °C is computed in software by subtracting the thermal sensor reading from the maximum junction temperature. Thermal data from a sensor can be obtained using special CPU registers of the corresponding core. The data from

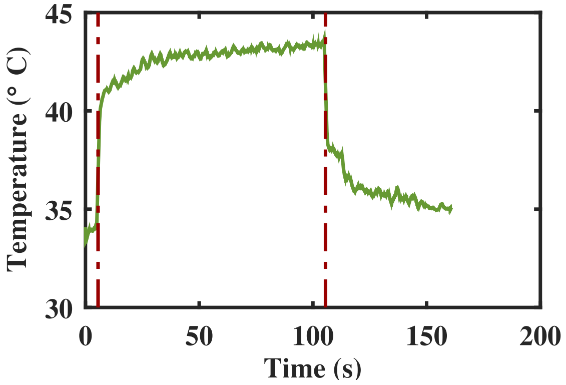


Figure 1: **Thermal Response of a CPU Intensive Application.** Temperature trace resulting from the execution of an application that does RSA decryption in a loop for 100 s. The start and end times of the application's execution are indicated using the two red lines. Temperature increases rapidly initially and saturates over time as an application runs. Similarly, it falls rapidly as soon as the core becomes idle and gradually returns to the ambient temperature.

all sensors is exposed using the `coretemp` kernel module [1] on Linux systems. This information is accessible from user space through the `sysfs` filesystem which is refreshed every 2 ms.

3.3 Example Temperature Trace

To illustrate how computations affect the temperature of a core, we ran a CPU intensive application – more specifically, one that does an RSA decryption continuously in a loop. We ran the application on core 3 of an octa-core processor (for setup details, refer to Section 5.1). Figure 1 shows the recorded temperature trace of core 3 during the execution of the application for 100 s (between the dotted red lines) on it and for about 50 s thereafter when the core cools. We observe that 25 ms after the application begins execution, the temperature rises by 5°C from approximately 35 °C to 40 °C. Following this rapid rise, the temperature increases very slowly and saturates at 43°C. As soon as the application stops executing, the temperature falls rapidly to 38 °C in about 25 ms and takes an additional 11 s to reach 35 °C.

The exponential nature of the temperature rise and fall is a result of the system's thermal capacitance and resistance. The temperature fall curve shows that the temperature changes caused by such an application's execution can be observed for sometime after it has stopped.

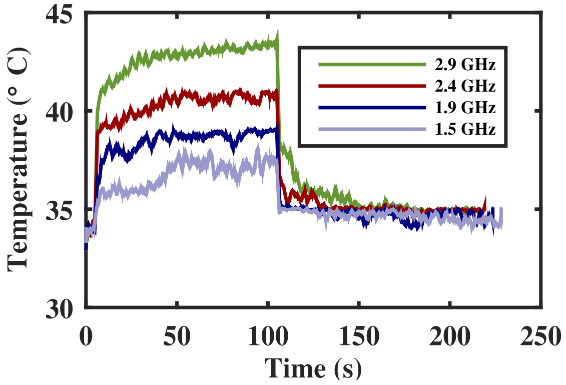


Figure 2: **Effect of Processor Frequency on Thermal Behaviour.** Temperature profiles produced by running a CPU intensive application on a core at different processor frequencies for 100 s.

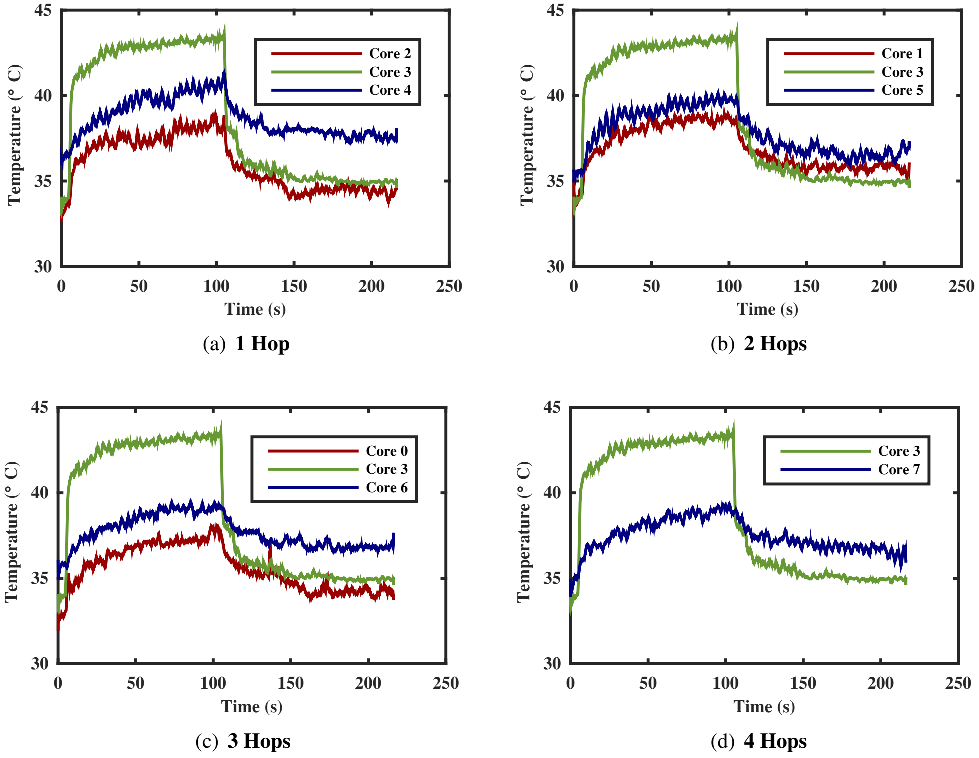
3.4 Factors Influencing a Core's Temperature

The major factors affecting the temperature at a particular core are the fan speed, processor frequency and heat propagation from neighbouring cores. Since, in our experiments, we do not control the server fan speed (see Section 5.1), we only discuss the effect of the processor frequency and heat propagation on a specific core's temperature below.

CPU Frequency. Most Intel processors are designed to run at a set of discrete frequencies for optimising power consumption. For example, in our setup (Section 5.1), the Xeon server can run at frequencies between 1.2 GHz and 2.9 GHz. All cores within a single processor chip run at the same frequency. Changes in frequency at a given core are reflected across all the other cores. The actual frequency can be controlled either by the user or by the kernel; for example, Ubuntu systems allow users to control this using the `sysctl` interface.

Figure 2 shows how the processor frequency affects temperature when a CPU-intensive application runs for 100 s. We can observe that higher frequencies result in more heat and higher saturation temperatures. This is because processor operation at a higher frequency results in a larger power density and therefore, more heat.

Heat Propagation From a Neighbour. The heat resulting from computations on one core will propagate to neighbouring cores. As a result, the temperature at a certain core depends not only on that core's workload (type of computation and schedule) but also those of its neighbours.



(a) **1 Hop**

(b) **2 Hops**

(c) **3 Hops**

(d) **4 Hops**

Figure 3: **Heat Propagation from a Neighbouring Core.** Effect of running a CPU-intensive application on core 3 of an octa-core processor for 100 s on the temperature sensors of its adjacent cores.

Figure 3 shows the effects of a CPU-intensive application executing on a central core (core 3) of an octa-core processor for 100 s. We notice that the computation on core 3 affects the temperature sensors of its neighbouring cores which remain idle all through. Additionally, we observe that the saturation temperature of a neighbouring core decreases with increasing distance from core 3. This effect is not symmetric as one would expect on either side of core 3. We suspect that this is due to an asymmetrically located processor hotspot or asymmetrically positioned thermal sensor.

4 Exploiting Thermal Behaviour

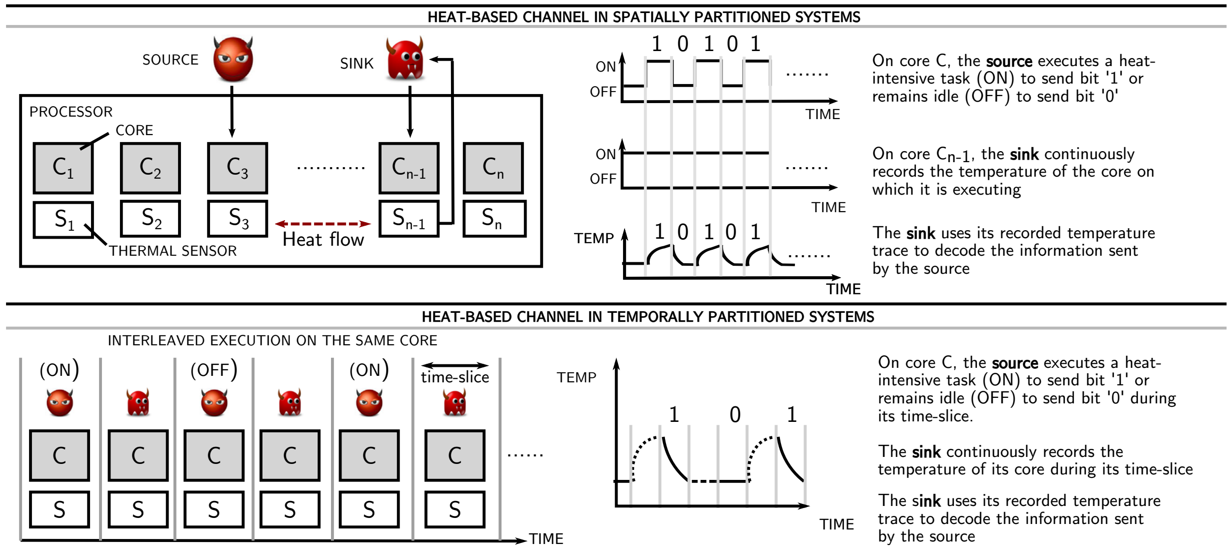
In this section, we present the intuition underlying the construction of thermal channels on multi-core systems.

4.1 Isolation based on Spatial and Temporal Partitioning

Isolation techniques that rely on resource partitioning are becoming increasingly popular and there have been a number of proposals for using such partitioning to prevent covert and side channels [31, 68]. In our work, we consider two most common types of process isolation and partitioning techniques: Spatial and Temporal as shown in Figure 4. In spatially partitioned systems, processes are isolated by being assigned exclusive computation resources, i.e. no two processes share cores or memory. Such an approach prevents certain types of side channels between processes that execute concurrently. For example, cache-partitioning prevents any information leakage that may occur based on the state of the cache-lines in a processor (e.g. how many cache-lines are full). In such systems, processes do not share any processor temperature sensors because they do not use any common CPU resources.

In temporally partitioned systems, the processes share the same resources but run in a time-multiplexed manner. For example, this technique is used by TEEs like Intel TXT in which only one of two partitions (trusted or untrusted) are active at a time but have access to common cores and memory. In systems that employ temporal partitioning, processes that share one or more cores have access to the corresponding temperature sensor(s) during their execution time-slice.

Thermal channels that leverage system thermal behaviour can be used to circumvent both these types of isolation techniques as we describe below.



On core C, the **source** executes a heat-intensive task (ON) to send bit '1' or remains idle (OFF) to send bit '0'

On core Cn-1, the **sink** continuously records the temperature of the core on which it is executing

The **sink** uses its recorded temperature trace to decode the information sent by the source

On core C, the **source** executes a heat-intensive task (ON) to send bit '1' or remains idle (OFF) to send bit '0' during its time-slice.

The **sink** continuously records the temperature of its core during its time-slice

The **sink** uses its recorded temperature trace to decode the information sent by the source

Figure 4: **Covert communication using Thermal Channels.** We demonstrate that the temperature sensors on commodity multi-core platforms can be misused for covert communication by two colluding processes in spatially and temporally isolated systems.

4.2 Constructing Thermal Channels

Based on our discussion in Section 3, we make two observations that can be used to construct thermal channels.

Remnant Heat. Since temperature variations that result from a computation can be observed even after it stops, these variations leak information regarding the computation to the process that follows it in the execution schedule especially if they share the same core. This remnant heat can be exploited as a thermal side channel and may allow a process to exfiltrate sensitive information from its predecessor thereby violating temporal partitioning. Furthermore, it can also be used for communication between two colluding processes that time-share a core. Note that while it is possible to *reset* most resources (e.g. CPU registers, caches) to prevent other types of channels before switching between applications, the remnant heat from a computation (and hence, a thermal channel) is hard to eliminate.

Heat Propagation to a Neighbouring Core. The thermal conductivity of the processor results in heat propagation between cores, i.e, the heat that results from a computation not only affects its underlying core's temperature but also its neighbouring cores. This heat flow can be exploited as a thermal channel by an attacker to make inferences about a potentially sensitive computation at a neighbouring core. Colluding processes can also use the heat flow to communicate covertly. Since it is hard to eliminate heat flows within processors, thermal channels are a viable threat even in spatially partitioned systems.

There are several challenges involved in the construction of thermal channels. First, the nature of temperature changes makes it hard to control the effect that an application's execution will have on the temperature of its own core and its neighbours. Second, the limited resolution of the temperature sensors available on current x86 platforms prevents fine-grained temperature monitoring. Finally, fan-based cooling mechanisms affect the rate and extent of temperature variations.

5 Covert Communication Using Thermal Channels

In this section, we present the feasibility and throughput of communication using thermal covert channels in spatially and temporally partitioned multi-core systems. We first describe our experimental setup that implements such isolation mechanisms. Throughout, we refer to the data sender as the *source* and the recipient as the *sink*.

5.1 Experimental Setup

Our setup is based on an Intel server containing two octa-core Xeon processor chips and running an Open SUSE installation (Figure 5). We use *cpusets* [4] to implement spatial and temporal partitioning. Using cpusets, we restrict the OS to one of the processor chips (Processor 2) and isolate it from the rest of the system. We achieve spatial partitioning by running the source and the sink on

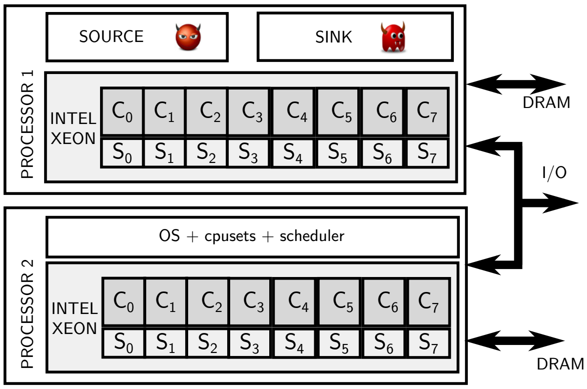

SOURCE
SINK
PROCESSOR 1
INTEL XEON
C_0 C_1 C_2 C_3 C_4 C_5 C_6 C_7
S_0 S_1 S_2 S_3 S_4 S_5 S_6 S_7
DRAM
I/O
OS + cpusets + scheduler
PROCESSOR 2
INTEL XEON
C_0 C_1 C_2 C_3 C_4 C_5 C_6 C_7
S_0 S_1 S_2 S_3 S_4 S_5 S_6 S_7
DRAM

Figure 5: **Our Experimental Setup.** Our framework consists of an Intel Xeon-based server platform running Suse Linux. We use *cpusets* to achieve spatially or temporally isolated source and sink applications. We wrote a custom source application that uses RSA decryption operations to generate heat and a sink application that records its own core's temperature continuously.

separate cores on the Processor 1 with minimal interference from the OS. To realise temporal partitioning, our system incorporates a scheduler that controls the duration and cores on which the source and sink execute.

We wrote a custom application that performs an RSA decryption (using PolarSSL [10]) continuously in a loop and we use it as the source application of the covert channel. We chose a compute intense benchmark like RSA because thermal sensors are typically located in the processor's region which is most likely to experience very high temperatures, such as the ALU [53]. Hence, this benchmark can quickly increase CPU temperature. This choice of benchmark also complies with popular thermal benchmarks (e.g. CStress [5], mprime [8]), that contain applications which extensively use the CPU register file and ALU.

We rely on the server fan for cooling the cores. Our server allows the fan-speed to be configured only through the BIOS. We set the fan-speed to the maximum allowed value (15000 rpm) for our entire study. We chose this setting because it is the most likely setting for servers which run computationally intensive tasks. Our server is currently in a room whose ambient temperature is around 22°C. We also implemented a custom sink application that records the temperature of the core on which it executes continuously.

Our experimental framework is implemented using C. It allows configuration of run-time parameters like the processor frequency, set of applications to run, their schedule and mapping to cores. Initialisation and tear down of the measurement framework is performed using a set of Perl and Bash scripts. Our setup allows us to achieve spatial and temporal isolation; this makes it an ideal platform for our investigation of thermal channels.

5.2 Covert Communication in Spatially Partitioned Systems

This section addresses the construction of thermal channels in the scenario where the source and sink applications run on dedicated cores and execute concurrently. The sink has access only to its own core's temperature sensor and not that of the source as described in the upper part of Figure 4. To communicate covertly in such a scenario, the source exploits the heat propagation from its own core to the sink that runs on a neighbouring core. In this section, we demonstrate the feasibility of achieving this on a commodity multi-core platform and evaluate the throughput of such a communication channel. Below, we first present the encoding scheme that we use for data transmission and then describe the experiments that realise covert communication using the thermal channel.

Encoding and Decoding. The source and sink use *On-Off Keying* for their communication. To send bit '1', the source application runs RSA decryption operations to generate heat and to transmit bit '0', it remains idle. It is important that the source application runs long enough to affect the sink's temperature sensor on a neighbouring core to send bit '1', i.e. it must generate enough heat to raise the temperature of the sink's core above the ambient temperature. We denote the minimum duration for which the source application needs to execute to transmit a '1' bit to the neighbouring cores as T_b. The source remains idle for the same duration to send a '0' bit. We assume that the source and sink *a priori* agree on T_b and a fixed preamble to mark the start of the data.

The sink that records the temperature of its own core continuously does the following to decode the data. It first searches the recorded temperature trace for a fixed preamble. We choose a preamble starting with bit '1' because it can clearly be identified by the sink. To detect the start of the preamble, the sink searches for a '1' bit by detecting the first rising edge, i.e. a temperature increment $\geq$ 2°C given its ambient temperature. We use this threshold because the resolution of the sensors on the platform is ±1°C. It then tries to decode the bits following this to see if they match the preamble. The source repeats this until it recovers the preamble from the temperature trace. It then decodes the remaining bits using a simple edge detection mechanism in which a rising edge indicates bit '1', a falling edge indicates bit '0' and a no-change implies that the value is the same as the previous bit.

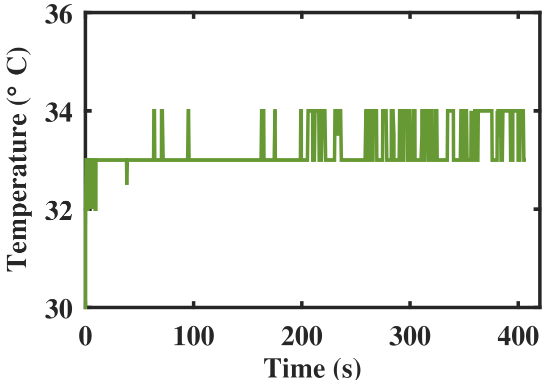


Figure 6: **Temperature Drift due to the Sink's Execution.** The temperature drift caused by the execution of the sink itself is very slow as shown here. This trace was recorded over 400 s by running the sink application on core 3 with all the other cores idle. Note that the resolution of the sensor is ±1°C.

T_b(ms)	Bit Error (%)	
	Core 2 (1-hop)	Core 1 (2-hop)
250	18	–
500	14	–
750	13	–
1000	11	24
1250	9	26
1500	8	15

Table 1: **Calibration of T_b in Spatially Partitioned Systems.** We send a block of 100 bits consisting of alternating ones and zeroes using different T_b values from the source (core 3) to the sink that runs at one and two hop distances from it. The processor frequency was set to 2.9 GHz and this table shows the resulting bit-error rates ('–' indicates that the data could not be decoded). We observe when $T_b \geq$ 500 ms, we can decode data with less than 15% error at one hop but this does not improve much by increasing T_b to 1500 ms. We also notice that the required T_b increases with greater distance from the source. At a one hop distance, setting T_b = 750 ms and using Hamming(7,4) error correction code results in a channel throughput of up to **0.33 bps**.

Temperature Drift due to the Sink's Execution. The sink relies on the source to affect its core's temperature sensor for communication. However, to do this, it is necessary to isolate any temperature drifts that may be caused by the sink's execution itself on its own temperature sensor.

To understand these drifts better, we run the sink for a long time and observe its temperature. Figure 6 shows the temperature trace of core 3 when the sink is running on it for 400 s and the other cores are idle. The core's temperature remains stable at around 33°C for 200 s and later drifts slowly towards 34°C. Therefore, we conclude that the temperature changes caused by the execution of the sink process itself is negligible over a long duration of time (e.g., 200 s).

Calibration of T_b. Before the actual transmission of data, we have to determine the optimal value of T_b, i.e. the duration for which the source executes or remains idle to send bit '1' and bit '0' respectively. Note that the actual value of T_b depends on the relative locations of the source and the sink. This is because the effect of the source's execution affects the cores farther away from it to a lesser extent (see Section 3). For our first experiments, we fix the source to execute on a core 3 because it is a central core and the sink to execute on core 2. We later describe the effects of increasing the distance between the source and the sink on T_b.

To estimate T_b, we first set it to a value between 50 ms and 1500 ms. We then attempt to send 100 data bits from the source on core 3 to the sink on core 2 and observe the resulting temperature traces on core 2. We do this by configuring the source application to be active and idle for T_b alternately. Our data consist of 50 alternating ones and zeros. We choose this data sequence because it is important to ensure that the chosen T_b consistently results in the desired temperature increment on core 2.

We were unable to decode data when T_b was smaller than 250 ms. We observe that data transmission using $T_b \geq$ 500 ms results in about 10% bit errors (Table 1). Furthermore, we notice that the bit error rate does not improve much by increasing T_b from 500 ms to 1500 ms. Figure 7 shows the temperature traces of the two cores during the data exchange using a T_b = 750 ms. The data shown here has been post-processed to remove noise using a smoothing function. We observe that the temperatures of core 3 and core 2 are well-correlated (correlation co-efficient $\simeq$ 0.55, p-value = 0).

Error Rate. To understand the nature of errors in thermal channels, we send a pseudorandom sequence of a 1000 bits in 100-bit blocks. Each block begins with a preamble to enable the sink to detect the start of data transmission.

From our initial experiments, we observe that the temperature traces of core 2 and core 3 are well-correlated in time over a sequence of alternating ones and zeros (Figure 7). Therefore, we choose a preamble of five alternating ones and zeroes (10 bits in total). The source and sink synchronise in 9 out of 10 tests and the average error rate is 13.22 % (± 5.19) for a T_b value of 500 ms. On increasing T_b to 750 ms and 1000 ms, the source and

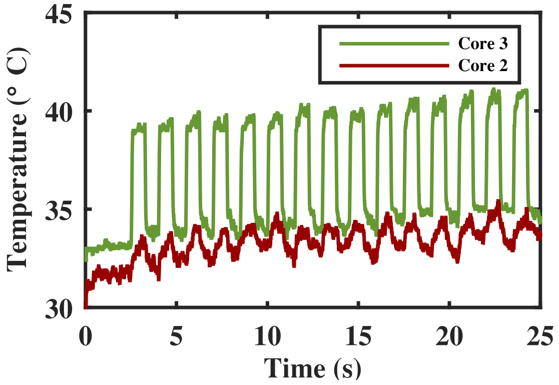


Figure 7: **Thermal Communication in Spatially Isolated Systems.** Temperature traces recorded during the transmission of the 30 bits (15 ones and 15 zeroes) from the source (core 3) to the sink (core 2) using a T_b = 750 ms.

sink synchronise over all 10 tests and the average error rate is 11.3% (± 2.83) and 11% (± 3.83) respectively.

Varying the Sink's Location. We repeat similar experiments by running the sink on core 1 and core 0 which are two and three hops away from the core 3 to see how the error rate varies with increasing distance from the source. At a two hop distance, we observe that for a given T_b, the error rate is higher than in the case of the one hop (Table 1). At a three hop distance, we were unable to decode data at 1500 ms. However, increasing the value of T_b sufficiently will allow data transmission at a 3-hop distance. For example, Figure 3(c) shows an extreme case in which T_b is set to 200 s to transmit bit '1'.

The increased error rate and deterioration in the ability to decode data is expected. This is because heat resulting from computations at a given core affects the cores closer to it more than the cores farther away. We repeated the experiments to estimate the error rate from the source (core 3) to a sink running on core 1 at a two hop distance. We observe that the source and sink synchronise successfully in 9 out of 10 tests. We can transmit data at the rate of 1 bit in 1.5 s (T_b = 1500 s) with an error rate of 18.33% (±4.21).

T_b(ms)	Bit Error (%)	
	2.9 GHz	2.4 GHz
250	18	–
500	14	23
750	13	24
1000	11	23
1250	9	14
1500	8	14

Table 2: **Effect of Processor Frequency on Required T_b.** We send a block of 100 bits consisting of alternating ones and zeroes using different T_b values from the source (core 3) to the sink (core 2). The table shows the resulting bit-error rates at different processor frequencies ('–' indicates that the data could not be decoded). We observe that when the processor runs at lower frequencies, T_b has to be increased to achieve lower bit-error rates.

Effect of Frequency on T_b. To understand the effect of processor frequency on T_b, we repeated our experiments for 1-hop communication at lower frequencies, namely, 2.4 GHz and 1.9 GHz. As shown in Table 2, for a given T_b, the error rate increases at lower processor frequencies. When the processor frequency is set to 1.9 GHz, we could not decode data even at 1500 ms. We note that using a larger value for T_b would solve this problem and can be done using the same methodology we used for our experiments. This deterioration in error rates and the ability to decode data itself is expected because lower frequencies result in lesser heat generation from a given computation. Therefore, the rise in temperature may not be significant enough to detect a bit '1' .

We repeated the data transmission experiments when the processor frequency is set to 2.4 GHz. We transmit a pseudo-random sequence of 1000 bits in 100-bit blocks. Each block is preceded by a preamble and is sent from the source (core 3) to the sink (core 2) using T_b = 1250 ms and 1500 ms. In both cases, the source and sink synchronise in all 10 tests. The observed error rates in both cases is similar, i.e. 14.9% (± 3.9) and 15.9% (± 6.08) for T_b = 1250 ms and T_b = 1500 ms respectively.

Throughput Estimation. From the above discussion, we conclude that the throughput of thermal covert channels in spatially partitioned systems depends on number of factors. This includes the time required to transmit one bit of information (T_b) and error rates. Both these parameters in turn depend on the processor frequency and the distance between the colluding processes.

At 1-hop distances, given a T_b of 750 ms, the throughput would be 1.33 bps in the ideal case without any errors. However, due to the 11% errors that we observe in the experiments, actual communication would require error correction to be implemented. When we analysed the nature of the errors, we found that for every four bits, with a probability of over 0.9, there was one or no errors. Therefore, we could use a Hamming (7,4) error-correction code to correct for these errors. This would result in 75% overhead and hence, an effective throughput of 0.33 bps. When the frequency is changed to 2.4 GHz, the throughput is about 0.2 bps using a Hamming (7,4) error correction code. A similar trend was observed on

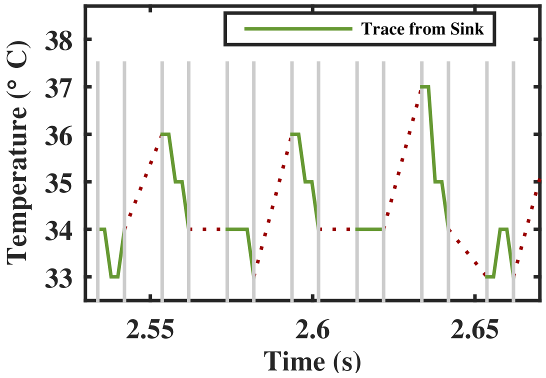


Figure 8: **Thermal Communication in Temporally Isolated Systems.** Temperature traces recorded during the transmission of the 6 bits (3 ones and 3 zeroes) from the source to the sink. The source and the sink execute in alternate time-slots of 10 ms (marked in grey) on core 3. The thick lines are the actual temperature traces recorded by the sink and the dotted lines represent the temperature changes that occur as a result of the source's execution.

increasing the distance between the source and sink. Although the thermal channel's throughput is low, it still allows the transfer of sensitive data like credit-card information (16 digit) in a few minutes. We discuss other factors that affect the thermal channel's throughput in Section 5.4.

5.3 Covert Communication in Temporally Partitioned Systems

Temporal partitioning schemes securely multiplex the same resources (e.g. cores, memory) between several applications. Systems that use this technique mitigate information leakage through side channels by clearing caches, registers, etc. while switching between processes. However, the thermal footprint of an application (the source) remains intact for observation by the other application that executes after it on the same core (the sink). This is a result of the thermal capacitance and resistance of processors and can be exploited to communicate covertly as our experiments demonstrate.

Scheduling, Encoding and Decoding Schemes. In temporally partitioned systems, a scheduler determines the order in which different partitions execute on a core. Therefore, we implement a scheduler (Figure 5) that realises this functionality. Since the sink and source share the same core, they run in an interleaved manner and the sink has access to the temperature sensors only during its execution time-slice (t_s). Note that t_s is controlled by the system's scheduler while T_b is controlled by the applications. If the execution time-slice (t_s) $\leq T_b$, then communication becomes difficult because the source cannot generate enough heat to transmit '1' bits. However, if $t_s \geq T_b$, then the source can choose to execute for long enough to cause a temperature change that the sink notices. Note that temperature variations over the course of the source's execution within one time-slice are not visible to the sink; instead the sink only has access to the final temperature after the source's execution time-slice. Therefore, in our implementation, the source sends a single bit per time-slice (using the temperature at the end of that time-slice) to the sink over a thermal covert channel, i.e. $T_b = t_s$. We use the same On-Off keying technique as before in Section 5.2.

T_b(ms)	Bit Error (%)		
	2.9 GHz	2.4 GHz	1.9 GHz
10	0	4	0
15	0	1	1
20	0	0	1
25	0	0	0
30	0	0	0

Table 3: **Effect of Processor Frequency on Required T_b.** We send a block of 100 bits consisting of alternating ones and zeroes using different T_b values from the source (core 3) to the sink that runs on the same core. The table shows the resulting bit-error rates at different processor frequencies. We observe that the error rates do not change much even at lower processor frequencies. Setting the frequency to 2.9 GHz, T_b to 10 ms and using Hamming(7,4) error correction code leads to the channel throughput of up to **12.5 bps**.

Calibration of T_b. We consider the scenario in which the sink and the source share a core and run in a round-robin fashion. The source heats up the processor (bit '1') or stays idle (bit '0') to send one bit of information to the sink application that runs immediately after.

In order to understand how fast one can transmit bits over such a channel, we do the following. We try to send an alternating sequence of 50 ones and 50 zeroes from the source to the sink. We vary T_b between 10 ms which is the minimum value that our framework allows and 30 ms. The source and sink run on core 3 in our experiments. Figure 8 shows an example temperature trace that the sink records during its execution. Note that the sink has access to the shared core's temperature sensor only during its own time-slice. We observed no errors in the data that the sink decodes for $T_b \geq 10$ ms and therefore, use this value for further experiments. We also repeated the experiment on the cores at the corners (core 0 and core 7) and noticed similar results.

Error Rate. To understand the nature of errors in this channel, we send a pseudorandom stream of 1000 bits in 100-bit blocks using T_b = 10 ms. We send 10 bits for synchronisation at the beginning of every block similar to the experiments in Section 5.2. The synchronisation using the preamble was successful in all cases and the data transmission resulted in error rates of 7.6% (± 1.9), 9.5% (± 4.86) and 7.1% (± 2.23) for experiments on cores 0, core 3 and core 7 respectively.

Effect of processor frequency on T_b. We repeated our experiments at two lower processor frequencies to understand how it may affect the required T_b for reliable communication. In the T_b calibration experiments, the error rates remain low despite the decrease in frequency (Table 3). In the actual data transmission tests (of 1000 bits in 100 bit blocks) at 2.4 GHz, the source and the sink successfully synchronise in all 10 tests and the error rate is about 6.5% (±3.58). At 1.9 GHz however, the synchronisation succeeds only 5 out of 10 times and the error rate is 9.5% (±2.55). This indicates that a higher T_b value is required for more reliable communication at this lower processor frequency.

Throughput Estimation. The throughput of the thermal channel in temporally partitioned systems depends on the execution schedule of the applications and the time required to send one bit of information ($T_b = t_s$). We note that typical Linux systems have a time-slice of about 100 ms which is 10 times bigger than the one we need for implementing thermal covert channels.

When T_b is 10 ms, we would expect the throughput of the thermal channel would be 50 bps. However, since the communication is error prone and results in up to 10% error, the encoding scheme would have to incorporate error correction codes. On analysing the nature of the errors during the transmission of a 1000 bits, we see that with a probability of over 0.9, there is 1 or no errors in every four data bits. Therefore, we can use a Hamming(7,4) code to overcome these errors and this results in an effective throughput of about 12.5 bps. This throughput is independent of which particular core the source and sink share (core 0/3/7). A T_b of 10 ms results in low error rates even at a processor frequency of 2.4 GHz and hence, the throughput is roughly 12.5 bps. We note that this data rate would allow the transmission of sensitive information such as credit-card details (16 digits) in about 5 s.

5.4 Other Factors Affecting Throughput

We have explored how factors like processor frequency and relative locations of the source and sink affect the throughput of the thermal covert channel. Below we discuss additional parameters that affect the throughput. An exhaustive evaluation of these factors is beyond the scope of the paper and is intended as part of future work.

Noise from Other Workloads. On a given system, the throughput of a thermal communication channel will depend on the actual workloads running on that platform. In the case of thermal channels in spatially partitioned systems, the exact effect of a concurrent workload on the throughput will likely depend on the nature of the workload and its relative distance from the sink's core. On the one hand, a workload that saturates its own core's temperature is only likely to increase the sink's temperature by a constant amount without disturbing the actual communication patterns. On the other hand, a workload that runs in the opposite schedule as the source (i.e. it is active whenever the source is idle and vice-versa) is likely to result in increased errors at the sink's core. Analogous discussions hold in the case of temporal channels. Finally, if the attacker controls more than one core on the platform, then he could potentially generate more heat and build faster channels but this requires further exploration.

Other Encoding and Error-Correction Schemes. In our experiments, we used the On-Off keying technique to transmit data. Instead, to improve throughput, one could borrow techniques from signal processing and telecommunications such as multi-level encoding. One could alternatively use bi-phase encoding schemes such as Manchester-coding that would lower the data rate (e.g. by half) but also result in fewer errors. Furthermore, in order to detect and correct more than single-bit errors, one could implement alternative error correction schemes such as Reed-Solomon [62] or BCH [18]. For example, a Reed-Solomon RS(32,28) code encodes a 28-word data into a 32-word codeword and is capable of correcting errors up to 2-words in length.

We note that our experiments were performed in conditions that minimise any noise that may arise from other concurrent workloads such as the OS. Given this and the low resolution of the thermal sensors (±1°C), we believe that an order of magnitude improvement in the throughput is unlikely.

6 Thermal Channels for Unauthorised Profiling

In this section, we present a preliminary study of how thermal side channels enable unauthorised thermal profiling of processes even in systems that implement strong isolation mechanisms like spatial resource partitioning. In contrast to thermal covert channels in which the source and the sink collude to exchange data, thermal side channels allow an attacker to exfiltrate information from a *victim* without requiring any cooperation from the victim.

The heat generated from an application (which we refer to as the victim) can be observed from its neighbours. This may leak information regarding the nature of its computation to processes at other cores. More specifically, if the attacker has reference thermal traces for victim applications, he can recognise if and when such an application executes on a neighbouring partition. For example, identifying that a sensitive or potentially vulnerable application is running on a neighbouring core may aid an attacker in preparing or launching an attack. Application identification based on temperature traces has not been addressed previously in literature. Below we present a first study that tries to understand its effectiveness as an attack vector.

Goal and Intuition of the Attack. We assume that an attacker has access to a reference thermal trace of the victim application (say RSA decryption). Such a trace can be obtained by the attacker if he has access to a similar platform as the one he is attacking. The attacker's goal is to verify if the temperature trace of his core is a result of the victim application's execution on a neighbouring core. Note that the attacker does not have access to the temperature trace of its neighbouring core(s) but only to that of his own core. For simplicity, we assume that only the attacker and the victim are active during the attack and that they are collocated on adjacent cores. The attacker continuously monitors his own core's temperature and then, correlates it with a reference trace of the victim application. A strong correlation indicates that the attacker's temperature trace was a result of the execution of the victim application with high probability.

Experiments and Analysis. We chose a set of five CPU-intensive applications including RSA decryption and four applications from a benchmark suite, MiBench [32] (ADPCM, Quick Sort, BitCount, BasicMath) and use them as the universal set of applications that a victim core (core 3) executes. We intentionally chose similar applications all of which stress the ALU and register file region of the core. This choice makes our task harder than distinguishing between applications with very distinct thermal behaviour, for example, an idle application vs. a thermal benchmark. A deeper exploration of the thermal behaviour of different classes of applications (memory intensive, I/O intensive, etc.) and their distinguishability is out of the scope of this work.

To understand the feasibility of identifying these applications, we ran each of them for 200 s on core 3 of our setup (Section 5.1) and collected the temperature traces of a neighbouring core (core 2). We repeated this five times for each of the applications and Figure 9 shows one such trace for each of them. We observe that the saturation temperature for the RSA decryption application is higher than the rest.

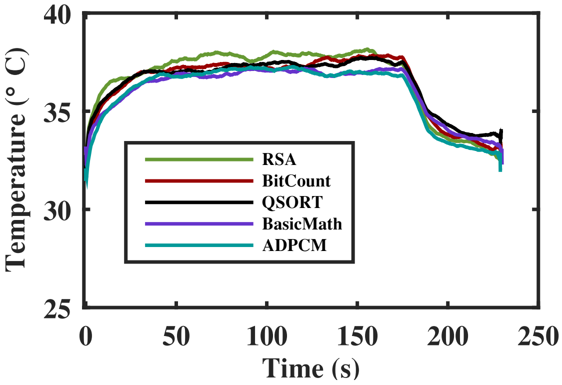


Figure 9: **Example Temperature Traces of Different Applications.** We used a set of five CPU intensive applications (RSA decryption, BitCount, QSort, BasicMath, ADPCM) from a popular MiBench benchmark suite [32]. We ran each application for 200 s separately and recorded the resulting temperature traces on a neighbouring core.

We use simple correlation as a metric to measure similarity/differences between pairs of applications. We first correlated the traces from the RSA application. Since there are 5 runs, we have 10 pairs to correlate. We observe that the correlation is higher than 85% in seven out of ten occasions. Using this same correlation threshold of 85% also results in 28% false positives when the RSA application is correlated with the others from the benchmark suite. In general, traces belonging to the same application have high correlation values ($\geq$ 80%). However, traces belonging to different applications also show high correlation because they are all CPU intensive and stress similar parts of the CPU ($\geq$ 75%). Therefore, we conclude that using a simple correlation metric would only allow distinguishing applications that behave very differently. More sensitive metrics (such as thermal models [60] or machine-learning based classifiers) are required for better accuracy in other cases.

Finally, more fine-grained data exfiltration such as deducing AES or RSA keys on commodity x86-systems using the thermal side channel is an open, unexplored problem. A key challenge is the limited resolution of the temperature sensors which is ±1°C and the rate at which the sensors are refreshed (currently, once every 2 ms).

7 Discussion

In this section, we present possible countermeasures against thermal channels and discuss their limitations. We also discuss a potential security application for thermal channels.

Countermeasures. Since we leveraged the temperature information exposed to software to construct thermal channels, a natural solution to this problem would be to restrict access to temperature sensors on the system. However, such information cannot always remain hidden. For example, centralised control and monitoring of thermal states does not scale well with the advent of many-core processors [41]. Such processors contain hundreds of cores and host a large number of autonomous processes on separate cores. In fact, research prototypes like Intel's SCC platform [41] already allow subsets of cores to administer their frequency and voltage independently for power-efficiency; temperature information is a vital input to this decision process. The software at each core should track thermal information to schedule intelligently and to detect if any of its threads are misbehaving. *Therefore, there is a tension between securing platforms and improving their energy-efficiency by exposing thermal data to software applications.*

Even if one restricts access to the temperature sensors, related parameters may still leak information about the system's thermal state. Examples of such parameters are clock skew, fan speed and even processor frequency in systems that allow dynamic frequency scaling. Since all these parameters are usually common across cores or subsets of cores within a processor chip, they can still provide a signalling mechanism. Finally, while it may be possible to separate processes temporally and spatially to limit the effectiveness of thermal channels, such resource allocation strategies are wasteful and result in low resource utilisation.

Thermal Fingerprinting For Security. So far we discussed only how thermal behaviour of systems can be exploited by attackers. The same properties could be used for achieving better security. Since temperature changes resulting from computations are difficult to avoid, we hypothesise that thermal profiling techniques can also be used to detect any anomalous behaviour in the execution of an application. More specifically, it is highly likely that run-time compromise of an application results in a temperature trace that does not match its original *thermal fingerprint*. It has been shown to be possible to extract thermal models by monitoring the application under controlled conditions [60, 61]. By comparing the actual execution trace to the expected trace, it may be possible to detect run-time compromise of software applications.

More generally, understanding the capacity of thermal channels using information theory will help assess the throughput of covert communication. Similarly, a theoretical estimation of the entropy of such channels will help bound fingerprint accuracy and hence, side channel leakage. A more detailed study along these lines is an interesting direction for future work.

8 Related Work

We review previous work on covert and side channels on x86 systems and on thermal channels in general. We also provide examples of existing literature on optimising computing systems for thermal efficiency because it highlights the advantages of exposing thermal data as opposed to the other work that misuses this data to undermine security.

Thermal Channels and Attacks. There is no previous work that demonstrates the feasibility of thermal covert and side channels on commodity multi-core systems as we do in this paper. Previously, two works discussed and one implemented thermal covert channels on FPGA boards [19, 38, 51]. There have also been attempts to transmit data between two processes by changing fan-speed [20]. The ability to remotely monitor a system's clock-skew (influenced by the changes in the system temperature) has also been exploited in the past for exposing anonymous servers [56, 75] and covert communication with a remote entity [74]. We note that although some of these works [20, 74] use the term thermal channel, none of them use the thermal information available on modern systems to covertly communicate between processes on the same host as we do in this paper.

More recently, it has been shown that it is possible to use temperature variations to induce processor faults [59] which can in turn be also be used to extract sensitive information like RSA keys [37]. Thermal information can also be used for coarse-grained data-exfiltration. For example, since temperature directly reflects the intensity of computation, it can be used to estimate the load or resource utilisation of a machine. This was illustrated by Liu et al. who computed the resource utilisation of servers in Amazon's EC2 using the temperature data that is exposed to virtual machines [49].

Previous research has identified other security risks that arise from hardware and software thermal management techniques on modern systems. For example, malicious processes may cause a denial-of-service by slowing down the processor [33] or permanently damaging hardware by causing thermal hotspots [27]. Such processes could exploit the fact that certain architecture components (e.g. instruction cache) are ignored by thermal optimisation approaches on processors [45].

Covert and Side Channels on x86-Systems. Originally defined by Lampson as part of the confinement problem [46], today, several covert channels have been identified and explored in the context of x86 systems. Covert channels can be classified either as timing or storage channels. Timing covert channels transmit information in terms of the timing of certain events. Examples of timing covert channels include cache-based and memory bus-based channels both of which were first demonstrated by Hu [34, 35]. Wang et al. identify more timing channels that arise out of processor extensions such as multi-threading and speculative execution [68]. Wu et al. achieve improved data rates on bus-based channels [71]. Covert timing channels that use other types of shared resources like virtual memory deduplication [72] and input devices such as keyboards [64] have also been studied.

In contrast to timing channels, storage channels rely on the source writing the data (indirectly) into a shared resource which the sink reads at a later point in time. Lampson's file system based covert channel [46] and covert channels that exploit CPU registers (e.g. FPU registers that signal exceptions) [65] are examples of such channels. Interestingly, certain types of covert channels such as those based on the hard-disk [70] and processor cache [22, 35] can be used to realise timing and storage channels. We could classify the thermal covert channels as storage channels because they use the CPU registers to (indirectly) exchange information.

While covert channels rely on two colluding entities for data exfiltration, side channels can be used to extract information from a unsuspecting victim without any co-operation from it. Side channels can be classified as access-driven channels, trace-driven channels and timing-driven channels. Access-driven side channels rely on a victim's modifications to a shared resource (e.g. cache) to extract sensitive information (e.g. AES keys [76]). Trace-driven channels require measuring a certain aspect of the system such as power (e.g. [43]) or electromagnetic emanations (e.g. [29]) continuously as the victim executes. Finally, timing-based side channels measure the time consumed by sensitive operations (e.g. cryptographic functions) to extract information (e.g. such as AES keys [17]). In general, side channels are used for cryptanalysis. Example attacks include extraction of AES keys [11, 17, 42, 76], DES keys [58] and RSA keys [22, 43, 44]. They can also be used to extract more coarse-grained information such as co-residency [63], existence of files [67], etc. The thermal side channel can be viewed as a trace-driven side channel that continuously tracks temperature to identify the computation at a neighbouring core.

One way to mitigate timing channels (both covert and side channels) in general is to expose less accurate timing information [34]. This technique is unlikely to be effective against thermal channels because they do not exploit timing information. Another general approach against side and covert channels is to partition system resources. This is for example used to mitigate cache-based channels [68] and bus-based channels [31]. However, such partitioning techniques will not eliminate temperature-based channels completely as demonstrated in this paper.

Thermal Monitoring of Processors. Temperature management of computing systems has gained importance over the last few years due to the increasing on-chip temperatures of modern processors. This has resulted in efforts to design and implement better thermal management techniques for processors. Examples include optimisation of sensor placement (e.g. [53, 55]), improving algorithms for dynamic temperature management (e.g. [73]) and cooling techniques [25]. There are also ongoing efforts to develop frameworks to thermally profile applications [61], build temperature-aware schedulers [24, 26] and micro-architectures [48, 66]. Thermal profiling can further be used to detect compromised process in embedded systems [69] and design schedulers such that they do not leak information through thermal fingerprints of applications [16].

9 Conclusion

In this paper, we demonstrated the feasibility and potential of thermal channels on commodity multi-core systems. We showed that such channels can be built by exploiting the thermal behaviour of current platforms. Thermal channels can be used to circumvent strong isolation guarantees provided by temporal and spatial partitioning techniques. Our experiments indicate that it is possible to use them for covert communication between processes and achieve a throughput of up to 12.5 bps. We also demonstrated that thermal channels can be exploited to profile applications running on a neighbouring core. Our work points to a limitation in the isolation guarantees that resource partitioning techniques can provide.

Attacks based on thermal channels are further facilitated by the increasing trend towards exposing system temperature information to users. This would enable users to make thermal management decisions for efficient system operation. This paper highlights the tension between designing systems to support user-centric thermal management for efficiency and security.

10 Acknowledgments

This work was carried out as a part of the SAFURE project, funded by the European Union's Horizon 2020 research and innovation program under grant agreement number 644080. This work was partially supported by the Zurich Information Security and Privacy Center. It represents the views of the authors.

References

[1] CoreTemp. http://www.alcpu.com/CoreTemp/.

[2] CPU frequency and voltage scaling code in the Linux(TM) kernel. https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt.

[3] CPU Monitoring With DTS/PECI. http://www.intel.com/content/www/us/en/embedded/testing-and-validation/cpu-monitoring-dts-peci-paper.html.

[4] CPU Sets. http://man7.org/linux/man-pages/man7/cpuset.7.html.

[5] CPUBurn-in. http://manpages.ubuntu.com/manpages/precise/man1/cpuburn.1.html.

[6] Hitachi Embedded Virtualisation Technology. http://www.hitachi-america.us/supportingdocs/forbus/ssg/pdfs/Hitachi_Datasheet_Virtage_3D_10-30-08.pdf.

[7] lstopo. http://manpages.ubuntu.com/manpages/oneiric/man1/lstopo.1.html.

[8] Mersenne Prime Search. http://www.mersenne.org/.

[9] Open Hardware Monitor. http://openhardwaremonitor.org/.

[10] SSL Library PolarSSL. https://polarssl.org.

[11] ACIIÇMEZ, O., SCHINDLER, W., AND KOÇ, C. Cache based remote timing attack on the aes. In *Topics in Cryptology — CT-RSA*. 2007.

[12] ADAPTEVA. Ephiphany Multicore IP. http://www.adapteva.com/products/epiphany-ip/epiphany-architecture-ip/.

[13] ADVANCED MICRO DEVICES. Cool and Quiet Technology Installation Guide for AMD Athlon 64 Processor Based Systems. http://www.amd.com/Documents/Cool_N_Quiet_Installation_Guide3.pdf.

[14] AMIT DANIEL, VINCENT GUITTOT, R. L. A Simplified Thermal Framework for ARM Platforms.

[15] ARM. Building a Secure System using TrustZone Technology. http://www.arm.com, 2009.

[16] BAO, C., AND SRIVASTAVA, A. A Secure Algorithm for Task Scheduling Against Side-channel Attacks. In *Workshop on Trustworthy Embedded Devices* (2014), TrustED '14.

[17] BERNSTEIN, D. J. Cache-timing Attacks on AES. http://palms.ee.princeton.edu/system/files/Cache-timing+attacks+on+AES.pdf, 2005.

[18] BOSE, R. C., AND RAY-CHAUDHURI, D. K. On a Class of Error Correcting Binary Group Codes. *Information and control*.

[19] BROUCHIER, J., DABBOUS, N., KEAN, T., MARSH, C., AND NACCACHE, D. Thermocommunication, 2009.

[20] BROUCHIER, J., KEAN, T., MARSH, C., AND NACCACHE, D. Temperature Attacks. *Security Privacy, IEEE* (2009).

[21] BROWN, L., AND SESHADRI, H. Cool Hand Linux* Handheld Thermal Extensions. In *Linux Symposium* (2007).

[22] BRUMLEY, D., AND BONEH, D. Remote timing attacks are practical. In *Proceedings of the USENIX Security Symposium* (2003), USENIX-SS '03.

[23] BUERKI, R., AND RUEEGSEGGER, A.-K. Muen-an x86/64 separation kernel for high assurance. http://muen.codelabs.ch/muen-report.pdf, 2013.

[24] CHOI, J., CHER, C.-Y., FRANKE, H., HAMANN, H., WEGER, A., AND BOSE, P. Thermally-Aware Task Scheduling at the System Software Level. In *Symposium on Low Power Electronics and Design* (2007), ISLPED '07.

[25] CHU, R. C., SIMONS, R. E., ELLSWORTH, M. J., SCHMIDT, R. R., AND COZZOLINO, V. Review of Cooling Technologies for Computer Products. *IEEE Transactions on Device and Materials Reliability* (2004).

[26] COSKUN, A. K., ROSING, T. S., WHISNANT, K. A., AND GROSS, K. C. Static and Dynamic Temperature-aware Scheduling for Multiprocessor SoCs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2008).

[27] DADVAR, P., AND SKADRON, K. Potential Thermal Security Risks. In *Semiconductor Thermal Measurement and Management Symposium* (2005).

[28] DONALD, J., AND MARTONOSI, M. Techniques for Multicore Thermal Management: Classification and New Exploration. In *International Symposium on Computer Architecture* (2006), ISCA '06.

[29] GANDOLFI, K., MOURTEL, C., AND OLIVIER, F. Electromagnetic analysis: Concrete results. In *Cryptographic Hardware and Embedded Systems*, CHES '01. 2001.

[30] GLIGOR, V. A guide to understanding covert channel analysis of trusted systems. Tech. rep., DTIC Document, 1993.

[31] GUNDU, A., SREEKUMAR, G., SHAFIEE, A., PUGSLEY, S., JAIN, H., BALASUBRAMONIAN, R., AND TIWARI, M. Memory Bandwidth Reservation in the Cloud to Avoid Information Leakage in the Memory Controller. In *Workshop on Hardware and Architectural Support for Security and Privacy* (2014), HASP '14.

[32] GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. MiBench: A free, commercially representative embedded benchmark suite. In *Workshop on Workload Characterization* (2001), WWC '01.

[33] HASAN, J., JALOTE, A., VIJAYKUMAR, T., AND BRODLEY, C. Heat stroke: Power-density-based Denial of Service in SMT. In *Symposium on High-Performance Computer Architecture* (2005), HPCA'05.

[34] HU, W.-M. Reducing Timing Channels With Fuzzy Time. In *Research in Security and Privacy* (1991).

[35] HU, W.-M. Lattice Scheduling and Covert Channels. In *Research in Security and Privacy* (1992).

[36] HUANG, W., GHOSH, S., VELUSAMY, S., SANKARANARAYANAN, K., SKADRON, K., AND STAN, M. HotSpot: A Compact Thermal Modeling Methodology for Early-stage VLSI Design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2006).

[37] HUTTER, M., AND SCHMIDT, J.-M. The Temperature Side Channel and Heating Fault Attacks. *IACR Cryptology ePrint Archive* (2014).

[38] IAKYMCHUK, T., NIKODEM, M., AND KEPA, K. Temperature-based Covert Channel in FPGA Systems. In *Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)* (2011).

[39] INTEL CORPORATION. Intel SpeedStep FAQ. http://www.intel.com/support/processors/sb/CS-032349.htm?wapkw=intel+speedstep.

[40] INTEL CORPORATION. Intel Trusted Execution Technology Measured Launched Environment Programming Guide. http://www.intel.eu/content/www/eu/en/software-developers/intel-txt-software-development-guide.html.

[41] Intel Corporation. SCC External Architecture Specification. https://communities.intel.com/servlet/JiveServlet/previewBody/5852-102-1-9012/SCC_EAS.pdf.

[42] Irazoqui, G., Inci, M., Eisenbarth, T., and Sunar, B. Wait a Minute! A Fast, Cross-VM Attack on AES. In *Research in Attacks, Intrusions and Defenses* (2014), RAID'14.

[43] Kocher, P., Jaffe, J., and Jun, B. Differential power analysis. In *Advances in Cryptology* (1999), CRYPTO '99.

[44] Kocher, P. C. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology* (1996), CRYPTO'96.

[45] Kong, J., John, J., Chung, E.-Y., Chung, S. W., and Hu, J. On the Thermal Attack in Instruction Caches. *IEEE Transactions on Dependable and Secure Computing* (2010).

[46] Lampson, B. W. A Note on the Confinement Problem. *Commun. ACM* (1973).

[47] Lee, K.-J., Skadron, K., and Huang, W. Analytical Model for Sensor Placement on Microprocessors. In *International Conference on Computer Design: VLSI in Computers and Processors* (2005), ICCD '05.

[48] Lim, C. H., Daasch, W. R., and Cai, G. A Thermal-Aware Superscalar Microprocessor. In *International Symposium on Quality Electronic Design* (2002).

[49] Liu, H. A Measurement Study of Server Utilization in Public Clouds. In *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing* (2011), DASC '11.

[50] Marforio, C., Ritzdorf, H., Francillon, A., and Capkun, S. Analysis of the Communication Between Colluding Applications on Modern Smartphones. In *Computer Security Applications Conference* (2012), ACSAC '12.

[51] Marsh, C., and McLaren, D. Poster: Temperature side channels. In *Workshop on Cryptographic Hardware and Embedded Systems* (2007), CHES'07.

[52] McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C. V., Shafi, H., Shanbhogue, V., and Savagaonkar, U. R. Innovative Instructions and Software Model for Isolated Execution. In *Workshop on Hardware and Architectural Support for Security and Privacy* (2013), HASP'13.

[53] Memik, S. O., Mukherjee, R., Ni, M., and Long, J. Optimizing Thermal Sensor Allocation for Microprocessors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2008).

[54] Mileva, A., and Panajotov, B. Covert channels in TCP/IP protocol stack - extended version. *Central European Journal of Computer Science* (2014).

[55] Mukherjee, R., and Memik, S. O. Systematic Temperature Sensor Allocation and Placement for Microprocessors. In *Design Automation Conference* (2006), DAC '06.

[56] Murdoch, S. J. Hot or Not: Revealing Hidden Services by Their Clock Skew. In *Computer and Communications Security* (2006), CCS '06.

[57] Noorman, J., Agten, P., Daniels, W., Strackx, R., Van Herrewege, A., Huygens, C., Preneel, B., Verbauwhede, I., and Piessens, F. Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base. In *USENIX Conference on Security* (2013), USENIX-SS '13.

[58] Page, D. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. https://eprint.iacr.org/2002/169.pdf.

[59] Rahimi, A., Benini, L., and Gupta, R. Analysis of Instruction-Level Vulnerability to Dynamic Voltage and Temperature Variations. In *Design, Automation Test in Europe Conference Exhibition* (2012), DATE'12.

[60] Rai, D., and Thiele, L. A Calibration Based Thermal Modeling Technique for Complex Multicore Systems. In *Design, Automation Test in Europe Conference Exhibition* (2015), DATE'15.

[61] Rai, D., Yang, H., Bacivarov, I., and Thiele, L. Power Agnostic Technique for Efficient Temperature Estimation of Multicore Embedded Systems. In *Compilers, Architectures and Synthesis for Embedded Systems* (2012), CASES '12.

[62] Reed, I. S., and Solomon, G. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics* (1960).

[63] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. Hey, you, get off of my cloud: Exploring Information Leakage in Third-party Compute Clouds. In *Computer and Communications Security* (2009), CCS'09.

[64] Shah, G., Molina, A., and Blaze, M. Keyboards and Covert Channels. In *USENIX Security Symposium* (2006), USENIX-SS'06.

[65] Sibert, O., Porras, P., and Lindell, R. An Analysis of the Intel 80x86 Security Architecture and Implementations. *Software Engineering, IEEE Transactions on* (1996).

[66] Skadron, K., Stan, M. R., Huang, W., Velusamy, S., Sankaranarayanan, K., and Tarjan, D. Temperature-aware Microarchitecture. *SIGARCH Comput. Archit. News* (2003).

[67] Suzaki, K., Iijima, K., Yagi, T., and Artho, C. Memory Deduplication As a Threat to the Guest OS. In *European Workshop on System Security* (2011), EUROSEC '11.

[68] Wang, Z., and Lee, R. Covert and Side Channels Due to Processor Architecture. In *Computer Security Applications Conference* (2006), ACSAC '06.

[69] Wolf, T., Mao, S., Kumar, D., Datta, B., Burleson, W., and Gogniat, G. Collaborative Monitors for Embedded System Security. https://hal.archives-ouvertes.fr/hal-00089605, 2006.

[70] Wray, J. An Analysis of Covert Timing Channels. In *Research in Security and Privacy* (1991).

[71] Wu, Z., Xu, Z., and Wang, H. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *USENIX Security Symposium* (2012), Usenix-SS '12.

[72] Xiao, J., Xu, Z., Huang, H., and Wang, H. Security Implications of Memory Deduplication in a Virtualized Environment. In *Dependable Systems and Networks* (2013), DSN'13.

[73] Yeo, I., Liu, C. C., and Kim, E. J. Predictive Dynamic Thermal Management for Multicore Systems. In *Design Automation Conference* (2008), DAC '08.

[74] Zander, S., Branch, P., and Armitage, G. Capacity of Temperature-Based Covert Channels. *Communications Letters, IEEE* (2011).

[75] Zander, S., and Murdoch, S. J. An Improved Clock-skew Measurement Technique for Revealing Hidden Services. In *USENIX Security Symposium* (2008), USENIX-SS'08.

[76] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Computer and Communications Security* (2012), CCS '12.

Rocking Drones with Intentional Sound Noise on Gyroscopic Sensors

Yunmok Son, Hocheol Shin, Dongkwan Kim, Youngseok Park, Juhwan Noh, Kibum Choi,
Jungwoo Choi, and Yongdae Kim
Korea Advanced Institute of Science and Technology (KAIST),
Daejeon, Republic of Korea
{*yunmok00, h.c.shin514, dkay, raccoon7, juwhan, kibumchoi, khepera, yongdaek*}*@kaist.ac.kr*

Abstract

Sensing and actuation systems contain sensors to observe the environment and actuators to influence it. However, these sensors can be tricked by maliciously fabricated physical properties. In this paper, we investigated whether an adversary could incapacitate drones equipped with Micro-Electro-Mechanical Systems (MEMS) gyroscopes using intentional sound noise. While MEMS gyroscopes are known to have resonant frequencies that degrade their accuracy, it is not known whether this property can be exploited maliciously to disrupt the operation of drones.

We first tested 15 kinds of MEMS gyroscopes against sound noise and discovered the resonant frequencies of seven MEMS gyroscopes by scanning the frequencies under 30 kHz using a consumer-grade speaker. The standard deviation of the resonant output from those gyroscopes was dozens of times larger than that of the normal output. After analyzing a target drone's flight control system, we performed real-world experiments and a software simulation to verify the effect of the crafted gyroscope output. Our real-world experiments showed that in all 20 trials, one of two target drones equipped with vulnerable gyroscopes lost control and crashed shortly after we started our attack. A few interesting applications and countermeasures are discussed at the conclusion of this paper.

1 Introduction

Sensors are devices that detect physical properties in nature and convert them to quantitative values for actuators and control systems. In many sensing and actuation systems, actuations are determined on the basis of information from sensors. However, these systems can malfunction because of physical quantities that sensors fail to measure or measure insensitively. Furthermore, most sensors cannot distinguish between normal and abnormal physical properties. Therefore, sensors can measure malicious inputs that are intentionally crafted by an attacker in addition to the physical stimuli that the sensors should detect. Because providing detection capabilities for attacks against sensors increases production costs, most commercial devices with sensors are not equipped with any ability to detect or protect against such attacks.

Recently, many sensor-equipped devices, such as smartphones, wearable healthcare devices, and drones, have been released to make the devices easier and more convenient to use. In particular, commercial and open-source drones have been widely used for aerial photography, distribution delivery [2, 3], and private hobbies. These drones have multiple sensors, such as gyroscopes, accelerometers, and barometers. A gyroscope measures changes in tilt, orientation, and rotation based on angular momentum. It is thus a core sensor for flight attitude control and position balancing.

To make the flight control modules of drones small, lightweight, and inexpensive, Micro-Electro-Mechanical Systems (MEMS) gyroscopes are used. MEMS gyroscopes are designed as Integrated Circuit (IC) packages. Each design has a unique mechanical structure in the IC package. Depending on the structure of the MEMS gyroscope, resonance occurs as a result of sound noise at resonant frequencies [37, 38, 39, 49]. This resonance causes performance degradation of the gyroscope.

The resonant frequencies of MEMS gyroscopes are usually designed to be higher than the audible frequency band to prevent malfunctioning of the sensing and actuation systems. However, in our experiments, we discovered that some MEMS gyroscopes that are popularly used in commercial drones resonate at audible frequencies as well as ultrasonic frequencies. Our experiments were designed and conducted to analyze how drones are affected by this phenomenon from an adversary point of view. The flight control software of our target drone was also analyzed to examine the propagation of this phenomenon through the whole system. The results of our real-world experiments and a software simulation show that this phenomenon could be exploited to launch incapacitating attacks against commercial drones.

The contributions of this research to the field can be summarized as follows:

- We found, using a consumer-grade speaker, that the resonant frequencies of several popular MEMS gyroscopes are not only in the ultrasonic frequency band but also in the audible frequency band, and we analyzed their resonant output.

- We investigated the effect of the resonant output of MEMS gyroscopes on the flight control of drones via software analysis and simulations.

- We developed a novel approach to attacking drones equipped with vulnerable MEMS gyroscopes using intentional sound noise, and we demonstrated the consequences of our attack in real-world experiments [1]

This paper is organized as follows: Section 2 outlines security research to date on sensor systems. Section 3 provides background information on drone systems and MEMS gyroscopes. Section 4 describes the analyses and experiments conducted in this study to investigate the effects of sound noise on MEMS gyroscopes. Analysis of the flight control software, real-world experiments, and simulations for attacking drones are described in Section 5. A discussion of the results and conclusion drawn from the results are presented in Sections 6 and 7, respectively.

2 Related Work

The security of sensors recently started to draw attention with the introduction of consumer-grade sensing and actuation systems. As this study was focused on input spoofing attacks on gyroscopes, we review in this section previous researches on 1) privacy issues related to gyroscopes, 2) resonant frequencies of gyroscopes, 3) security analyses of commercial drones, and 4) input spoofing attacks on sensing circuitry.

Privacy Issues Related to Gyroscopes: Embedded devices can be used to record the private information of users without their recognition. Because a gyroscope can be used to measure changes in tilt, orientation, and rotation, it can be used to steal a smartphone user's keystroke information, such as unlock passwords, banking passwords, and credit card numbers [36, 47]. By exploiting the capability of the gyroscopes of smartphones to measure acoustic vibrations at a low frequency band, a new attack was proposed to eavesdrop speech [59]. The focus of these studies differed from that of this paper in that they examined the use of gyroscopes to extract private information, without affecting actuation.

Resonant Frequencies of Gyroscopes: Resonant frequency has been identified as a problem that causes the performance degradation of MEMS gyroscopes. In general, the vibrating structures of MEMS gyroscopes have resonant frequencies. Resonance can occur as a result of sound noise [37, 38, 39]. Some mechanisms for mitigating interference from sound have been proposed. Roth suggested a simple and cheap defense technique that involves surrounding the gyroscope with foam [49]. Soobramaney proposed the use of an additional structure in a gyroscope that responds only to the resonant frequency to cancel out the resonant output from the gyroscope [52]. Using an additional feedback capacitor connected to the sensing electrode, the resonant frequency and the magnitude of the resonance effect can be tuned [35, 43]. It is widely believed that most consumer-grade MEMS gyroscopes have resonant frequencies. However, these resonant frequencies are often considered to be commercial secrets or are designed to be just higher than the audible frequency range.

Security Analysis of Commercial Drones: There were a couple of works on hacking commercial drones. Samland et. al. showed that *AR.Drone* [5] was vulnerable to network attacks due to unencrypted Wireless LAN (WLAN) communication and the lack of authentication for Telnet and FTP [50]. Kamkar showed that a drone can be hijacked by another drone using similar vulnerabilities [44]. Attacks such as these are focused on hijacking network connections or system privileges.

Input Spoofing Attacks on Sensing Circuitry: All sensing and actuation systems have sensing circuitry that is composed of the sensor itself and a wire that connects the sensor to other components of the system. Kune et. al. showed that an adversary can inject an Electro Magnetic Interference (EMI) signal into the wire connecting an analog sensor and Analog-to-Digital Converter (ADC) to fake a sensing signal [45]. By injecting fake waveforms, the researchers were able to inhibit pacing or induce defibrillation shocks in Cardiac Implantable Electrical Devices (CIEDs). Without affecting the sensor itself, they were able to spoof the sensing signal by injecting an EMI signal into the sensing circuitry.

It is also possible to affect the sensor itself. For example, biometric imaging sensors have frequently been targeted in sensor spoofing attacks. Tsutomu et al. showed that a verification rate of more than 68 % could be achieved against 11 different fingerprint systems using artificial fingers [46]. Galbally et al. fabricated fake

[1] A demo video of our attack against the target drone in the real world is available at `https://sites.google.com/site/rockingdrone/`.

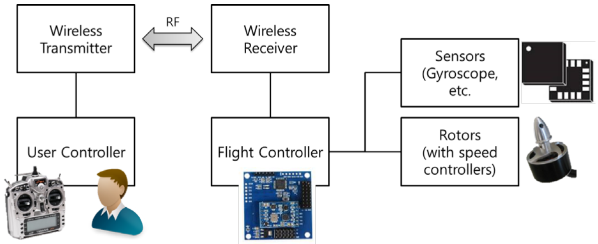


Figure 1: Block diagram of a typical drone system

fingerprints from standard minutiae templates, and more than 70 % of the fake fingerprints were accepted by the system tested [42]. In addition, a method for bypassing the user authentication of facial cognitive biometric systems was proposed as an example of sensor input spoofing against the imaging sensor systems of commercial laptops [40].

We were able to find only one notable and relevant study not related to biometric image sensors. Shoukry et al. injected magnetic fields to spoof the wheel speed of vehicles by placing a magnetic actuator near the Anti-lock Braking System (ABS) wheel speed sensor of which is also a magnetic sensor also [51]. In other words, the researchers used the same physical property as that intended to be sensed through the sensing channel of the target sensor for their spoofing attack. This work is similar to ours in that it explored intentional interference with sensors to cause malfunctioning of actuators However, we investigated whether intentional sound noise at the resonant frequency of a gyroscope can incapacitate a drone. This means that our attack is an interference attack through a channel other than the sensing channel that has to be insensitive for the gyroscope. Note that a MEMS gyroscope is *the most basic sensor used in maintaining a drone in an upright position* without any external torque.

3 Background

In this section, we explain the operation and characteristics of the drone considered in this study, its flight control system and a MEMS gyroscope.

3.1 Drone (Multicopter)

A drone is a kind of Unmanned Aerial Vehicle (UAV). Drones are used not only for military purposes but also for various non-military purposes such as delivery services, aerial photography, search and rescue (S&R), crop-dusting, and hobbies. Because of accessibility reasons, military drones were not considered in this paper. Many commercial drones have been released in recent years as the non-military drone market has grown [2, 3]. Both finished drones and DIY drones with open-source drone projects for the flight control software are commercially available. *AR.Drone* [5] is a popular commercial finished drone product. *Multiwii* [24] and *ArduPilot* [7] are open-source flight control software used widely with both DIY and commercial drones. These drones are also known as multicopters (quadcoptors if they have four rotors) because they usually have multiple rotors.

Typically, a drone system consists of multiple rotors, one flight controller, one wireless receiver, and one wireless transmitter (remote controller). Figure 1 shows a block diagram of a drone system. The flight controller receives control signals from the wireless transmitter through the receiver, and manipulates the speed of the rotors in accordance with the user's control supported by the flight controller.

3.2 Flight Attitude Control

It is very important for the drone flight controller to adjust each rotor's speed for horizontally leveling off in the air, because multiple rotors are not always exactly the same and the center of mass cannot always be ensured. To stabilize a drone's balance automatically, a flight attitude control system is implemented in the flight control software. This flight attitude control system computes the proper control signal for multiple rotors with algorithms based on the data from Inertial Measurement Units (IMUs), including gyroscopes.

IMUs, which consist of sets of sensors, are fundamental components of flight control systems for aircraft, spacecraft, and UAVs, including drones. An IMU measures the orientation, rotation, and acceleration of a drone, using a combination of a gyroscope and an accelerometer, and in some cases also a magnetometer and a Global Positioning System (GPS) [55]. MEMS gyroscopes are thus necessary components of drones and must be robust to control drones successfully.

In the case of open-source flight control software [7, 24], the most common algorithm for flight attitude control is Proportional-Integral-Derivative (PID) control. The PID control algorithm is a control loop feedback mechanism that minimizes the difference between the desired control and the current status. It is made up of three terms: the proportional, the integral, and the derivative terms, denoted by *P*, *I*, and *D*, respectively. The *P* term applies control to the system in proportion to the difference (error) between the current state and the desired state to the system. The *I* term is used to reduce the steady-state error through proportional control of the accumulation of past errors. The *D* term is used to reduce overshoot and increase stability through proportional control of the changing rate of errors. Each term has a gain (G_P, G_I, and G_D) for tuning the control system, and users can change each gain for stability and sensitivity of drones of various types, sizes, and weights.

3.3 MEMS Gyroscope

3.3.1 Operation

The principle underlying the MEMS gyroscope [1, 9] is the law of physics known as the Coriolis effect or Coriolis force. The Coriolis effect is the deflection of a moving object in a rotating reference frame. This effect appears only to an observer in the same rotating reference frame. In the observer's view, the path of the moving object is observed to be bent by a fictitious force, i.e. the Coriolis force. In other words, when an object is moving in a rotating container or package, the path of the moving object is bent in a direction different from the moving direction. Therefore, the observer on the container or package can sense this bending. Figure 2 illustrates the concept of a MEMS gyroscope structure for one axis. To sense motion with respect to one axis such as Z-axis rotation, there is a mechanical structure called a sensing mass in a MEMS gyroscope. While a sensing mass is continuously vibrating at a certain frequency with respect to the X-axis, the Coriolis force is applied in the Y-axis direction as a result of the Z-axis rotation. The amount of rotation is proportional to the amount of bending.

Figure 3 shows an example of a MEMS gyroscope structure with three axes. This gyroscope is manufactured by *STMicroelectronics* [10]. In Figure 3, M1 through M4 correspond to continuous vibrations of the sensing masses. Bending occurs in the direction orthogonal to both the vibrating axis and the rotating axis when this structure rotates with respect to each axis [10].

MEMS gyroscopes support digital interfaces such as Inter-Integrated Circuits (I^2Cs) and Serial Peripheral Interfaces (SPIs) that communicate with the processors of application systems. By reading registers of the gyroscopes that contain the sensed values, a system's processor can calculate the amount of rotation that occurs. The maximum sampling frequency for reading the registers of the MEMS gyroscopes varies from a few hundred to a few thousand samples per second. This means that gyroscopes cannot sense and recover correctly from fast changes in rotation over a few kHz without additional signal processing, according to the sampling theorem. The sampling theorem defines the minimum sampling frequency as a frequency higher than $2 \times B$ Hz when the given signal contains no frequency components higher than B Hz. If this condition is not satisfied, distortion occurs in the frequency response. This is referred to as aliasing. Because of the aliasing problem, a frequency analysis of the gyroscope output is not very useful.

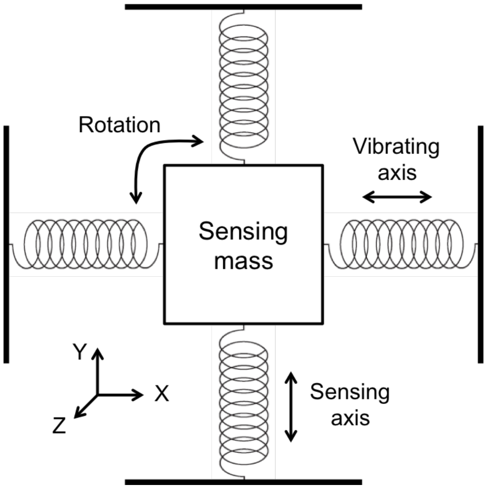


Figure 2: Concept of MEMS gyroscope structure for one axis

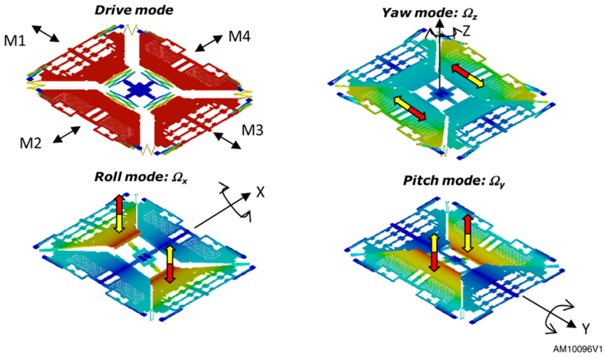


Figure 3: Operation of a three-axis MEMS gyroscope [10] (the X-, Y-, and Z-axes are defined as the pitch, roll, and yaw, respectively.)

3.3.2 Acoustic Noise Effect

The accuracy degradation of MEMS gyroscopes by harsh acoustic noise is well known to researchers who have studied the performance of MEMS sensors [37, 38, 39, 49]. A MEMS gyroscope has a resonant frequency that is related to the physical characteristics of its structure, and high-amplitude acoustic noise at the resonant frequency can produce resonance in the MEMS structure. As a result of this resonance, the MEMS gyroscope generates an unexpected output that may cause the related systems to malfunction. To minimize the resonance effect of acoustic noise in daily life, MEMS gyroscopes are typically designed with resonant frequencies above the audible frequency limit (i.e., above 20 kHz).

However, we found that some MEMS gyroscopes have resonant frequencies in both the audible and ultrasonic frequency ranges, and these sensors generate ghost outputs with injected sound noise by an attacker. In addi-

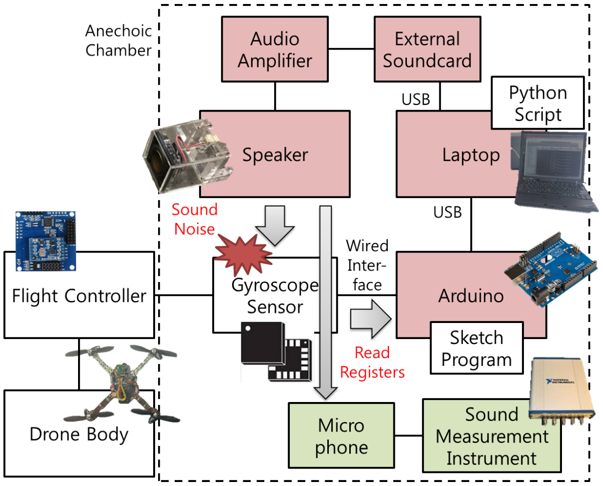


Figure 4: Overview of our experiment

tion, these MEMS gyroscopes are widely used in drone flight controllers and smartphones. The accuracy degradation problem of MEMS gyroscopes has only been considered in the context of performance issues, but this phenomenon can be used as a new attack vector. Therefore, it is important to study this phenomenon as a vulnerability that can cause critical loss of control of MEMS gyroscope application systems, such as drones.

4 Analysis of Sound Noise Effects

To explore the effects of sound noise on drones, it is necessary to identify the resonant frequencies of MEMS gyroscopes used for drones precisely. However, the datasheets of some MEMS gyroscopes do not include information on their exact resonant frequencies, and the resonant frequencies are even classified in some cases. A simple and reliable way to find the resonant frequency of a MEMS gyroscope is exhaustive search, i.e., scanning with pure single-tone sound over a chosen frequency band. In this section, the measurement and analysis of the effect of sound noise on MEMS gyroscopes are described.

4.1 Overview

An overview of our experiment is shown in Figure 4. Python scripts to generate sound noise with a single frequency and to collect data from the target gyroscopes are run on a laptop computer. A consumer-grade speaker connected to the laptop is used as the noise source and is set 10 cm above the top of the target gyroscope. We used Arduino [6], a programmable microprocessor board, to read and write registers of the target sensors. A single-tone sound noise scanning the sound frequency range was maintained until 1,000 samples had been collected from the target gyroscopes. We generated single-tone noises at frequencies from 100 Hz to 30 kHz at intervals of 100 Hz. In other words, this experiment was performed using not only audible noise (below 20 kHz) but also ultrasonic noise (above 20 kHz).

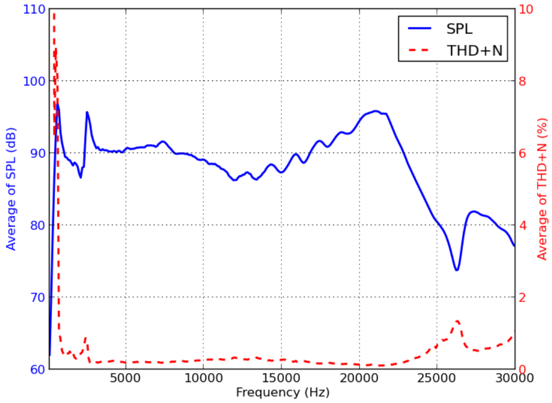


Figure 5: SPL and THD+N measurement using sound measurement instrument (National Instruments USB-4431)

We evaluated 15 kinds of MEMS gyroscopes manufactured by four vendors, which are readily available on online websites. Most of the target gyroscopes were from *STMicroelectronics* and *InvenSense*, two leading vendors of MEMS gyroscopes [22]. Each kind of gyroscope requires a different application circuit and register configuration for proper operation. We therefore implemented simple application circuits and Arduino codes for the target gyroscopes by referring to their datasheets. The effects produced on each gyroscope by sound noise were measured in an anechoic chamber (indicated by the dotted line box in Figure 4).

4.2 Sound Source

We considered the loudness and linearity of the sound source to select a sound source for further analysis.

A common noise measurement unit for the loudness of sound is the Sound Pressure Level (SPL), because sound is a pressure wave in a medium such as air or water. To show the noise level generated by our sound source [12], a consumer-grade speaker, SPL values were measured with no weighting using a professional sound measurement instrument [26] and a microphone [8]. The speaker was placed 10 cm from the microphone, and single-tone noises were generated from 100 Hz to 30 kHz at intervals of 100 Hz. We used an audio amplifier to make the sound noise sufficiently loud. In addition, we set the sampling

Table 1: Summary of experiment results (investigation of the resonant frequencies of MEMS gyroscopes using intentional sound noise)

Sensor	**Vender∗**	**Axis**	**Inter-face**	**Resonant freq. in the datasheet (axis)**	**Resonant freq. in the experiment (axis)**
L3G4200D†	STM	X, Y, Z	Digital	no information	7,900 ∼ 8,300 Hz (X, Y, Z)
L3GD20†	STM	X, Y, Z	Digital	no information	19,700 ∼ 20,400Hz (X, Y, Z)
LSM330	STM	X, Y, Z	Digital	no information	19,900 ∼ 20,000 Hz (X, Y, Z)
LPR5150AL	STM	X, Y	Analog	no information	not found in our experiments
LPY503AL	STM	X, Z	Analog	no information	not found in our experiments
MPU3050	IS	X, Y, Z	Digital	33 ± 3 kHz (X) 30 ± 3 kHz (Y) 27 ± 3 kHz (Z)	not found in our experiments
MPU6000†	IS	X, Y, Z	Digital		26,200 ∼ 27,400 Hz (Z)
MPU6050	IS	X, Y, Z	Digital		25,800 ∼ 27,700 Hz (Z)
MPU6500	IS	X, Y, Z	Digital	27 ± 2 kHz (X, Y, Z)	26,500 ∼ 27,900 Hz (X, Y, Z)
MPU9150	IS	X, Y, Z	Digital	33 ± 3 kHz (X) 30 ± 3 kHz (Y) 27 ± 3 kHz (Z)	27,400 ∼ 28,600 Hz (Z)
IMU3000	IS	X, Y, Z	Digital		not found in our experiments
ITG3200	IS	X, Y, Z	Digital		not found in our experiments
IXZ650	IS	X, Z	Analog	24 ± 4 kHz (X), 30 ± 4 kHz (Z)	not found in our experiments
ADXRS610	AD	Z	Analog	14.5 ± 2.5 kHz	not found in our experiments
ENC-03MB	Murata	X	Analog	no information	not found in our experiments

∗ *STM: STMicroelectronics, IS: InvenSense, AD: Analog Devices*

† *12 sample chips for experiments (2 sample chips for others)*

Table 2: Effect of sound noise (standard deviations and their ratios for vulnerable gyroscopes, averaged for all sample chips)

Sensor	**Without noise**			**With noise**			**Ratio**		
	$\sigma_{X_{wo}}$	$\sigma_{Y_{wo}}$	$\sigma_{Z_{wo}}$	σ_{X_w}	σ_{Y_w}	σ_{Z_w}	$\sigma_{X_w}/\sigma_{X_{wo}}$	$\sigma_{Y_w}/\sigma_{Y_{wo}}$	$\sigma_{Z_w}/\sigma_{Z_{wo}}$
L3G4200D	3.15	2.69	2.88	12.1	22.04	4.45	**3.84**	**8.21**	**1.55**
L3GD20	2.92	2.47	2.3	62.03	76.67	3.09	**21.21**	**31.04**	**1.35**
LSM330	13.09	16.03	21.45	177.71	114.34	30.44	**13.57**	**7.13**	**1.42**
MPU6000	11.79	13.92	12.8	12.48	14.74	111.21	1.06	1.06	**8.69**
MPU6050	13.21	12.32	11.17	13.8	12.55	58.17	1.04	1.02	**5.21**
MPU6500	17.34	19.63	18.21	363.21	71.04	56.15	**20.95**	**3.62**	**3.08**
MPU9150	10.69	11.47	10.71	10.98	11.97	58.59	1.03	1.04	**5.47**

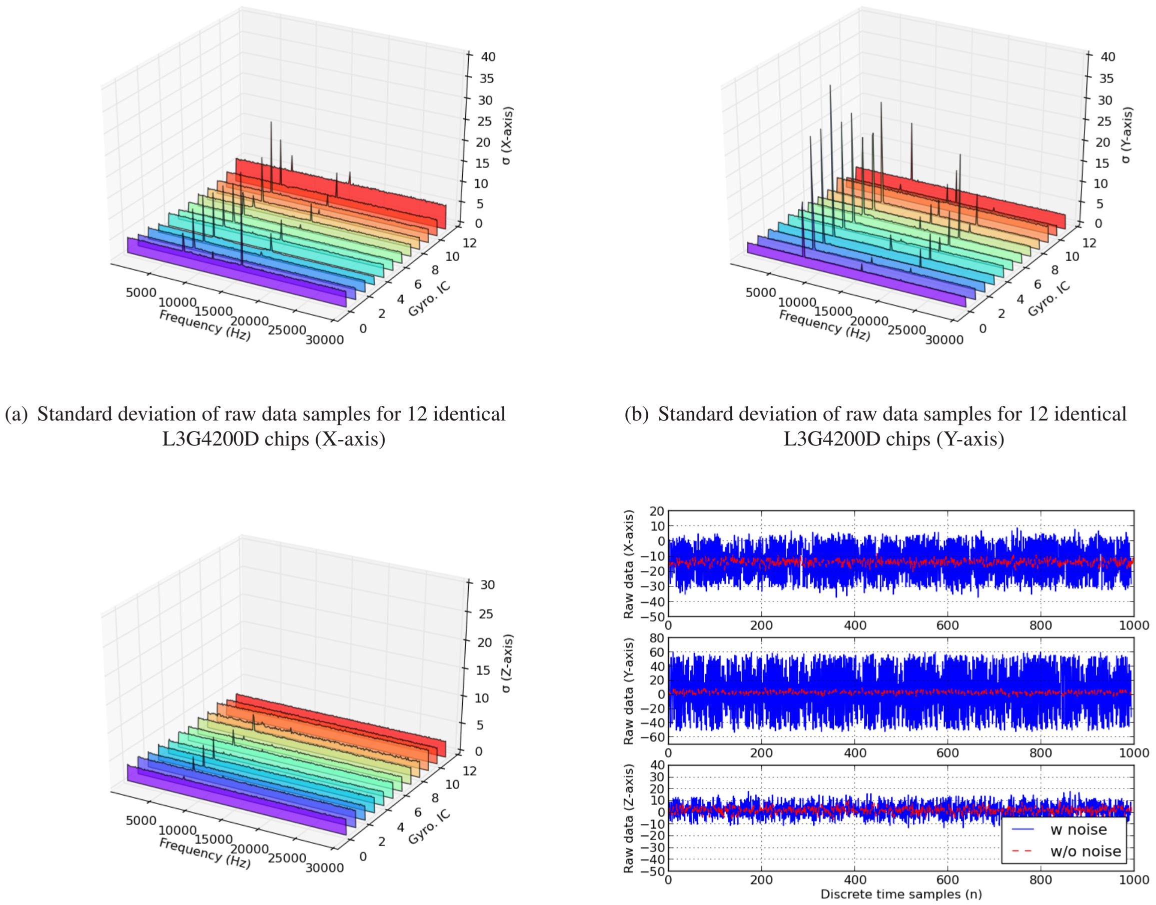
rate of the sound source to 96 kHz rather than 48 kHz to remove aliasing of the generating sound signal.

Another important property of a sound source is Total Harmonic Distortion plus Noise (THD+N), which is the ratio of the power of the harmonics and noise components to that of a fundamental component, expressed as a percentage. Every speaker has a nonlinear characteristic to its frequency response. This nonlinearity leads to harmonic distortions and noise of output sound at frequencies that are different from a fundamental frequency. If the power of these harmonics and noise is high (i.e., high THD+N), it is hard to regard the identified response as the effect from a single frequency. However, it is not necessary for low THD+N of the sound source to attack.

Figure 5 shows the average values of both the SPL and THD+N for all of the experiments. In most frequency bands, the SPL values were above 80 dB and the THD+N values were less than 2 %. Because the sound source we used was a tweeter that is usually used for high-frequency sound, the performance was not good in the low-frequency region (below 1 kHz). It is usually difficult to hear sound noise at frequencies above approximately 15 kHz, although we set the maximum volume at those frequencies. The measured SPL in our experiment was equivalent to the noise level (around 90 dB SPL) of a hand drill, hair dryer, heavy city traffic, noisy factory, and subway in the real world.

4.3 Effect of Sound Noise

Raw data samples from the registers of the target gyroscopes were collected for use in this analysis. The target gyroscopes were fixed on a stable frame in an ane-



(a) Standard deviation of raw data samples for 12 identical L3G4200D chips (X-axis)

(b) Standard deviation of raw data samples for 12 identical L3G4200D chips (Y-axis)

(c) Standard deviation of raw data samples for 12 identical L3G4200D chips (Z-axis)

(d) Raw data samples of one L3G4200D chip with the single tone sound noise at 8,000Hz

Figure 6: Sound noise effect on L3G4200D gyroscopes (all samples were collected as raw data stored in the gyroscope's register)

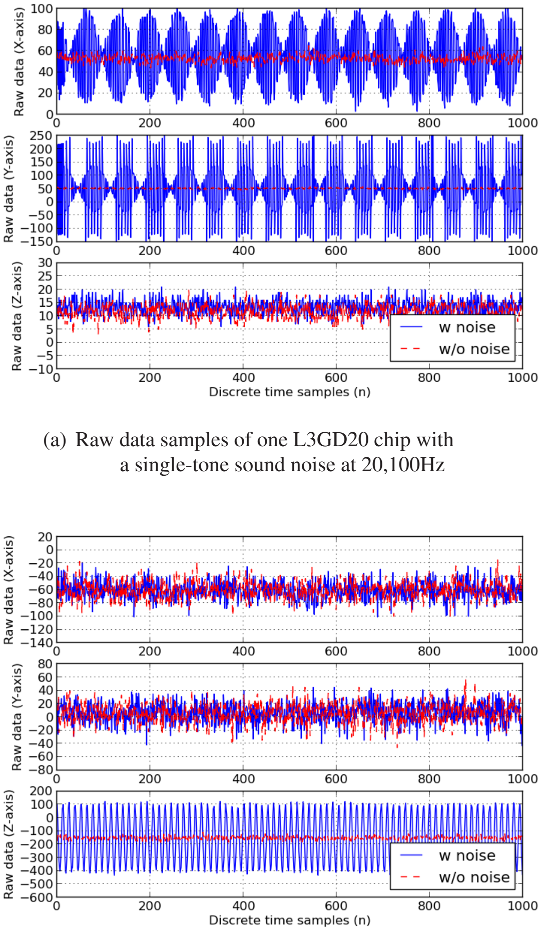
choic chamber, with and without sound noise. Because the standard deviation of the raw data samples should ideally be zero without sound noise when the target gyroscopes are on the frame, we consider the difference in the standard deviations with and without sound noise as a criterion for the resonance of the target gyroscopes.

The results of the experiment are summarized in Table 1. The third and fourth columns indicate the degrees of freedom and the interface type of each gyroscope, respectively. The resonant frequencies [2] and axes from the datasheets [4, 13, 14, 15, 16, 17, 18, 19, 25, 28, 29, 30, 31, 32] are listed in the fifth column, and the resonant frequencies identified in our experiment are listed in the last column.

Our results show that seven of these gyroscopes (i.e., vulnerable gyroscopes) resonated at their own resonant frequencies in response to sound noise. Three of the vulnerable gyroscopes were manufactured by *STMicroelectronics*, and the others were manufactured by *InverSense*. No documentation on the resonant frequencies of the tested gyroscopes was available from vendors other than *InvenSense* and *Analog Devices*. We figured out that the gyroscopes manufactured by *STMicroelectronics* had resonant frequencies in the audible range (almost below 20 kHz), and that they were affected considerably more along the X-axis and Y-axis than along the Z-axis. In contrast, the gyroscopes manufactured by *InvenSense* resonated in the ultrasound range (above 20 kHz) and were affected in the Z-axis direction only.

Both keeping resonant frequencies secret and raising them to the higher-frequency region are good ways to

[2] These are described as mechanical frequencies in the datasheets for the *InvenSense* gyroscopes.



(a) Raw data samples of one L3GD20 chip with a single-tone sound noise at 20,100Hz

(b) Raw data samples of one MPU6000 chip with a single-tone sound noise at 26,800Hz

Figure 7: Sound noise effects on two vulnerable MEMS gyroscopes (all samples were collected as raw data stored in the gyroscope's register)

reduce resonance due to sound noise. However, as our results show, resonance can be induced by a malicious attacker, as long as resonant frequencies exist in gyroscopes. Additionally, the standard deviations of the output data from these gyroscopes are largely increased without any rotation or tilt when the resonance occurs as a result of intentional sound noise. This abnormal output can potentially make gyroscope application systems malfunction.

We did not detect resonance effects for the other eight gyroscopes evaluated in our experiments. Particularly, for five of these gyroscopes, no resonant frequencies were observed, even though their resonant frequencies are described in their datasheets. We obtained additional measurements with the frequency resolution enhanced by a factor of two (50 Hz), but resonant frequencies were not found. It might be possible that the frequency intervals (100 Hz and 50 Hz) used in our tests were not sufficiently narrow. The fact that resonant frequencies were not detected in our experiments does not necessarily mean that they do not exist in the frequency range below 30 kHz.

A comparison between the standard deviations (σ_{axis}) with and without sound noise for the seven vulnerable gyroscopes is presented in Table 2. To validate our attack method, 12 individual gyroscope chips were tested for L3G4200D, L3GD20, and MPU6000, whereas only two chips were tested for the others. All of the values shown in Table 2 are average for all outputs from the same kind of vulnerable gyroscopes. The standard deviations of the gyroscope outputs with sound noise at the resonant frequencies are relatively large. The ratios of the standard deviations with sound noise to those without sound noise are summarized in the last three columns. The standard deviations changed by factors up to dozens, with the greatest change being by a factor of 31.04 (for the Y-axis of L3GD20).

Figures 6(a), 6(b), and 6(c) show the standard deviations of the raw data samples for each axis from the 12 individual L3G4200D chips. The different L3G4200D chips have different output characteristics because of manufacturing variances. However, every L3G4200D chip has a peak in the range of 7,900 to 8,300 Hz. To investigate what happens at these frequencies in more detail, the raw data samples for one L3G4200D gyroscope with and without sound noise at 8,000 Hz were compared, as shown in Figure 6(d). This graph clearly shows that resonances occur for all axes, and the amplitudes are dozens of times larger than the normal output. These amplitudes are equivalent to the output produced by sudden and fast shaking of the gyroscope or the target drone's body by hands or rapidly changing winds. Raw data samples of two other vulnerable gyroscopes, L3GD20 and MPU6000, are shown in Figure 7. L3G4200D and MPU6000, two of the vulnerable gyroscopes in our experiments, were used in the target drones described in the next section.

It should be noted that a speaker generates sound from a vibrating membrane fixed to the enclosure of the speaker, and thus vibration from the enclosure itself was unavoidable in the experiments. However, our experimental results indicate that vibration had very little effect on the identification of the resonant frequencies of the target gyroscopes. Because we tested all of the gyroscopes in the same environment, there should have been consistent resonance frequencies for all of the gyroscopes if any enclosure vibration had influenced the motion of the gyroscopes. In addition, some of the gyroscopes listed in Table 1 exhibited no resonance (i.e., almost constant standard deviation), which would not have

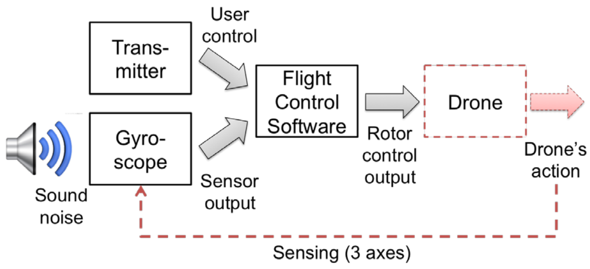


Figure 8: Propagation of the effect of sound noise

been possible if there had been a strong vibration due to vibration of the enclosure.

5 Attacking Drone

As described in the previous section, the outputs of MEMS gyroscopes fluctuate with the sound noise at the gyroscopes' own resonant frequencies. This section describes the impact of this fluctuation on the control of a drone. To understand this, we first need to understand how the user input from a remote controller and the input from the gyroscope propagate to the operation of a drone. Figure 8 shows each step in this propagation. The flight control software calculates each control signal for four rotors based on the user input and gyroscope output. This control signal mechanically controls the speed of each rotor, which determines the tilt, orientation, and rotation of the drone in turn. This section describes the analysis of how sound noise at the resonant frequency of a gyroscope affects control of target drones.

We took the following three steps. 1) To understand the reaction of the target drones as actuators to the fluctuation of the gyroscope output as abnormal sensing, the flight control software was analyzed statically. 2) We then launched our attack on two target drones under real-world conditions to assess the effect of the maximum sound noise against them. 3) To identify cost-efficient parameters for our attack, we performed software simulations with gyroscope outputs varying from 1% to 100 % of the maximum noise.

5.1 Target Drones

For this experiment, two DIY drones were built for use as the target drones, and they were equipped with L3G4200D and MPU6000 respectively, two of the vulnerable gyroscopes. This approach was taken because the gyroscopes on most finished drones are not user selectable, and it was necessary to evaluate the effect of sound noise in the sensing and actuation systems. The main specifications of the two target drones are given in Table 3. All DIY drones require calibration for stable operation. Following the instructions in the manual for the flight control software, we calibrated the IMU sensors and four rotor controllers, and we adjusted the PID gains (see Section 3.2) for stable flight.

5.2 Software Analysis

Target drone A's flight control software, *Multiwii* [24], supports various gyroscopes. However, the main routine of this software is essentially the same for all gyroscopes except with respect to the way the sensors are prepared and the way the raw data are accessed. The main processor reads the raw data from the gyroscope's registers through an I^{2}C interface, along with the raw data from the transmitter controlled by the user. Each raw data sample for each axis was stored in two 8-bit registers. These raw data were the main inputs to the flight control software, and the outputs were the rotor control data calculated by the PID control algorithm. The PID controller seeks to minimize the difference between the measured control and the desired control for the control systems. While PID controller implementation and PID gains vary depending on their application and the gyroscope used, the fundamental algorithm remains the same.

Algorithm 1 describes a high-level implementation of the default PID control algorithm in this flight control software. The details of the software are omitted for simplicity. Conceptually, the P, I, and D terms influence the target drone's control as follows:

- P is proportional to the present output of the gyroscope, and if the present output value ($gyro[axis]$) of the gyroscope is abnormally large, the desired control from the transmitter ($txCtrl[axis]$) can be ignored (line 7).
- I is proportional to the accumulated error between the output from the transmitter and the gyroscope (line 10), which can be ignored, because the default value of the I term gain (G_I) for the target drone is very small.

Table 3: Specifications of two target drones for the real world attacking experiment

Spec.	Target Drone A	Target Drone B
Processor	STM32F103CBT6	ATMEGA2560
Gyroscope	L3G4200D	MPU6000
Flight Ctrl. Software	Multiwii [24]	ArduPilot [7]
Diagonal Frame Size	45 cm	55 cm
Propeller Size	10 × 4.5	10 × 4.5

Algorithm 1: Simplified PID algorithm of *Multiwii* flight controller (calculating the rotor control data according to the output of the gyroscope)

```
Input: The sensed data from the MEMS gyroscope
Input: The received data from the transmitter
Output: The data to control the rotor
1  initialization;
2  $G_P$, $G_I$, and $G_D$: pre-configured P, I, and D gain by
   user (configured as the default values);
3  while True do
4      read data from the gyroscope for 3 axes;
5      receive data from the transmitter for 4 channels
       (3 axes and throttle);
6      for axis do
7          $P = txCtrl[axis] - gyro[axis] \times G_P[axis]$;
8          $error = txCtrl[axis]/G_P[axis] - gyro[axis]$;
9          $error_{accumulated} = error_{accumulated} + error$;
10         $I = error_{accumulated} \times G_I[axis]$;
11         $delta = gyro[axis] - gyro_{last}[axis]$;
12         $delta_{sum}$ = sum of the last three delta values;
13         $D = delta_{sum} \times G_D[axis]$;
14         $PIDCtrl[axis] = P + I - D$;
15     end
16     for rotor do
17         for axis do
18             $rotorCtrl[rotor] =$
               $txCtrl[throttle] + PIDCtrl[axis]$;
19         end
20         limit $rotorCtrl[rotor]$ within the pre-defined
           MIN (1,150) and MAX (1,850) values;
21     end
22     actuate rotors;
23 end
```

- D is proportional to the changes ($delta_{sum}$) between the previous and present output values of the gyroscope (line 13).

These three terms directly affect the PID control values ($PIDCtrl[axis]$) for each axis (line 14). If the values of P and D are abnormally large, the PID control values will also increase abnormally. The desired throttle control ($txCtrl[throttle]$) can thus be ignored (line 18). In the end, all rotor control values are constrained by the pre-defined minimum and maximum values (line 20). Throughout the process, the raw data from the gyroscope were not checked, filtered, or verified. In other words, the target drone system fully trusted the integrity of the gyroscope output in its sensing and actuation. Therefore, the control of the target drone could be directly affected by our attack.

We also analyzed the flight control software of *ArduPilot* [7] for target drone B. A manual software analysis shows that the PID algorithm used in *ArduPilot* is essentially the same as that used with target drone A. The only difference between two algorithms is in slight changes of the gains that are multiplied to each of the P, I, and D terms. This can be considered a discrepancy in the configuration values of the sensors.

5.3 Real-World Experiment

While the software analysis described in the previous section led us to believe that the $PIDCtrl[axis]$ values would fluctuate when the gyroscope outputs fluctuated, this information was not sufficient to answer the following questions: 1) Given user inputs $txCtrl[throttle]$ and fluctuating $PIDCtrl[axis]$, how much does $rotorCtrl[rotor]$ change? 2) How does a change in $rotorCtrl[rotor]$ affect the behavior of the drone? To answer these questions, we decided to launch our attack in the real world with sound noise causing the fluctuation.

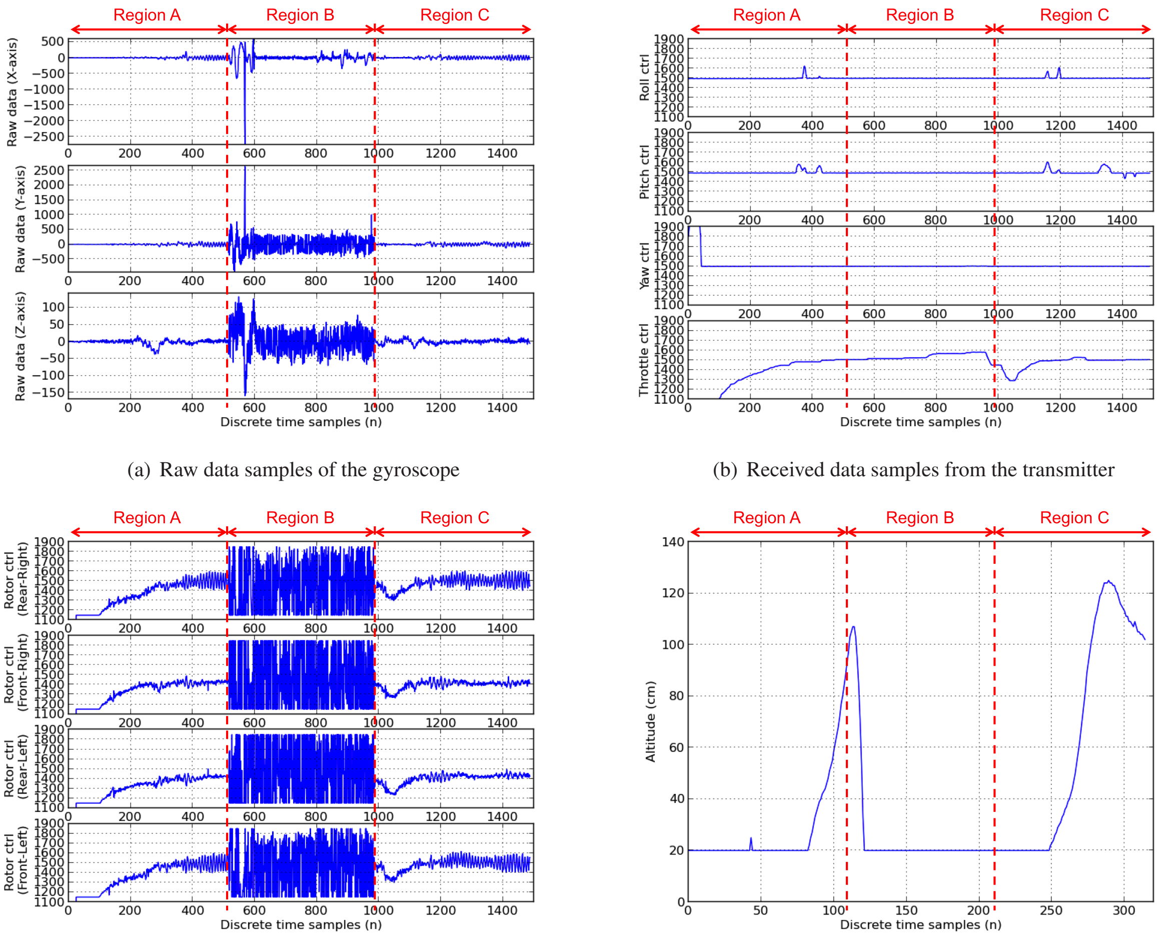
Attack Setup: In this experiment, we attached a small Bluetooth speaker above the target system's gyroscope at a distance of 10 cm to serve as an attacking sound source. The SPL of the fundamental frequency component was 113 dB with the maximum volume of the speaker. Low THD+N was not a consideration for the sound source used in the attack. The sound noise was turned on while the target drones were stably maintained in the air. To observe the status of the target drones before, during, and after the attack, sound noise at the resonant frequency was turned off, turned on (attack), and turned off again for every 10 seconds.

Attack Results: The results of our attack experiment are summarized on two target drones (A and B) in Table 4. Our attacks successfully disrupted control of target drone A, but it did not affect target drone B. The reason of attack failure on target drone B is that the gyroscope of target drone B resonated only along the Z-axis. The Z-axis of target drone B corresponds to the horizontal orientation that is also sensed by the magnetometer on the board.

We also attached a sonar device to gauge the altitude

Table 4: Result of attacking two target drones

Item	**Target Drone A**	**Target Drone B**
Resonant Freq. (Gyroscope)	8,200 Hz (L3G4200D)	26,200 Hz (MPU6000)
SPL at Resonant Freq.	97 dB	95 dB
Affected Axes	X, Y, Z	Z
Attack Result	Fall down	Not affected



(a) Raw data samples of the gyroscope

(b) Received data samples from the transmitter

(c) Rotor control data samples (from the flight control software)

(d) Altitude data samples from sonar

Figure 9: The results of our attack against target drone A in a real-world experiment (sound noise turned off, on, and off every 10 seconds; note that the sonar's sampling rate was different from that for the data in other figures)

and two Bluetooth-to-UART (Universal Asynchronous Receiver/Transmitter) modules to collect real-time data from target drone A. The Bluetooth-to-UART modules were connected to a UART interface on target drone A's flight controller board and the sonar module. Using this UART interface, we were able to communicate with a computer for configuration purpose. We were also able to monitor the status of target drone A, including the raw data from the sensors and the rotor control data, using the *Multiwii* [24] Graphical User Interface (GUI) program. By analyzing the *Multiwii* source code, we were able to understand the protocol used for the UART communication. Each request or response message consists of a 3-bytes fixed header, 1 byte for the data length (n), a 1-byte command, n bytes of data, and a 1-byte checksum. Using this protocol and the Bluetooth-to-UART modules, we were able to record the resonant outputs of the gyroscope, the control data from the transmitter, the rotor control data of the flight control software, and the altitude data from target drone A in the air. Note that the altitude data were sampled at a different rate than the other data because of a technical limitation of the sonar module, and the minimum sensing distance of the sonar was 20 cm.

Figure 9 shows the detailed results of the attack against target drone A in the real-world experiment. Region A in Figure 9 corresponds to the period before the attack. The user gradually raised the throttle (Figure 9(b)), and the speeds of the four rotors were increased correspondingly (Figure 9(c)). In response, target drone A rose over 100 cm in the air (Figure 9(d)). When the attack was started (Region B), the output of the gyroscope fluctuated because of the sound noise at the resonant frequency (Figure 9(a)). According to the resonant output of the gyroscope, the rotor control data fluctuate between the maximum and minimum values (Region B in Figure 9(c)).

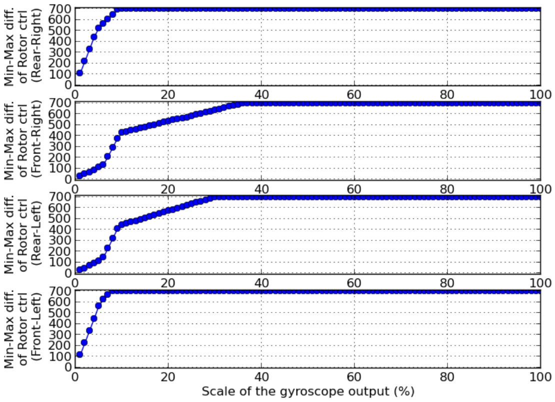


Figure 10: Rotor control outputs from our software simulation (the maximum change of the rotor control output was 700)

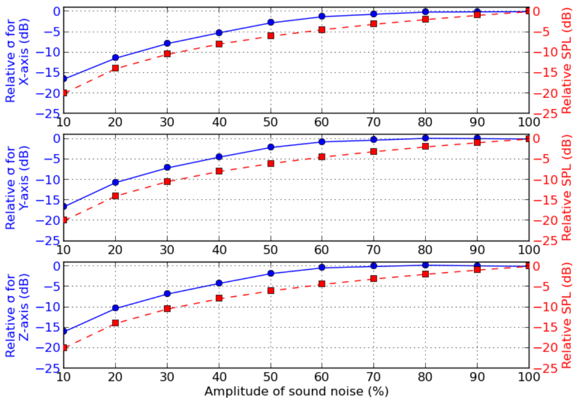


Figure 11: Sound noise effect on one L3G4200D gyroscope versus sound noise amplitude with theoretical relative SPL (data averaged for ten identical experiments and 1,000 raw data samples collected per experiment)

When the attack started, target drone A dropped instantaneously. During the attack, target drone A could not ascend or recover its control, even though throttle control was maintained to allow it to ascend slowly (Figure 9(b)). After the attack was stopped (Region C), target drone A ascended normally again and recovered its control. We attacked target drone A 20 times in the real-world experiments, and it lost control and crashed shortly after our attack in every test.

To assess the effectiveness and practicality of our attack, more real-world attack experiments are required. However, there are obstacles such as the damage to the target drone (e.g., broken arms) and the repetitive recalibration required after each crash, because the unpredictable changes in the drone's balancing are fed back into the gyroscope by our attack (see the dotted line and box in Figure 8).

5.4 Attack Distance

Our real-world experiments showed that an acoustic attack can completely incapacitate a target drone equipped with a gyroscope vulnerable to X-axis and Y-axis resonance due to sound incidence. We also want to determine the conditions or bounds of a cost-effective attack. For example, we need to find out possible attack distance or sound level of a sound source required to destabilizing a target drone in the air.

We may try to conduct tests at various distances to discover either the approximate minimum distance or the sound level required to incapacitate target drone A in the air. However, it would disrupt the stability of the target drone to attach a longer structure with the sound source on the target drone. It is also difficult to take aim at the target drone with sound noise from outside during its flight without attaching any structure to it.

Therefore, to minimize the number of trials and overcome the practical limitations mentioned above, we first ran a simulation using the functions of Algorithm 1, which were extracted from the source code for target drone A. Based on the results of this simulation, we found out the effective fluctuation (i.e., standard deviation) of the gyroscope output with a few real-world tests. Then, we measured the standard deviations of target drone A on a desk exposed to sounds of various amplitudes. By combining the results of this simulation and our measurements, we were able to identify an approximate range of sound amplitude for testing the target drone in the air. We then derived the feasible attack distance theoretically using the SPL value that we had measured in our attack with the effective amplitude of the sound noise.

Simulation: For the software simulation, the recorded gyroscope output and the control data from the transmitter in the real-world attack experiments were used as the inputs. The recorded gyroscope output was linearly scaled from 1 % to 100 % in increments of 1 %, and the control data from the transmitter were the same as in the real-world experiment. Figure 10 shows the results of the simulation. Because the rotor control output was bounded between 1,150 and 1,850 in Algorithm 1, the maximum change of the rotor control output was 700. The minimum scale of the gyroscope output that could achieve the maximum change in all rotor controls was 37 % in our simulation (Figure 10).

Indoor Measurements: The standard deviation of the gyroscope output with respect to the sound noise amplitude was measured for the L3G4200D gyroscope of target drone A on a desk. Figure 11 shows the relative standard deviation of the gyroscope output measured at a 10 cm distance, which decreases logarithmically as the sound level decreases. Theoretically, the relationship between the sound amplitude and SPL is described by Equation 1 [27]. At the point of the reference SPL (SPL_{ref}), the amplitude of the sound noise signal is A_{ref}.

$$SPL = SPL_{ref} + 20\log\left(\frac{A}{A_{ref}}\right) \tag{1}$$

The relative SPL obtained by changing the amplitude is the second term in Equation 1, and it is illustrated in Figure 11, along with the measured relative standard deviations. The decreasing trend in our measurements is similar to that for the theoretical relative SPLs, but the amount of decrease in our measurements was smaller than that for the theoretical relative SPLs from the amplitude range over 70 %. This mismatch is the typical output characteristic of consumer-grade speakers at high amplitude levels, which is caused by the nonlinear distortion that also leads to the leakage of sound energy into harmonic and subharmonic frequencies.

Distance Analysis: The amplitude of the sound noise corresponding to 37 % (-8.64 dB) of the standard deviation in Figure 11 is approximately 27 %, because the standard deviation of the gyroscope output is proportional to the scale of the gyroscope output. Accordingly, the sound noise greater than 27 % in amplitude can induce the maximum changes in all rotor controls for target drone A, if the drone is tested at the same environment as that of our real-world attack.

In the real-world experiments, we changed the amplitude of the sound noise in the same environment and observed that around 30 % sound amplitude is the lower bound for making target drone A crash. The SPL measured at this 30 % sound amplitude was 108.5 dB. Using the following relationship between the distance and SPL [58], we can derive a possible attack distance of a remotely located sound source, where the reference distance (d_{ref}) and SPL (SPL_{ref}) are those measured from the real-world attack experiments.

$$SPL = SPL_{ref} - 20\log\left(\frac{d}{d_{ref}}\right) \tag{2}$$

According to this prediction, the possible attack distance is approximately 16.78 cm using the same sound source that we used for the real-world attack with the maximum volume (113 dB). This attack distance range might not be sufficient for a malicious attacker. However, attackers can overcome this distance limitation by using a more powerful and directional source (e.g., a loudspeaker array) than the single speaker used in our experiments. For instance, *SB-3F* [23] from *Meyersound* can generate sound of 120 dB at 100 m, and *450XL* [21] from *LRAD* and *HyperShield* [33] from *UltraElectronics* can produce 140 dB at 1 m, which is equivalent to 108.5 dB at 37.58 m. Therefore, the possible attack distance is 37.58 m, if an attacker uses a sound source that can generate 140 dB of SPL at 1 m.

6 Discussion

In this section, we present a discussion of potential attack scenarios and countermeasures.

6.1 Potential Attack Scenarios

The attack model used in this paper seems to be too strong in two ways: 1) Use of audible sound can be easily detected, and 2) the speaker is close to the drone body. However, the more practical attack can be designed to weaken this attack model from the analysis result of this study.

First, several gyroscopes listed in Table 1 have resonant frequencies in the inaudible band (i.e., above 20 kHz). If the resonant frequency is above 20 kHz, a successful attack is possible using an ultrasonic sound generator and transducer. In addition, sound at frequencies higher than 15 kHz is difficult for humans to hear.

Second, the distance analysis shows that various remote attacks are also possible using different types of sound generators. Some of promising ways for the remote attack are described below.

Compromising the Sound Source: It is not hard to imagine drones with speakers (consider police and military operations or search-and-rescue operations). If one can compromise the source of the sound from the speaker, the effect will be the same as that of our original attack model. For example, insecurity of the Hybrid Broadcast-Broadband Television (HbbTV) standard and implementation would allow an adversary to control the TV stream [48].

Drone to Drone Attack: In 2013, Kamkar demonstrated the 'SkyJack' attack, in which an adversary drone hijacks a victim drone using a wireless denial-of-service attack [44]. A similar attack could involve following and taking a picture of a moving object, which could become a popular drone application. An adversary drone equipped with a speaker could steer itself toward a victim drone and generate a sound with the resonant frequency of the victim's gyroscope to drag it down. Of course, in this case, the resonant frequency of the adversary's gyroscope has to be different from that of the victim.

Long Range Acoustic Device: Long Range Acoustic Device (LRAD) [56] could be used as a sonic weapon [57] or Acoustic Hailing Device (AHD) [54]. Sonic weapons can cause damages to human organs by inducing intense sound waves at certain frequencies, even if the sound source is not in contact with opponents [41, 53]. AHDs are specially designed loudspeakers that communicate over longer distances than normal loudspeakers [21, 23, 33]. In both cases, the most important requirement is a high SPL in a specific frequency band. Obviously, these technologies could be used to increase the range of our attack.

Sonic Wall/Zone: Because drones can be made small, they can be difficult to detect using radar. Therefore, it might be desirable to enforce no-fly zones for drones, as illustrated by recent drone incidents [11, 34]. One might consider building a sonic wall or a zone that radiates continuous sound noise (at various frequencies) in a specific area to enforce the no-fly zone.

6.2 Countermeasures

Several researches that have been conducted to improve the performance of MEMS gyroscopes in harsh acoustic environments are discussed below.

Physical Isolation: The simplest way to mitigate our attack is to provide physical isolation from the sound noise. This is the same concept as shielding against Electro Magnetic Interference (EMI). For example, the iPhone 5S, which is equipped with an L3G4200D gyroscope [20], would not be affected by our attack, because of the compact casing of the hardware circuit. Surrounding the gyroscope with foam would also be a simple and inexpensive countermeasure. Foam that is 1 inch thick has approximately 120 dB insertion loss in SPL [49].

Figure 12 shows the result of physical isolation experiments conducted using four different materials: a paper box, an acrylic panel, an aluminum plate, and foam. We put these materials between the sound source and the target gyroscope. The isolation performances of the different materials were not very different. Using these materials, the effect of the sound noise on one L3G4200D gyroscope was decreased to 23.78%, 16.25%, and 60.49% for the three axes.

Differential Comparator: While physical isolation is a passive approach to mitigation, use of a differential comparator is an active approach to mitigation. Using an additional gyroscope with a special structure that responds only to the resonant frequency, the application systems can cancel out the resonant output from the main gyroscope [52]. The concept of this countermeasure was introduced by Kune et al. [45] to detect and cancel out analog sensor input spoofing against CIEDs.

Resonance Tuning: In the operation of MEMS gyroscopes, the bending mentioned in Section 3.3.1 changes the capacitance between the sensing mass and the sensing electrode, and this capacitance change is sensed as the output of the gyroscope. By using an additional feedback capacitor connected to the sensing electrode, the resonant frequency and the magnitude of the resonance effect can be tuned [35, 43].

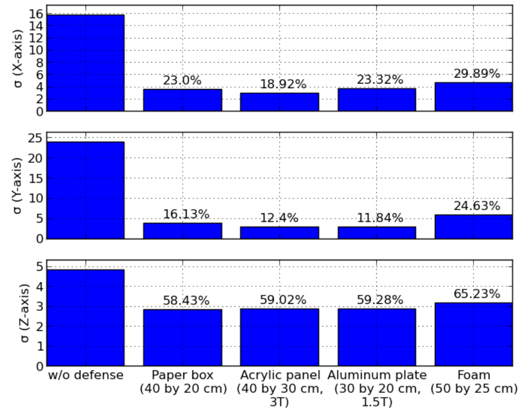


Figure 12: Physical isolation test for one L3G4200D gyroscope with four different materials (data averaged for ten identical experiments and 1,000 raw data samples collected per experiment)

These countermeasures may be used to mitigate our attack. However, physically surrounding the gyroscope sensor with certain materials could cause several problems, such as affecting other sensors or components and raising the temperature of the board. These problems may cause malfunctions of the drone control systems. In addition, use of a differential comparator with another gyroscope implies an additional cost. The resonance tuning countermeasure also has the limitation that the resonant frequency does not disappear as a result of tuning. Because the resonant frequency still exists, an attack at that frequency remains possible.

7 Conclusions and Future Work

Many sensing and actuation systems trust their measurements, and actuate according to them. Unfortunately, this trust can lead to security vulnerabilities that cause critically unintended actuations. We found that the sound channel can be used as a side channel for MEMS gyroscopes from a security point of view. In our experiment, we tested 15 kinds of MEMS gyroscopes, and seven of them were found to be vulnerable to disruption using intentional sound noise. The output of the vulnerable MEMS gyroscopes was found using a consumer-grade spaeker to fluctuate up to dozens of times as a result of sound noise.

To demonstrate the effects of this vulnerability, we implemented an attack against two target drones equipped with different kinds of vulnerable MEMS gyroscopes.

As a result of a firmware analysis of the target drones and a simulation of the flight control software output, the control signals of four rotors were found to fluctuate up to the maximum value and down to the minimum value by the injected gyroscope output. One of the target drones, which was equipped with with a small speaker, lost control and crashed in all 20 real-world attack experiments. We found in these experiments that an attacker with only 30% of the amplitude of the maximum sound noise could achieve almost the same effect at the same distance.

The countermeasures that are mentioned in the last subsection have limitations and require hardware modifications and additional materials. Because these mitigations would increase the production costs, it is necessary to develop a low-cost, software-based defense mechanism against sensor attacks for various types of embedded devices.

Some MEMS gyroscopes are integrated with accelerometers in the same IC package. In our experiments, we found that some accelerometers are also affected by high-power sound noise at certain frequencies. It would be interesting to further investigate this finding.

Acknowledgements

This work was supported by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-TB1403-01.

References

[1] A Critical Review of MEMS Gyroscopes Technology and Commercialization Status. http://invensense.com/mems/gyro/documents/whitepapers/MEMSGyroComp.pdf.

[2] Alibaba begins drone delivery trials in China. http://www.bbc.com/news/technology-31129804.

[3] Amazon Prime Air (Amazon.com, Inc.). http://www.amazon.com/b?node=8037720011.

[4] Analog Devices ADXRS610 datasheet. http://www.analog.com/media/en/technical-documentation/data-sheets/ADXRS610.pdf.

[5] AR.Drone (Parrot, Inc). http://ardrone2.parrot.com/.

[6] Arduino UNO. http://arduino.cc/.

[7] ArduPilot (open-source drone project). https://github.com/diydrones/ardupilot.

[8] Brüel & Kjær Microphone Unit Type 4189-A-021. http://www.bksv.com/Products/transducers/acoustic/microphones/microphone-cartridges/4189.

[9] Design and Analysis of MEMS Gyroscopes (Tutorial at IEEE Sensor 2013). http://ieee-sensors2013.org/sites/ieee-sensors2013.org/files/Serrano_Slides_Gyros2.pdf.

[10] Everything about STMicroelectronics' 3-axis digital MEMS gyroscopes. http://www.st.com/web/en/resource/technical/document/technical_article/DM00034730.pdf.

[11] German pirate party uses drone to crash angela merkel event. http://www.slate.com/blogs/future_tense/2013/09/18/german_pirate_party_uses_drone_to_crash_event_with_chancellor_angela_merkel.html.

[12] Hi-Vi B1S Full Range Loudspeaker. https://www.madisoundspeakerstore.com/approx-1-fullrange/hi-vi-b1s-full-range/.

[13] InvenSense IMU3000 datasheet. http://www.invensense.com/mems/gyro/documents/PS-IMU-3000A.pdf.

[14] InvenSense ITG3200 datasheet. http://www.invensense.com/mems/gyro/documents/EB-ITG-3200-00-01.1.pdf.

[15] InvenSense IXZ650 datasheet. http://invensense.com/mems/gyro/documents/PS-IXZ-0650B-00-01.pdf.

[16] InvenSense MPU3050 datasheet. http://www.invensense.com/mems/gyro/documents/PS-MPU-3000A.pdf.

[17] InvenSense MPU6000/6050 datasheet. http://www.invensense.com/mems/gyro/documents/PS-MPU-6000A-00v3.4.pdf.

[18] InvenSense MPU6500 datasheet. http://www.invensense.com/mems/gyro/documents/PS-MPU-6500A-01.pdf.

[19] InvenSense MPU9150 datasheet. http://dlnmh9ip6v2uc.cloudfront.net/datasheets/Sensors/IMU/PS-MPU-9150A.pdf.

[20] iPhone 5s Teardown. https://www.ifixit.com/Teardown/iPhone+5s+Teardown/17383.

[21] LRAD 450XL datasheet. http://www.lradx.com/wp-content/uploads/2015/05/LRAD_Datasheet_450XL.pdf.

[22] Market share information of MEMS gyroscope in 2013 (page 17). http://www.semiconwest.org/sites/semiconwest.org/files/data14/docs/SW2014_JCEloy_YoleDeveloppement_0.pdf.

[23] Meyersound SB-3F datasheet. http://www.meyersound.com/sites/default/files/sb-3f_ppi.pdf.

[24] Multiwii (open-source drone project). https://github.com/multiwii/baseflight and https://code.google.com/p/multiwii/.

[25] Murata ENC-03MB datasheet. http://www.mouser.com/catalog/specsheets/ENC-03M_ref.pdf.

[26] National Instruments USB-4431, Sound and Vibration Data Acquisition Instrument. http://www.ni.com/pdf/products/us/cat_usb4431.pdf.

[27] Relative Sound Pressure according to Amplitude. http://www.indiana.edu/~emusic/acoustics/amplitude.htm.

[28] STMicroelectronics L3G4200D datasheet. http://www.st.com/web/en/resource/technical/document/datasheet/CD00265057.pdf.

[29] STMicroelectronics L3GD20 datasheet. http://www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/DM00036465.pdf.

[30] STMicroelectronics LPR5150AL datasheet. http://www.st.com/web/en/resource/technical/document/datasheet/CD00237211.pdf.

[31] STMicroelectronics LPY503AL datasheet. http://www.st.com/web/en/resource/technical/document/datasheet/CD00237199.pdf.

[32] STMicroelectronics LSM330 datasheet. http://www.st.com/web/en/resource/technical/document/datasheet/DM00059856.pdf.

[33] UltraElectronics HyperShield datasheet. `http://www.ultra-hyperspike.com/Data/Pages/26fa8e2abe074313d60fe15a9af35440-HyperShield_Dat_Sheet.pdf`.

[34] White house drone crash described as a u.s. workers drunken lark. `http://www.nytimes.com/2015/01/28/us/white-house-drone.html`.

[35] ADAMS, S., BERTSCH, F., SHAW, K., HARTWELL, P., MACDONALD, N. C., AND MOON, F. Capacitance Based Tunable Micromechanical Resonators. In *International Conference on Solid-State Sensors and Actuators* (1995).

[36] CAI, L., AND CHEN, H. On the practicality of motion based keystroke inference attack. In *Trust and Trustworthy Computing*. Springer Berlin Heidelberg, 2012.

[37] CASTRO, S., DEAN, R., ROTH, G., FLOWERS, G. T., AND GRANTHAM, B. Influence of acoustic noise on the dynamic performance of MEMS gyroscopes. In *International Mechanical Engineering Congress and Exposition* (2007), American Society of Mechanical Engineers.

[38] DEAN, R. N., CASTRO, S. T., FLOWERS, G. T., ROTH, G., AHMED, A., HODEL, A. S., GRANTHAM, B. E., BITTLE, D. A., AND BRUNSCH, J. P. A characterization of the performance of a MEMS gyroscope in acoustically harsh environments. *IEEE Transactions on Industrial Electronics 58* (2011).

[39] DEAN, R. N., FLOWERS, G. T., HODEL, A. S., ROTH, G., CASTRO, S., ZHOU, R., MOREIRA, A., AHMED, A., RIFKI, R., GRANTHAM, B. E., ET AL. On the degradation of MEMS gyroscope performance in the presence of high power acoustic noise. In *IEEE International Symposium on Industrial Electronics* (2007).

[40] DUC, N. M., AND MINH, B. Q. Your face is not your password face authentication bypassing Lenovo–Asus–Toshiba. *Black Hat Briefings* (2009).

[41] FOWLKES, J. B., AND HOLLAND, C. K. Section 4: Bioeffects in tissues with gas bodies. *Journal of ultrasound in medicine 19* (2000).

[42] GALBALLY, J., CAPPELLI, R., LUMINI, A., MALTONI, D., AND FIERREZ, J. Fake fingertip generation from a minutiae template. In *International Conference on Pattern Recognition* (2008).

[43] JEONG, C., SEOK, S., LEE, B., KIM, H., AND CHUN, K. A study on resonant frequency and Q factor tunings for MEMS vibratory gyroscopes. *Journal of Micromechanics and Microengineering 14* (2004).

[44] KAMKAR, S. SkyJack. `http://samy.pl/skyjack/`, 2013.

[45] KUNE, D. F., BACKES, J., CLARK, S. S., KRAMER, D., REYNOLDS, M., FU, K., KIM, Y., AND XU, W. Ghost talk: mitigating EMI signal injection attacks against analog sensors. In *IEEE Symposium on Security and Privacy* (2013).

[46] MATSUMOTO, T., MATSUMOTO, H., YAMADA, K., AND HOSHINO, S. Impact of artificial gummy fingers on fingerprint systems. In *Electronic Imaging* (2002), International Society for Optics and Photonics.

[47] MILUZZO, E., VARSHAVSKY, A., BALAKRISHNAN, S., AND CHOUDHURY, R. R. Tapprints: your finger taps have fingerprints. In *Proceedings of the ACM international conference on Mobile Systems, Applications, and Services* (2012).

[48] OREN, Y., AND KEROMYTIS, A. D. From the aether to the ethernet–attacking the internet using broadcast digital television. In *Proceedings of the USENIX Security Symposium* (2014).

[49] ROTH, G. Simulation of the Effects of Acoustic Noise on MEMS Gyroscopes. Master's thesis, Auburn University, 2009.

[50] SAMLAND, F., FRUTH, J., HILDEBRANDT, M., HOPPE, T., AND DITTMANN, J. AR.Drone: security threat analysis and exemplary attack to track persons. In *Society of Photo-Optical Instrumentation Engineers Conference Series* (2012).

[51] SHOUKRY, Y., MARTIN, P., TABUADA, P., AND SRIVASTAVA, M. Non-invasive spoofing attacks for anti-lock braking systems. In *Cryptographic Hardware and Embedded Systems*. Springer, 2013.

[52] SOOBRAMANEY, P. *Mitigation of the Effects of High Levels of High-Frequency Noise on MEMS Gyroscopes*. PhD thesis, Auburn University, 2013.

[53] TANDY, V., AND LAWRENCE, T. R. The ghost in the machine. *Journal of the Society for Psychical Research 62* (1998).

[54] WIKIPEDIA. Acoustic hailing device — wikipedia, the free encyclopedia, 2015. [Online; accessed 17-June-2015].

[55] WIKIPEDIA. Inertial measurement unit — wikipedia, the free encyclopedia, 2015. [Online; accessed 17-June-2015].

[56] WIKIPEDIA. Long range acoustic device — wikipedia, the free encyclopedia, 2015. [Online; accessed 17-June-2015].

[57] WIKIPEDIA. Sonic weapon — wikipedia, the free encyclopedia, 2015. [Online; accessed 17-June-2015].

[58] WIKIPEDIA. Sound pressure — wikipedia, the free encyclopedia, 2015. [Online; accessed 17-June-2015].

[59] YAN MICHALEVSKY AND DAN BONEH AND GABI NAKIBLY. Gyrophone: Recognizing speech from gyroscope signals. In *Proceedings of the USENIX Security Symposium* (2014).

Cache Template Attacks:
Automating Attacks on Inclusive Last-Level Caches

Daniel Gruss, Raphael Spreitzer, *and* Stefan Mangard
Graz University of Technology, Austria

Abstract

Recent work on cache attacks has shown that CPU caches represent a powerful source of information leakage. However, existing attacks require manual identification of vulnerabilities, i.e., data accesses or instruction execution depending on secret information. In this paper, we present *Cache Template Attacks*. This generic attack technique allows us to profile and exploit cache-based information leakage of any program automatically, without prior knowledge of specific software versions or even specific system information. Cache Template Attacks can be executed online on a remote system without any prior offline computations or measurements.

Cache Template Attacks consist of two phases. In the profiling phase, we determine dependencies between the processing of secret information, e.g., specific key inputs or private keys of cryptographic primitives, and specific cache accesses. In the exploitation phase, we derive the secret values based on observed cache accesses. We illustrate the power of the presented approach in several attacks, but also in a useful application for developers. Among the presented attacks is the application of Cache Template Attacks to infer keystrokes and—even more severe—the identification of specific keys on Linux and Windows user interfaces. More specifically, for lower-case only passwords, we can reduce the entropy per character from $\log_2(26) = 4.7$ to 1.4 bits on Linux systems. Furthermore, we perform an automated attack on the T-table-based AES implementation of OpenSSL that is as efficient as state-of-the-art manual cache attacks.

1 Introduction

Cache-based side-channel attacks have gained increasing attention among the scientific community. First, in terms of ever improving attacks against cryptographic implementations, both symmetric [4, 6, 16, 39, 41, 53] as well as asymmetric cryptography [3, 7, 9, 54], and second, in terms of developing countermeasures to prevent these types of attacks [31, 34]. Recently, Yarom and Falkner [55] proposed the Flush+Reload attack, which has been successfully applied against cryptographic implementations [3, 17, 22]. Besides the possibility of attacking cryptographic implementations, Yarom and Falkner pointed out that their attack might also be used to attack other software as well, for instance, to collect keystroke timing information. However, no clear indication is given on how to exploit such vulnerabilities with their attack. A similar attack has already been suggested in 2009 by Ristenpart et al. [44], who reported being able to gather keystroke timing information by observing cache activities on an otherwise idle machine.

The limiting factor of all existing attacks is that sophisticated knowledge about the attacked algorithm or software is necessary, i.e., access to the source code or even modification of the source code [7] is required in order to identify vulnerable memory accesses or the execution of specific code fragments manually.

In this paper, we make use of the Flush+Reload attack [55] and present the concept of *Cache Template Attacks*,[1] a generic approach to exploit cache-based vulnerabilities in any program running on architectures with shared inclusive last-level caches. Our attack exploits four fundamental concepts of modern cache architectures and operating systems.

1. Last-level caches are shared among all CPUs.
2. Last-level caches are inclusive, i.e., all data which is cached within the L1 and L2 cache must also be cached in the L3 cache. Thus, any modification of the L3 cache on one core immediately influences the cache behavior of all other cores.
3. Cache lines are shared among different processes.
4. The operating system allows programs to map any other program binary or library, i.e., code and static data, into their own address space.

[1] The basic framework can be found at `https://github.com/IAIK/cache_template_attacks`.

Based on these observations, we demonstrate how to perform Cache Template Attacks on any program automatically in order to determine memory addresses which are accessed depending on secret information or specific events. Thus, we are not only able to attack cryptographic implementations, but also any other event, e.g., keyboard input, which might be of interest to an attacker.

We demonstrate how to use Cache Template Attacks to derive keystroke information with a deviation of less than 1 microsecond from the actual keystroke and an accuracy of almost 100%. With our approach, we are not only able to infer keystroke timing information, but even to infer specific keys pressed on the keyboard, both for GTK-based Linux user interfaces and Windows user interfaces. Furthermore, all attacks to date require sophisticated knowledge of the attacked software and the executable itself. In contrast, our technique can be applied to any executable in a generic way. In order to demonstrate this, we automatically attack the T-table-based AES [10, 35] implementation of OpenSSL [37].

Besides demonstrating the power of Cache Template Attacks to exploit cache-based vulnerabilities, we also discuss how this generic concept supports developers in detecting cache-based information leaks within their own software, including third party libraries. Based on the insights we gained during the development of the presented concept, we also present possible countermeasures to mitigate specific types of cache attacks.

Outline. The remaining paper is organized as follows. In Section 2, we provide background information on CPU caches, shared memory, and cache attacks in general. We describe Cache Template Attacks in Section 3. We illustrate the basic idea on an artificial example program in Section 4 and demonstrate Cache Template Attacks against real-world applications in Section 5. In Section 6, we discuss countermeasures against cache attacks in general. Finally, we conclude in Section 7.

2 Background and Related Work

In this section, we give a basic introduction to the concept of CPU caches and shared memory. Furthermore, we provide a basic introduction to cache attacks.

2.1 CPU Caches

The basic idea of CPU caches is to hide memory accesses to the slow physical memory by buffering frequently used data in a small and fast memory. Today, most architectures employ set-associative caches, meaning that the cache is divided into multiple cache sets and each cache set consists of several cache lines (also called ways). An index is used to map specific memory locations to the sets of the cache memory.

We distinguish between virtually indexed and physically indexed caches, which derive the index from the virtual or physical address, respectively. In general, virtually indexed caches are considered to be faster than physically indexed caches. However, the drawback of virtually indexed caches is that different virtual addresses mapping to the same physical address are cached in different cache lines. In order to uniquely identify a specific cache line within a cache set, so-called tags are used. Again, caches can be virtually tagged or physically tagged. A virtual tag has the same drawback as a virtual index. Physical tags, however, are less expensive than physical indices as they can be computed simultaneously with the virtual index.

In addition, there is a distinction between inclusive and exclusive caches. On Intel systems, the L3 cache is an inclusive cache, meaning that all data within the L1 and L2 caches are also present within the L3 cache. Furthermore, the L3 cache is shared among all cores. Due to the shared L3 cache, executing code or accessing data on one core has immediate consequences for all other cores. This is the basis for the Flush+Reload [55] attack as described in Section 2.3.

Our test systems (Intel Core i5-2/3 CPUs) have two 32 KB L1 caches—one for data and one for instructions—per core, a unified L2 cache of 256 KB, and a unified L3 cache of 3 MB (12 ways) shared among all cores. The cache-line size is 64 bytes for all caches.

2.2 Shared Memory

Operating systems use shared memory to reduce memory utilization. For instance, libraries used by several programs are shared among all processes using them. The operating system loads the libraries into physical memory only once and maps the same physical pages into the address space of each process.

The operating system employs shared memory in several more cases. First, when forking a process, the memory is shared between the two processes. Only when the data is modified, the corresponding memory regions are copied. Second, a similar mechanism is used when starting another instance of an already running program. Third, it is also possible for user programs to request shared memory using system calls like mmap.

The operating system tries to unify these three categories. On Linux, mapping a program file or a shared library file as a read-only memory with mmap results in sharing memory with all these programs, respectively programs using the same shared library or program binary. This is also possible on Windows using the LoadLibrary function. Thus, even if a program is statically linked, its memory is shared with other programs which execute or map the same binary.

Another form of shared memory is content-based page deduplication. The hypervisor or operating system scans the physical memory for pages with identical content. All mappings to identical pages are redirected to one of the pages while the other pages are marked as free. Thus, memory is shared between completely unrelated processes and even between processes running in different virtual machines. When the data is modified by one process, memory is duplicated again. These examples demonstrate that code as well as static data can be shared among processes, even without their knowledge. Nevertheless, page deduplication can enhance system performance and besides the application in cloud systems, it is also relevant in smaller systems like smartphones.

User programs can retrieve information on their virtual and physical memory using operating-system services like `/proc/<pid>/maps` on Linux or tools like `vmmap` on Windows. The list of mappings typically includes all loaded shared-object files and the program binary.

2.3 Cache Attacks

Cache attacks are a specific type of side-channel attacks that exploit the effects of the cache memory on the execution time of algorithms. The first theoretical attacks were mentioned by Kocher [28] and Kelsey et al. [26]. Later on, practical attacks for DES were proposed by Page [41] as well as Tsunoo et al. [50]. In 2004, Bernstein [4] proposed the first time-driven cache attack against AES. This attack has been investigated quite extensively [36].

A more fine-grained attack has been proposed by Percival [42], who suggested to measure the time to access all ways of a cache set. As the access time correlates with the number of occupied cache ways, an attacker can determine the cache ways occupied by other processes. At the same time, Osvik et al. [39] proposed two fundamental techniques that allow an attacker to determine which specific cache sets have been accessed by a victim program. The first technique is Evict+Time, which consists of three steps. First, the victim program is executed and its execution time is measured. Afterwards, an attacker evicts one specific cache set and finally measures the execution time of the victim again. If the execution time increased, the cache set was probably accessed during the execution.

The second technique is Prime+Probe, which is similar to Percival's attack. During the Prime step, the attacker occupies specific cache sets. After the victim program has been scheduled, the Probe step is used to determine which cache sets are still occupied.

Later on, Gullasch et al. [16] proposed a significantly more powerful attack that exploits the fact that shared memory is loaded into the same cache sets for different processes. While Gullasch et al. attacked the L1 cache, Yarom and Falkner [55] presented an improvement called Flush+Reload that targets the L3 cache.

Flush+Reload relies on the availability of shared memory and especially shared libraries between the attacker and the victim program. An attacker constantly flushes a cache line using the `clflush` instruction on an address within the shared memory. After the victim has been scheduled, the attacker measures the time it takes to reaccess the same address again. The measured time reveals whether the data has been loaded into the cache by reaccessing it or whether the victim program loaded the data into the cache before reaccessing. This allows the attacker to determine the memory accesses of the victim process. As the L3 cache is shared among all cores, it is not necessary to constantly interrupt the victim process. Instead, both processes run on different cores while still working on the same L3 cache. Furthermore, the L3 cache is a unified inclusive cache and, thus, even allows to determine when a certain instruction is executed. Because of the size of the L3 cache, there are significantly fewer false negative cache-hit detections caused by evictions. Even though false positive cache-hit detections (as in Prime+Probe) are not possible because of the shared-memory-based approach, false positive cache hits can still occur if data is loaded into the cache accidentally (e.g., by the prefetcher). Nevertheless, applications of Flush+Reload have been shown to be quite reliable and powerful, for example, to detect specific versions of cryptographic libraries [23], to revive supposedly fixed attacks (e.g., Lucky 13) [24] as well as to improve attacks against T-table-based AES implementations [17].

As shared memory is not always available between different virtual machines in the cloud, more recent cache attacks use the Prime+Probe technique to perform cache attacks across virtual machine borders. For example, Irazoqui et al. [20] demonstrated a cross-VM attack on a T-Table-based AES implementation and Liu et al. [32] demonstrated a cross-VM attack on GnuPG. Both attacks require manual identification of exploitable code and data in targeted binaries. Similarly, Maurice et al. [33] built a cache-index-agnostic cross-VM covert channel based on Prime+Probe.

Simultaneous to our work, Oren et al. [38] developed a cache attack from within sandboxed JavaScript to attack user-specific data like network traffic or mouse movements. Contrary to existing attack approaches, we present a general attack framework to exploit cache vulnerabilities automatically. We demonstrate the effectiveness of this approach by inferring keystroke information and, for comparison reasons, by attacking a T-table-based AES implementation.

3 Cache Template Attacks

Chari et al. [8] presented template attacks as one of the strongest forms of side-channel attacks. First, side-channel traces are generated on a device controlled by the attacker. Based on these traces, the template—an exact model of signal and noise—is generated. A single side-channel trace from an identical device with unknown key is then iteratively classified using the template to derive the unknown key.

Similarly, Brumley and Hakala [7] described cache-timing template attacks to automatically analyze and exploit cache vulnerabilities. Their attack is based on Prime+Probe on the L1 cache and, thus, needs to run on the same core as the spy program. Furthermore, they describe a profiling phase for specific operations executed in the attacked binary, which requires manual work or even modification of the attacked software. In contrast, our attack only requires an attacker to know how to trigger specific events in order to attack them. Subsequently, Brumley and Hakala match these timing templates against the cache timing observed. In contrast, we match memory-access templates against the observed memory accesses.

Inspired by their work we propose *Cache Template Attacks*. The presented approach of Cache Template Attacks allows the exploitation of any cache vulnerability present in any program on any operating system executed on architectures with shared inclusive last-level caches and shared memory enabled. Cache Template Attacks consist of two phases: 1) a profiling phase, and 2) an exploitation phase. In the profiling phase, we compute a Cache Template matrix containing the cache-hit ratio on an address given a specific target event in the binary under attack. The exploitation phase uses this Cache Template matrix to infer events from cache hits.

Both phases rely on Flush+Reload and, thus, attack code and static data within binaries. In both phases the attacked binary is mapped into read-only shared memory in the attacker process. By accessing its own virtual addresses in the allocated read-only shared memory region, the attacker accesses the same physical memory and the same cache lines (due to the physically-indexed last level cache) as the process under attack. Therefore, the attacker completely bypasses address space layout randomization (ASLR). Also, due to shared memory, the additional memory consumption caused by the attacker process is negligible, i.e., in the range of a few megabytes at most.

In general, both phases are performed online on the attacked system and, therefore, cannot be prevented through differences in binaries due to different versions or the concept of software diversity [12]. However, if online profiling is not possible, e.g., in case the events must be triggered by a user or Flush+Reload is not possible on the attacked system, it can also be performed in a controlled environment. Below, we describe the profiling phase and the exploitation phase in more detail.

3.1 Profiling Phase

The profiling phase measures how many cache hits occur on a specific address during the execution of a specific event, i.e., the *cache-hit ratio*. The cache-hit ratios for different events are stored in the Cache Template matrix which has one column per event and one row per address. We refer to the column vector for an event as a *profile*. Examples of Cache Template matrices can be found in Section 4 and Section 5.1.

An *event* in terms of a Cache Template Attack can be anything that involves code execution or data accesses, e.g., low-frequency events, such as keystrokes or receiving an email, or high-frequency events, such as encryption with one or more key bits set to a specific value. To automate the profiling phase, it must be possible to trigger the event programmatically, e.g., by calling a function to simulate a keypress event, or executing a program.

The Cache Template matrix is computed in three steps. The first step is the generation of the cache-hit trace and the event trace. This is the main computation step of the Cache Template Attack, where the data for the Template is measured. In the second step, we extract the cache-hit ratio for each trace and store it in the Cache Template matrix. In a third post-processing step, we prune rows and columns which contain redundant information from the matrix. Algorithm 1 summarizes the profiling phase. We explain the corresponding steps in detail below.

Algorithm 1: Profiling phase.

```
Input: Set of events $E$, target program binary $B$,
       duration $d$
Output: Cache Template matrix $T$

Map binary $B$ into memory
foreach event $e$ in $E$ do
    foreach address $a$ in binary $B$ do
        while duration $d$ not passed do
            simultaneously
                Trigger event $e$ and save event trace $g_{a,e}^{(E)}$
                Flush+Reload attack on address $a$
                    and save cache-hit trace $g_{a,e}^{(H)}$
        end
        Extract cache-hit ratio $H_{a,e}$ from $g_{a,e}^{(E)}$
            and $g_{a,e}^{(H)}$ and store it in $T$
    end
end
Prune Cache Template matrix $T$
```

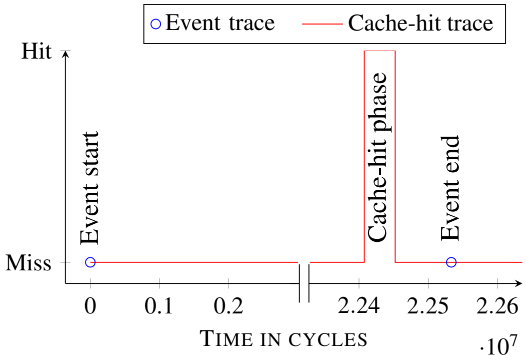


Figure 1: Trace of a single keypress event for address 0x4ebc0 of libgdk.so.

Cache-Hit Trace and Event Trace. The generation of the cache-hit trace and the event trace is repeated for each event and address for the specified duration (the while loop of Algorithm 1). The cache-hit trace $g_{a,e}^{(H)}$ is a binary function which has value 1 for every timestamp t where a cache hit has been observed. The function value remains 1 until the next timestamp t where a cache miss has been observed. We call subsequent cache hits a cache-hit phase. The event trace $g_{a,e}^{(E)}$ is a binary function which has value 1 when the processing of one specific event e starts or ends and value 0 for all other points.

In the measurement step, the binary under attack is executed and the event is triggered constantly. Each address of the attacked binary is profiled for a specific duration d. It must be long enough to trigger one or more events. Therefore, d depends only on the execution time of the event to be measured. The more events triggered within the specified duration d, the more accurate the resulting profile is. However, increasing the duration d increases the overall time required for the profiling phase.

The results of this measurement step are a cache-hit trace and an event trace, which are generated for all addresses a in the binary and all events e we want to profile. An excerpt of such a cache-hit trace and the corresponding event trace is shown in Figure 1. The start of the event is measured directly before the event is triggered. As we monitor library code, the cache-hit phase is measured before the attacked binary observes the event.

The generation of the traces can be sped up by two factors. First, in case of a cache miss, the CPU always fetches a whole cache line. Thus, we cannot distinguish between offsets of different accesses within a cache line and we can deduce the same information by probing only one address within each cache-line sized memory area.

Second, we reduce the overall number of triggered events by profiling multiple addresses at the same time. However, profiling multiple addresses on the same page can cause prefetching of more data from this page. Therefore, we can only profile addresses on different pages simultaneously. Thus, profiling all pages only takes as long as profiling a single page.

In case of low-frequency events, it is possible to profile all pages within one binary in parallel. However, this may lead to less accurate cache-hit traces $g_{a,e}^{(H)}$, i.e., timing deviations above 1 microsecond from the real event, which is only acceptable for low-frequency events.

Hit-Ratio Extraction. After the cache-hit trace and the event trace have been computed for a specific event e and a specific address a (the while loop of Algorithm 1), we derive the cache-hit ratio for each event and address. The cache-hit ratio $H_{a,e}$ is either a simple value or a time-dependent ratio function. In our case it is the ratio of cache hits on address a and the number of times the event e has been triggered within the profiling duration d.

To illustrate the difference between a cache-hit ratio with time dependency and without time dependency, we discuss two such functions. The cache-hit ratio with time dependency can be defined as follows. The event traces contain the start and end points of the processing of one event e. These start and end points define the relevant parts (denoted as *slices*) within the cache-hit trace. The slices are stored in a vector and scaled to the same length. Each slice contains a cache-hit pattern relative to the event e. If we average over this vector, we get the cache-hit ratio function for event e.

The second, much simpler approach is to define the cache-hit ratio without time dependency. In this case, we count the number of cache hits k on address a and divide it by the number of times n the event e has been triggered within the profiling duration d. That is, we define $H_{a,e} = \frac{k}{n}$. In case of a low-noise side channel and event detection through single cache hits, it is sufficient to use a simple hit-ratio extraction function.

Like the previous step, this step is repeated for all addresses a in the binary b and all events e to be profiled. The result is the full Cache Template matrix T. We denote the column vectors $\vec{p}_e$ as *profiles* for specific events.

Pruning. In the exploitation phase, we are limited regarding the number of addresses we can attack. Therefore, we want to reduce the number of addresses in the Cache Template. We remove redundant rows from the Cache Template matrix and merge events which cannot be distinguished based on their profiles $\vec{p}_e$.

As cache hits can be independent of an event, the measured cache-hit ratio on a specific address can be independent of the event, i.e., code which is always executed, frequent data accesses by threads running all the time, or code that is never executed and data that is never accessed. In order to be able to detect an event e, the set of events has to contain at least one event e' which does not include event e. For example, in order to be able to detect the event "user pressed key A" we need to profile at least one event where the user does not press key A.

The pruning happens in three steps on the matrix. First, the removal of all addresses that have a small difference between minimum and maximum cache-hit ratio for all events. Second, merging all similar columns (events) into one set of events, i.e., events that cannot be distinguished from each other are merged into one column. The similarity measure for this is, for example, based on a mean squared error (MSE) function. Third, the removal of redundant lines. These steps ensure that we select the most interesting addresses and also allows us to reduce the attack complexity by reducing the overall number of monitored addresses.

We measure the reliability of a cache-based side channel by true and false positives as well as true and false negatives. Cache hits that coincide with an event are counted as true positive and cache hits that do not coincide with an event as false positive. Cache misses which coincide with an event are counted as true negative and cache misses which do not coincide with an event as false negative. Based on these four values we can determine the accuracy of our Template, for instance, by computing the F-Score, which is defined as the harmonic mean of the cache-hit ratio and the positive predictive value (percentage of true positives of the total cache hits). High F-Score values show that we can distinguish the given event accurately by attacking a specific address. In some cases further lines can be pruned from the Cache Template matrix based on these measures. The true positive rate and the false positive rate for an event e can be determined by the profile $\vec{p}_e$ of e and the average over all profiles except e.

Runtime of the Profiling Phase. Measuring the cache-hit ratio is the most expensive step in our attack. To quantify the cost we give two examples. In both cases we want to profile a 1 MB library, once for a low-frequency event, e.g., a keypress, and once for a high-frequency event, e.g., an encryption. In both cases, we try to achieve a runtime which is realistic for offline and online attacks while maintaining a high accuracy.

We choose a profiling duration of $d = 0.8$ seconds for the low-frequency event. During 0.8 seconds we can trigger around 200 events, which is enough to create a highly accurate profile. Profiling each address in the library for 0.8 seconds would take 10 days. Profiling only cache-line-aligned addresses still takes 4 hours. Applying both optimizations, the full library is profiled in 17 seconds.

In case of the high-frequency event, we attack an encryption. We assume that one encryption and the corresponding Flush+Reload measurement take 520 cycles on average. As in the previous example, we profile each address 200 times and, thus, we need 40–50 microseconds per address, i.e., $d = 50\mu s$. The basic attack takes less than 55 seconds to profile the full library for one event. Profiling only cache-line-aligned addresses takes less than 1 second and applying both optimizations results in a negligible runtime.

As already mentioned above, the accuracy of the resulting profile depends on how many times an event can be triggered during profiling duration d. In both cases we chose durations which are more than sufficient to create accurate profiles and still achieve reasonable execution times for an online attack. Our observations showed that it is necessary to profile each event at least 10 times to get meaningful results. However, profiling an event more than a few hundred times does not increase the accuracy of the profile anymore.

3.2 Exploitation Phase

In the exploitation phase we execute a generic spy program which performs either the Flush+Reload or the Prime+Probe algorithm. For all addresses in the Cache Template matrix resulting from the profiling phase, the cache activity is constantly monitored.

We monitor all addresses and record whether a cache hit occurred. This information is stored in a boolean vector $\vec{h}$. To determine which event occurred based on this observation, we compute the similarity $S(\vec{h}, \vec{p}_e)$ between $\vec{h}$ and each profile $\vec{p}_e$ from the Cache Template matrix. The similarity measure S can be based, for example, on a mean squared error (MSE) function. Algorithm 2 summarizes the exploitation phase.

Algorithm 2: Exploitation phase.

```
Input: Target program binary $b$,
Cache Template matrix $T = (\vec{p}_{e_1}, \vec{p}_{e_2}, ..., \vec{p}_{e_n})$
Map binary $b$ into memory
repeat
    foreach address $a$ in $T$ do
        Flush+Reload attack on address $a$
        Store 0/1 in $\vec{h}[a]$ for cache miss/cache hit
    end
    if $\vec{p}_e$ equals $\vec{h}$ w.r.t. similarity measure then
        Event $e$ detected
    end
```

The exploitation phase has the same requirements as the underlying attack techniques. The attacker needs to be able to execute a spy program on the attacked system. In case of Flush+Reload, the spy program needs no privileges, except opening the attacked program binary in a read-only shared memory. It is even possible

```
int map[130][1024] = {{-1U},...,{-130U}};
int main(int argc, char** argv) {
  while(1) {
    int c = getchar(); // unbuffered
    if (map[(c % 128) + 1][0] == 0)
      exit(-1);
} }
```

Listing 1: Victim program with large array on Linux

to attack binaries running in a different virtual machine on the same physical machine, if the hypervisor has page deduplication enabled. In case of Prime+Probe, the spy program needs no privileges at all and it is even possible to attack binaries running in a different virtual machine on the same physical machine, as shown by Irazoqui et al. [20]. However, the Prime+Probe technique is more susceptible to noise and therefore the exploitation phase will produce less reliable results, making attacks on low-frequency events more difficult.

The result of the exploitation phase is a log file containing all detected events and their corresponding timestamps. The interpretation of the log file still has to be done manually by the attacker.

4 Attacks on Artificial Applications

Before we actually exploit cache-based vulnerabilities in real applications in Section 5, we demonstrate the basic working principle of Cache Template Attacks on two artificial victim programs. These illustrative attacks show how Cache Template Attacks automatically profile and exploit cache activity in any program. The two attack scenarios we demonstrate are: 1) an attack on lookup tables, and 2) an attack on executed instructions. Hence, our ideal victim program or library either contains a large lookup table which is accessed depending on secret information, e.g., depending on secret lookup indices, or specific portions of program code which are executed based on secret information.

Attack on Data Accesses. For demonstration purposes, we spy on simple events like keypresses. In our victim program, shown in Listing 1, each keypress causes a memory access in a large array called `map`. These key-based accesses are 4096 bytes apart from each other to avoid triggering the prefetcher. The array is initialized with static values in order to place it in the data segment and to guarantee that each page contains different data and, thus, is not deduplicated in any way. It is necessary to place it in the data segment in order to make it shareable with the spy program.

In the profiling phase of the Cache Template Attack, we simulate different keystroke events using the X11 automation library `libxdo`. This library can be linked statically into the spy program, i.e., it does not need to be installed. The Cache Template matrix is generated as described in Section 3. Within a duration of $d = 0.8$ seconds we simulated around 700 keypress events. The resulting Cache Template matrix can be seen in Figure 2 for all number keys. We observe cache hits on addresses that are exactly 4 096 bytes apart, which is due to the data type and the dimension of the `map` array. In our measurements, there were less than 0.3% false positive cache hits on the corresponding addresses and less than 2% false negative cache hits. The false positive and false negative cache hits are due to the high key rate in the keypress simulation.

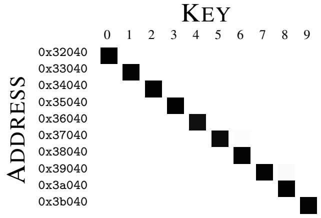


Figure 2: Cache Template matrix for the artificial victim program shown in Listing 1. Dark cells indicate high cache-hit ratios.

For verification purposes, we executed the generated keylogger for a period of 60 seconds and randomly pressed keys on the keyboard. In this setting we measured no false positives and no false negatives at all. This results from significantly lower key rates than in the profiling phase. The table is not used by any process other than the spy and the victim process and the probability that the array access happens exactly between the reload and the flush instruction is rather small, as we have longer idle periods than during the profiling phase. Thus, we are able to uniquely identify each key without errors.

Attack on Instruction Executions. The same attack can easily be performed on executed instructions. The source code for this example is shown in Listing 2. Each key is now processed in its own function, as defined by the `CASE(X)` macro. The functions are page aligned to avoid prefetcher activity. The `NOP1024` macro generates 1024 `nop` instructions, which is enough to avoid accidental code prefetching of function code.

Our measurements show that there is no difference between Cache Template Attacks on code and data accesses.

Performance Evaluation. To examine the performance limits of the exploitation phase of Cache Template Attacks, we evaluated the number of addresses which can

```
#define NOP1024 /*1024 times asm(”nop”);*/
#define CASE(X) case X:\
  { ALIGN(0x1000) void f##X() { NOP1024 };\
    f##X(); break; }
int main(int argc, char** argv) {
  while (1) {
    int c = getchar(); // unbuffered
    switch (c) {
      CASE(0);
      // ...
      CASE(128);
} } }
```

Listing 2: Victim program with long functions on Linux

be accurately monitored simultaneously at different key rates. At a key rate of 50 keys per second, we managed to spy on 16 000 addresses simultaneously on an Intel i5 Sandy Bridge CPU without any false positives or false negatives. The first errors occurred when monitoring 18 000 addresses simultaneously. At a key rate of 250 keys per second, which is the maximum on our system, we were able to spy on 4 000 addresses simultaneously without any errors. The first errors occurred when monitoring 5 000 addresses simultaneously. In both cases, we monitor significantly more addresses than in any practical cache attack today.

However, monitoring that many addresses is only possible if their position in virtual memory is such that the prefetcher remains inactive. Accessing several consecutive addresses on the same page causes prefetching of more data, resulting in cache hits although no program accessed the data. The limiting effect of the prefetcher on the Flush+Reload attack has already been observed by Yarom and Benger [54]. Based on these observations, we discuss the possibility of using the prefetcher as an effective countermeasure against cache attacks in Section 6.3.

5 Attacks on Real-World Applications

In this section, we consider an attack scenario where an attacker is able to execute an attack tool on a targeted machine in unprivileged mode. By executing this attack tool, the attacker extracts the cache-activity profiles which are exploited subsequently. Afterwards, the attacker collects the secret information acquired during the exploitation phase.

For this rather realistic and powerful scenario we present various case studies of attacks launched against real applications. We demonstrate the power of automatically launching cache attacks against any binary or library. First, we launch two attacks on Linux user interfaces, including GDK-based user interfaces, and an attack against a Windows user interface. In all attacks we simulate the user input in the profiling phase. Thus, the attack can be automated on the device under attack. To demonstrate the range of possible applications, we also present an automated attack on the T-table-based AES implementation of OpenSSL 1.0.2 [37].

5.1 Attack on Linux User Interfaces

There exists a variety of software-based side-channel attacks on user input data. These attacks either measure differences in the execution time of code in other programs or libraries [48], approximate keypresses through CPU and cache activity [44], or exploit system services leaking user input data [56]. In particular, Zhang et al. [56] use information about other processes from `procfs` on Linux to measure inter-keystroke timings and derive key sequences. Their proposed countermeasures can be implemented with low costs and prevent their attack completely. We, however, employ Cache Template Attacks to find and exploit leaking side-channel information in shared libraries automatically in order to spy on keyboard input.

Given root access to the system, it is trivial to write a keylogger on Linux using `/dev/input/event*` devices. Furthermore, the `xinput` tool can also be used to write a keylogger on Linux, but root access is required to install it. However, using our approach of Cache Template Attacks only requires the unprivileged execution of untrusted code as well as the capability of opening the attacked binaries or shared libraries in a read-only shared memory. In the exploitation phase one round of Flush+Reload on a single address takes less than 100 nanoseconds. If we measure the average latency between keypress and cache hit, we can determine the actual keypress timing up to a few hundred nanoseconds. Compared to the existing attacks mentioned above, our attack is significantly more accurate in terms of both event detection (detection rates near 100%) and timing deviations.

In all attacks presented in this section we compute time-independent cache-hit ratios.

Attack on the GDK Library. Launching the Cache Template profiling phase on different Linux applications revealed thousands of addresses in different libraries, binaries, and data files showing cache activity upon keypresses. Subsequently, we targeted different keypress events in order to find addresses distinguishing the different keys. Figure 3 shows the Cache Template of a memory area in the GDK library `libgdk-3.so.0.1000.8`, a part of the GTK framework which is the default user-interface framework on many Linux distributions.

Figure 3 shows several addresses that yield a cache hit with a high accuracy if and only if a certain key is

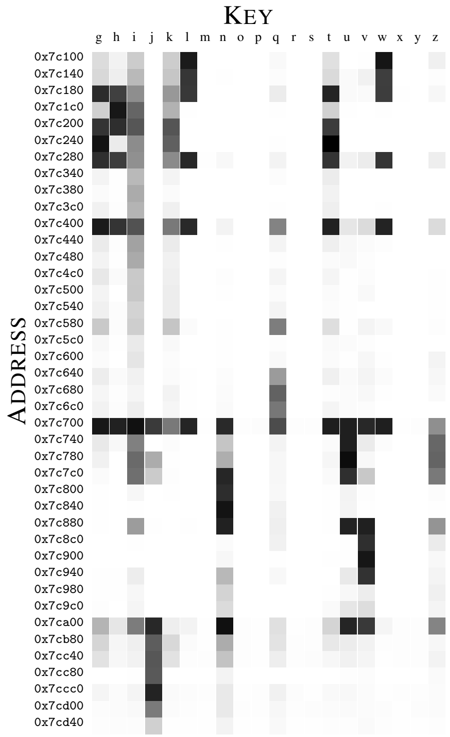


Figure 3: Excerpt of the GDK Cache Template. Dark cells indicate key-address-pairs with high cache-hit ratios.

pressed. For instance, every keypress on key `n` results in cache hit on address `0x7c800`, whereas the same address reacts in only 0.5% of our tests on other keypresses. Furthermore, we found a high cache-hit ratio on some addresses when a key is pressed (i.e., `0x6cd00` in `libgdk`), the mouse is moved (i.e., `0x28760` in `libgdk`) or a modifier key is pressed (i.e., `0x72fc0` in `libgdk`). We also profiled the range of keys a–f but it is omitted from Figure 3 because no high cache-hit ratios have been observed for the shown addresses.

We use the spy tool described in Section 3.2 in order to spy on events based on the Cache Template. We are able to accurately determine the following sets of pressed keys: $\{i\}$,$\{j\}$,$\{n\}$,$\{q\}$,$\{v\}$,$\{l,w\}$,$\{u,z\}$,$\{g,h,k,t\}$. That is, we cannot distinguish between keys in the same set, but keys in one set from keys in other sets. Similarly, we can deduce whether a key is contained in none of these sets.

Not as part of our attack, but in order to understand how keyboard input is processed in the GDK library, we analyzed the binary and the source code. In general, we found out that most of the addresses revealed in the profiling phase point to code executed while processing keyboard input. The address range discussed in this section contains the array `gdk_keysym_to_unicode_tab` which is used to translate key symbols to unicode special characters. The library performs a binary search on this array, which explains why we can identify certain keys accurately, namely the leaf nodes in the binary search.

As the corresponding array is used for keyboard input in all GDK user-interface components, including password fields, our spy tool works for all applications that use the GDK library. This observation allows us to use Cache Template Attacks to build powerful keyloggers for GDK-based user interfaces automatically. Even if we cannot distinguish all keys from each other, Cache Template Attacks allow us to significantly reduce the complexity of cracking a password. In this scenario, we are able to identify 3 keys reliably, as well as the total number of keypresses. Thus, in case of a lowercase password we can reduce the entropy per character from $\log_2(26) = 4.7$ to 4.0 bits. Attacking more than 3 addresses in order to identify more keys adds a significant amount of noise to the results, as it triggers the prefetcher. First experiments demonstrated the feasibility of attacking the lock screen of Linux distributions. However, further evaluation is necessary in order to reliably determine the effectiveness of this approach.

Attack on GDK Key Remapping. If an attacker has additional knowledge about the attacked system or software, more efficient and more powerful attacks are possible. Inspired by Tannous et al. [48] who performed a timing attack on GDK key remapping, we demonstrate a more powerful attack on the GDK library, by examining how the remapping of keys influences the sets of identifiable keypresses. The remapping functionality uses a large key-translation table `gdk_keys_by_keyval` which spreads over more than four pages.

Hence, we repeated the Cache Template Attack on the GDK library with a small modification. Before measuring cache activity for an address during an event, we remapped one key to the key code at that address, retrieved from the `gdk_keys_by_keyval` table. We found significant cache activity for some address and key-remapping combinations.

When profiling each key remapping for $d = 0.8$ seconds, we measured cache activity in 52 cache-line-sized memory regions. In verification scans, we found 0.2-2.5% false positive cache hits in these memory regions. Thus, we have found another highly accurate side channel for specific key remappings. The results are shown in the F-score graph in Figure 4. High values allow accurate detection of keypresses if the key is remapped to this address. Thus, we find more accurate results in terms of timing in our automated attack than Tannous et al. [48].

We can only attack 8 addresses in the profiled memory area simultaneously, since it spreads over 4 pages and we can only monitor 2 or 3 addresses without triggering the prefetcher. Thus, we are able to remap any 8

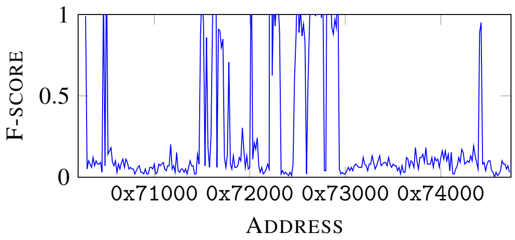


Figure 4: Excerpt of the F-score plot for the address range of the gdk_keys_by_keyval table. High values reveal addresses that can be exploited.

keys to these addresses and reliably distinguish them. In combination with the 3 addresses of our previous results, we are able to distinguish at least 11 keys and observe the timestamp of any keystroke in the system based on cache accesses simultaneously.

It is also possible to remap more than one key to the same key code. Hence, it is possible to distinguish between groups of keys. If we consider a lower-case password again, we can now reduce the entropy per character from $\log_2(26) = 4.7$ to 1.4 bits.

We also profiled keypresses on capslock and shift. Although we were able to log keypresses on both keys, we did not consider upper case or mixed case input. The exploitation phase automatically generates a log file containing the information observed through the cache side channel. However, interpretation of these results, such as deriving a program state from a sequence of events (shift key pressed or capslock active) and the influence of the program state on subsequent events is up to analysis of the results after the attack has been performed.

Tannous et al. [48] also described a login-detection mechanism in order to avoid remapping keys unless the user types in a password field. The spy program simply watches `/proc` to see whether a login program is running. Then the keys are remapped. As soon as the user pauses, the original key mappings are restored. The user will then notice a password mismatch, but the next password entry will work as expected.

Our completely automated password keylogger is a single binary which runs on the attacked system. It maps the GDK library into its own address space and performs the profiling phase. The profiling of each keypress requires the simulation of the keypress into a hidden window. Furthermore, some events require the key remapping we just described. Finally, the keylogger switches into the exploit mode. As soon as a logon screen is detected, for instance, after the screensaver was active or the screen was locked, the keys are remapped and all keypresses are logged into a file accessible by the attacker. Thus, all steps from the deployment of the keylogger to the final log file are fully automated.

5.2 Attacks on other Linux Applications

We also found leakage of accurate keypress timings in other libraries, such as the `ncurses` library (i.e., offset `0xbf90` in `libncurses.so`), and in files used to cache generated data related to user text input, such as `/usr/lib/locale/locale-archive`. The latter one is used to translate keypresses into the current locale. It is a generated file which differs on each system and which changes more frequently than the attacked libraries. In consequence, it is not possible to perform an offline attack, i.e., to use a pre-generated Cache Template in the exploitation phase on another system. Still, our concept of Cache Template Attacks allows us to perform an online attack, as profiling is fully automated by generating keystrokes through `libxdo` or comparable libraries. Thus, keystroke side channels are found within a few seconds of profiling. All keypress-timing side channels we found have a high accuracy and a timing deviation of less than 1 microsecond to the actual keypress.

In order to demonstrate Cache Template Attacks on a low-frequency event which is only indirectly connected to keypresses, we attacked `sshd`, trying to detect when input is sent over an active ssh connection. The received characters are unrelated to the local user input. When profiling for a duration of $d = 0.8$ seconds per address, we found 428 addresses showing cache activity when a character was received. We verified these results for some addresses manually. None of these checked addresses showed false positive hits within a verification period of 60 seconds. Thus, by exploiting the resulting Cache Template matrix, we are able to gain accurate timings for the transmitted characters (significantly less than 1 microsecond deviation to the transmission of the character). These timings can be used to derive the transmitted letters as shown by Zhang et al. [56].

5.3 Attack on Windows User Interfaces

We also performed Cache Template Attacks on Windows applications. The attack works on Windows using MinGW identically to Linux. Even the implementation is the same, except for the keystroke simulation which is now performed using the Windows API instead of the `libxdo` library, and the file under attack is mapped using `LoadLibrary` instead of `mmap`. We performed our attack on Windows 7 and Windows 8.1 systems with the same results on three different platforms, namely Intel Core 2 Duo, Intel i5 Sandy Bridge, and Intel i5 Ivy Bridge. As in the attacks on Linux user interfaces, address space layout randomization has been activated during both profiling and exploitation phase.

In an automated attack, we found cache activity upon keypresses in different libraries with reasonable accuracy. For instance, the Windows 7 common control library `comctl32.dll` can be used to detect keypresses on different addresses. Probing `0xc5c40` results in cache hits on every keypress and mouse click within text fields accurately. Running the generated keypress logger in a verification period of 60 seconds with keyboard input by a real user, we found only a single false positive event detection based on this address. Address `0xc6c00` reacts only on keypresses and not on mouse clicks, but yields more false positive cache hits in general. Again, we can apply the attack proposed by Zhang et al. [56] to recover typed words from inter-keystroke timings.

We did not disassemble the shared library and therefore do not know which function or data accesses cause the cache hit. The addresses were found by starting the Cache Template Attack with the same parameters as on Linux, but on a Windows shared library instead of a Linux shared library. As modern operating systems like Windows 7 and Windows 8.1 employ an immense number of shared libraries, we profiled only a few of these libraries. Hence, further investigations might even reveal addresses for a more accurate identification of keypresses.

5.4 Attack on a T-table-based AES

Cache attacks have been shown to enable powerful attacks against cryptographic implementations. Thus, appropriate countermeasures have already been suggested for the case of AES [15, 25, 30, 43]. Nevertheless, in order to compare the presented approach of Cache Template Attacks to related attacks, we launched an efficient and automated access-driven attack against the AES T-table implementation of OpenSSL 1.0.2, which is known to be insecure and susceptible to cache attacks [2, 4, 5, 16, 21, 22, 39, 53]. Recall that the T-tables are accessed according to the plaintext p and the secret key k, i.e., $T_j[p_i \oplus k_i]$ with $i \equiv j \mod 4$ and $0 \leq i < 16$, during the first round of the AES encryption. For the sake of brevity, we omit the full details of an access-driven cache attack against AES and refer the interested reader to the work of Osvik et al. [39, 49].

Attack of Encryption Events. In a first step, we profiled the two events "no encryption" and "encryption with random key and random plaintext". We profiled each cache-line-aligned address in the OpenSSL library during 100 encryptions. On our test system, one encryption takes around 320 cycles, which is very fast compared to a latency of at least 200 cycles caused by a single cache miss. In order to make the results more deterministically reproducible, we measure whether a cache line was used only after the encryption has finished. Thus, the profiling phase does not run in parallel and only one cache hit or miss is measured per triggered event.

This profiling step takes less than 200 seconds. We detected cache activity on 0.2%-0.3% of the addresses. Only 82 addresses showed a significant difference in cache activity depending on the event. For 18 of these addresses, the cache-hit ratio was 100% for the encryption event. Thus, our generated spy tool is able to accurately detect whenever an encryption is performed.

For the remaining 64 addresses the cache-hit ratio was around 92% for the encryption event. Thus, not each of these addresses is accessed in every encryption, depending on key and plaintext. Since we attack a T-table-based AES implementation, we know that these 64 addresses must be the T-tables, which occupy 4 KB respectively 64 cache lines. Although this information is not used in the first generated spy tool, it encourages performing a second attack to target specific key-byte values.

Attack on Specific Key-Byte Values. Exploiting the knowledge that we attack a T-table implementation, we enhance the attack by profiling over different key-byte values for a fixed plaintext, i.e., the set of events consists of the different key-byte values. Our attack remains fully automated, as we change only the values with which the encryption is performed. The result is again a log file containing the accurate timestamp of each event monitored. The interpretation of the log file, of course, involves manual work and is specific to the targeted events, i.e., key bytes in this case.

For each key byte k_i, we profile only the upper 4 bits of k_i as the lower 4 bits cannot be distinguished because of the cache-line size of 64 bytes. This means that we need to profile only 16 addresses for each key byte k_i. Furthermore, on average 92% of these addresses are already in the cache and the Reload step of the Flush+Reload attack is unlikely to trigger the prefetcher. Thus, we can probe all addresses after a single encryption. Two profiles for different values of k_0 are shown in Figure 5. The two traces were generated using 1 000 encryptions per key byte and address to show the pattern more clearly. According to Osvik et al. [39] and Spreitzer et al. [46] these plots (or patterns) reveal at least the upper 4 bits of a key byte and, hence, attacking the AES T-table implementation works as expected. In our case, experiments showed that 1 to 10 encryptions per key byte are enough to infer these upper 4 bits correctly.

In a T-table-based AES implementation, the index of the T-table is determined by $p_i \oplus k_i$. Therefore, the same profiles can be generated by iterating over the different plaintext byte values while encrypting with a fixed key. Osvik et al. [39] show a similar plot, generated using the Evict+Time attack. However, in our attack the profiles are aggregated into the Cache Template matrix, as de-

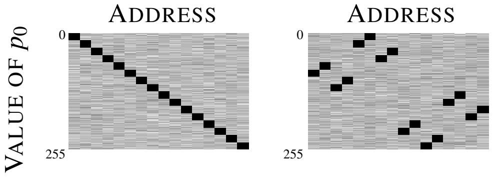


Figure 5: Excerpt of the Cache Template (address range of the first T-table). The plot is transposed to match [39]. In the left trace $k_0 =$ `0x00`, in the right trace $k_0 =$ `0x51`.

scribed in Section 3.1.

In the exploitation phase, the automatically generated spy tool monitors cache hits on the addresses from the Cache Template in order to determine secret key-byte values. We perform encryptions using chosen plaintexts. We attack the 16 key bytes k_i sequentially. In each step $i = 0,\dots,15$, the plaintext is random, except for the upper 4 bits of p_i, which are fixed to the same chosen value as in the profiling phase. Hence, the encryption is performed over a chosen plaintext. The spy tool triggers an encryption, detects when the encryption actually happens and after each encryption, reports the set of possible values for the upper 4 bits of key byte k_i. As soon as only one candidate for the upper 4 bits of key byte k_i remains, we continue with the next key byte.

Using Cache Template Attacks, we are able to infer 64 bits of the secret key with only 16–160 encryptions in a chosen-plaintext attack. Compared to the work of Osvik et al. [39] who require several hundred or thousands encryptions (depending on the measurement approach) targeting the L1 cache, and the work of Spreitzer and Plos [46] who require millions of encryptions targeting the L1 cache on the ARM platform, we clearly observe a significant performance improvement. More recent work shows that full key recovery is possible with less than 30 000 encryptions [17] using Flush+Reload.

The benefit of our approach, compared to existing cache attacks against AES, is that our attack is fully automated. Once the binary is deployed on the target system, it performs both profiling and exploitation phase automatically and finally returns a log file containing the key byte candidates to the attacker. Moreover, we do not need prior knowledge of the attacked system or the attacked executable or library.

AES T-table implementations are already known to be insecure and countermeasures have already been integrated, e.g., in the AES implementation of OpenSSL. Performing our attack on a non-T-table implementation (e.g., by employing AES-NI instructions) did not show key dependent information leakage, but still, we can accurately determine the start and end of the encryption through the cache behavior. However, we leave it as an interesting open issue to employ the presented approach of cache template attacks for further investigations of vulnerabilities in already protected implementations.

Trace-Driven Attack on AES. When attacking an insecure implementation of a cryptographic algorithm, an attacker can often gain significantly more information if it is possible to perform measurements during the encryption [2, 13], i.e., in case the exact trace of cache hits and cache misses can be observed. Even if we cannot increase the frequency of the Flush+Reload attack, we are able to slow down the encryption by constantly flushing the 18 addresses which showed cache activity in every profile. We managed to increase the encryption time from 320 cycles to 16 000–20 000 cycles. Thus, a more fine-grained trace of cache hits and cache misses can be obtained which might even allow the implementation of trace-driven cache attacks purely in software.

6 Countermeasures

We have demonstrated in Section 5 that Cache Template Attacks are applicable to real-world applications without knowledge of the system or the application. Therefore, we emphasize the need for research on effective countermeasures against cache attacks. In Section 6.1, we discuss several countermeasures which have been proposed so far. Subsequently, in Section 6.2, we discuss how Cache Template Attacks can be employed by developers to detect and eliminate cache-based information leakage and also by users to detect and prevent cache attacks running actively on a system. Finally, in Section 6.3, we propose changes to the prefetcher to build a powerful countermeasure against cache attacks.

6.1 Discussion of Countermeasures

Removal of the clflush Instruction is not Effective. The restriction of the `clflush` instruction has been suggested as a possible countermeasure against cache attacks in [54, 55, 58]. However, by adapting our spy tool to evict the cache line without using the `clflush` instruction (Evict+Reload instead of Flush+Reload), we demonstrate that this countermeasure is not effective at all. Thereby, we show that cache attacks can be launched successfully even without the `clflush` instruction.

Instead of using the `clflush` instruction, the eviction is done by accessing physically congruent addresses in a large array which is placed in large pages by the operating system. In order to compute physically congruent addresses we need to determine the lowest 18 bits of the physical address to attack, which can then be used to evict specific cache sets.

The actual mapping of virtual to physical addresses can be retrieved from `/proc/self/pagemap`. Even if such a mapping is not available, methods to find congruent addresses have been developed—simultaneously to this work—by Irazoqui et al. [20] by exploiting large pages, Oren et al. [38] by exploiting timing differences in JavaScript, and Liu et al. [32] by exploiting timing differences in native code.

The removal of the clflush instruction has also been discussed as a countermeasure to protect against DRAM disturbance errors (denoted as rowhammer bug). These disturbance errors have been studied by Kim et al. [27] and, later on, exploited by Seaborn et al. [45] to gain kernel privileges. Several researchers have already claimed to be able to exploit the rowhammer bug without the clflush instruction [14], This can be done by exploiting the Sandy Bridge cache mapping function, which has been reverse engineered by Hund et al. [18], to find congruent addresses.

Our eviction strategy only uses the lowest 18 bits and therefore, we need more than 12 accesses to evict a cache line. With 48 accessed addresses, we measured an eviction rate close to 100%. For performance reasons we use write accesses, as the CPU does not have to wait for data fetches from the physical memory. In contrast to the clflush instruction, which takes only 41 cycles, our eviction function takes 325 cycles. This is still fast enough for most Flush+Reload attacks.

While clflush always evicts the cache line, our eviction rate is only near 100%. Therefore, false positive cache hits occur if the line has not been evicted. Using Flush+Reload, there is a rather low probability for a memory access on the monitored address to happen exactly between the Reload step and the point where the clflush takes effect. This probability is much higher in the case of Evict+Reload, as the eviction step takes 8 times longer than the clflush instruction.

We compare the accuracy of Evict+Reload to Flush+Reload using previously found cache vulnerabilities. For instance, as described in Section 5.1, probing address 0x7c800 of libgdk-3.so.0.1000.8 allows us to detect keypresses on key n. The Flush+Reload spy tool detects on average 98% of the keypresses on key n with a 2% false positive rate (keypresses on other keys). Using Evict+Reload, we still detect 90% of the keypresses on key n with a 5% false positive rate. This clearly shows that the restriction of clflush is not sufficient to prevent this type of cache attack.

Disable Cache-Line Sharing. One prerequisite of Flush+Reload attacks is shared memory. In cloud scenarios, shared memory across virtual machine borders is established through page deduplication. Page deduplication between virtual machines is commonly disabled in order to prevent more coarse-grained attacks like fingerprinting operating systems and files [40, 47] as well as Flush+Reload. Still, as shown by Irazoqui et al. [20], it is possible to use Prime+Probe as a fallback. However, attacking low-frequency events like keypresses becomes infeasible, because Prime+Probe is significantly more susceptible to noise.

Flush+Reload can also be prevented on a system by preventing cache-line sharing, i.e., by disabling shared memory. Unfortunately, operating systems make heavy use of shared memory, and without modifying the operating system it is not possible for a user program to prevent its own memory from being shared with an attacker, even in the case of static linkage as discussed in Section 2.2.

With operating-system modifications, it would be possible to disable shared memory in all cases where a victim program cannot prevent an attack, i.e., shared program binaries, shared libraries, shared generated files (for instance, locale-archive). Furthermore, it would be possible to provide a system call to user programs to mark memory as "do-not-share."

A hardware-based approach is to change cache tags. Virtually tagged caches are either invalidated on context switches or the virtual tag is combined with an address space identifier. Therefore, shared memory is not shared in the cache. Thus, Flush+Reload is not possible on virtually tagged caches.

We emphasize that as long as shared cache lines are available to an attacker, Flush+Reload or Evict+Reload cannot be prevented completely.

Cache Set Associativity. Prime+Probe, Evict+Time and Evict+Reload exploit set-associative caches. In all three cases, it is necessary to fill all ways of a cache set, either for eviction or for the detection of evicted cache sets. Based on which cache set was reloaded (respectively evicted), secret information is deduced. Fully associative caches have better security properties, as such information deduction is not possible and cache eviction can only be enforced by filling the whole cache. However, a timing attack would still be possible, e.g., due to internal cache collisions [5] leading to different execution times. As fully associative caches are impractical for larger caches, new cache architectures have been proposed to provide similar security properties [29, 51, 52]. However, even fully associative caches only prevent attacks which do not exploit cache-line sharing. Thus, a combination of countermeasures is necessary to prevent most types of cache attacks.

6.2 Proactive Prevention of Cache Attacks

Instrumenting cache attacks to detect co-residency [57] with another virtual machine on the same physical machine, or even to detect cache attacks [58] and cache-based side channels in general [11] has already been proposed in the past. Moreover, Brumley and Hakala [7] even suggested that developers should use their attack technique to detect and eliminate cache vulnerabilities in their programs. Inspired by these works, we present defense mechanisms against cache attacks which can be improved by using Cache Template Attacks.

Detect Cache Vulnerabilities as a Developer. Similar to Brumley and Hakala [7], we propose the employment of Cache Template Attacks to find cache-based vulnerabilities automatically. Compared to [7], Cache Template Attacks allow developers to detect potential cache side channels for specifically chosen events automatically, which can subsequently be fixed by the developer. A developer only needs to select the targeted events (e.g., keystrokes, window switches, or encryptions) and to trigger these events automatically during the profiling phase, which significantly eases the evaluation of cache side channels. Ultimately, our approach even allows developers to find such cache vulnerabilities in third party libraries.

Detect and Impede Ongoing Attacks as a User. Zhang et al. [58] stated the possibility to detect cache attacks by performing a cache attack on one of the vulnerable addresses or cache sets. We propose running a Cache Template Attack as a system service to detect code and data under attack. If Flush+Reload prevention is sufficient, we simply disable page sharing for all pages with cache lines under attack. Otherwise, we disable caching for these pages as proposed by Aciiçmez et al. [1] and, thus, prevent all cache attacks. Only the performance for critical code and data parts is reduced, as the cache is only disabled for specific pages in virtual memory.

Furthermore, cache attacks can be impeded by performing additional memory accesses, unrelated to the secret information, or random cache flushes. Such obfuscation methods on the attacker's measurements have already been proposed by Zhang et al. [59]. The idea of the proposed obfuscation technique is to generate random memory accesses, denoted as cache cleansing. However, it does not address the shared last-level cache. In contrast, Cache Template Attacks can be used to identify possible cache-based information leaks and then to specifically add noise to these specific locations by accessing or flushing the corresponding cache lines.

6.3 Enhancing the Prefetcher

During our experiments, we found that the prefetcher influences the cache activity of certain access patterns during cache attacks, especially due to the spatial locality of addresses, as also observed in other work [16, 39, 54]. However, we want to discuss the prefetcher in more detail as it is crucial for the success of a cache attack.

Although the profiling phase of Cache Template Attacks is not restricted by the prefetcher, the spy program performing the exploitation phase might be unable to probe all leaking addresses simultaneously. For instance, we found 255 addresses leaking side-channel information about keypresses in the GDK library but we were only able to probe 8 of them simultaneously in the exploitation phase, because the prefetcher loads multiple cache lines in advance and, thus, generates numerous false positive cache hits.

According to the Intel 64 and IA-32 Architectures Optimization Reference Manual [19], the prefetcher loads multiple memory addresses in advance if "two cache misses occur in the last level cache" and the corresponding memory accesses are within a specific range (the so-called trigger distance). Depending on the CPU model this range is either 256 or 512 bytes, but does not exceed a page boundary of 4 KB. Due to this, we are able to probe at least 2 addresses per page.

We suggest increasing the trigger distance of the prefetcher beyond the 4 KB page boundary if the corresponding page already exists in the translation lookaside buffer. The granularity of the attack will then be too high for many practical targets, especially attacks on executed instructions will then be prevented.

As cache attacks constantly reaccess specific memory locations, another suggestion is to adapt the prefetcher to take temporal spatiality into consideration. If the prefetcher were to prefetch data based on that temporal distance, most existing attacks would be prevented.

Just as we did in Section 4, an attacker might still be able to establish a communication channel targeted to circumvent the prefetcher. However, the presented countermeasures would prevent most cache attacks targeting real-world applications.

7 Conclusion

In this paper, we introduced Cache Template Attacks, a novel technique to find and exploit cache-based side channels easily. Although specific knowledge of the attacked machine and executed programs or libraries helps, it is not required for a successful attack. The attack is performed on closed-source and open-source binaries in exactly the same way.

We studied various applications of Cache Template Attacks. Our results show that an attacker is able to infer highly accurate keystroke timings on Linux as well as Windows. For Linux distributions we even demonstrated a fully automatic keylogger that significantly reduces the entropy of passwords. Hence, we conclude that cache-based side-channel attacks are an even greater threat for today's computer architectures than assumed so far. In fact, even sensitive user input, like passwords, cannot be considered secure on machines employing CPU caches.

We argue that fundamental concepts of computer architectures and operating systems enable the automatic exploitation of cache-based vulnerabilities. We observed that many of the existing countermeasures do not prevent such attacks as expected. Still, the combination of multiple countermeasures can effectively mitigate cache attacks. However, the fact that cache attacks can be launched automatically marks a change of perspective, from a more academic interest towards practical attacks, which can be launched by less sophisticated attackers. This shift emphasizes the need to develop and integrate effective countermeasures immediately. In particular, it is not sufficient to protect only specific cryptographic algorithms like AES. More general countermeasures will be necessary to counter the threat of automated cache attacks.

8 Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Ben Ransford, for their valuable comments and suggestions. The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644052 (HECTOR).

Furthermore, this work has been supported by the Austrian Research Promotion Agency (FFG) and the Styrian Business Promotion Agency (SFG) under grant number 836628 (SeCoS).

References

[1] Aciiçmez, O., Brumley, B. B., and Grabher, P. New Results on Instruction Cache Attacks. In *Cryptographic Hardware and Embedded Systems – CHES* (2010), vol. 6225 of *LNCS*, Springer, pp. 110–124.

[2] Aciiçmez, O., and Koç, Ç. K. Trace-Driven Cache Attacks on AES (Short Paper). In *International Conference on Information and Communications Security – ICICS* (2006), vol. 4307 of *LNCS*, Springer, pp. 112–121.

[3] Benger, N., van de Pol, J., Smart, N. P., and Yarom, Y. "Ooh Aah... Just a Little Bit" : A Small Amount of Side Channel Can Go a Long Way. In *Cryptographic Hardware and Embedded Systems – CHES* (2014), vol. 8731 of *LNCS*, Springer, pp. 75–92.

[4] Bernstein, D. J. Cache-Timing Attacks on AES, 2004. URL: `http://cr.yp.to/papers.html#cachetiming`.

[5] Bogdanov, A., Eisenbarth, T., Paar, C., and Wienecke, M. Differential Cache-Collision Timing Attacks on AES with Applications to Embedded CPUs. In *Topics in Cryptology – CT-RSA* (2010), vol. 5985 of *LNCS*, Springer, pp. 235–251.

[6] Bonneau, J., and Mironov, I. Cache-Collision Timing Attacks Against AES. In *Cryptographic Hardware and Embedded Systems – CHES* (2006), vol. 4249 of *LNCS*, Springer, pp. 201–215.

[7] Brumley, B. B., and Hakala, R. M. Cache-Timing Template Attacks. In *Advances in Cryptology – ASIACRYPT* (2009), vol. 5912 of *LNCS*, Springer, pp. 667–684.

[8] Chari, S., Rao, J. R., and Rohatgi, P. Template Attacks. In *Cryptographic Hardware and Embedded Systems – CHES* (2002), vol. 2523 of *LNCS*, Springer, pp. 13–28.

[9] Chen, C., Wang, T., Kou, Y., Chen, X., and Li, X. Improvement of Trace-Driven I-Cache Timing Attack on the RSA Algorithm. *Journal of Systems and Software 86*, 1 (2013), 100–107.

[10] Daemen, J., and Rijmen, V. *The Design of Rijndael: AES – The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[11] Doychev, G., Feld, D., Köpf, B., Mauborgne, L., and Reineke, J. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *USENIX Security Symposium* (2013), USENIX Association, pp. 431–446.

[12] Franz, M. E unibus pluram: Massive-Scale Software Diversity as a Defense Mechanism. In *Workshop on New Security Paradigms – NSPW* (2010), ACM, pp. 7–16.

[13] Gallais, J., Kizhvatov, I., and Tunstall, M. Improved Trace-Driven Cache-Collision Attacks against Embedded AES Implementations. *IACR Cryptology ePrint Archive 2010/408*.

[14] Google Groups. Rowhammer without CLFLUSH, 2015. URL: `https://groups.google.com/forum/#!topic/rowhammer-discuss/ojgTgLr4q_M`.

[15] Gueron, S. White Paper: Intel Advanced Encryption Standard (AES) Instructions Set, 2010. URL: `https://software.intel.com/file/24917`.

[16] Gullasch, D., Bangerter, E., and Krenn, S. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *IEEE Symposium on Security and Privacy – S&P* (2011), IEEE Computer Society, pp. 490–505.

[17] Gülmezoğlu, B., Inci, M. S., Eisenbarth, T., and Sunar, B. A Faster and More Realistic Flush+Reload Attack on AES. In *Constructive Side-Channel Analysis and Secure Design – COSADE* (2015), LNCS, Springer. In press.

[18] Hund, R., Willems, C., and Holz, T. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *IEEE Symposium on Security and Privacy – SP* (2013), IEEE Computer Society, pp. 191–205.

[19] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. No. 248966-026. 2012.

[20] Irazoqui, G., Eisenbarth, T., and Sunar, B. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES. In *IEEE Symposium on Security and Privacy – S&P* (2015), IEEE Computer Society.

[21] Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Fine grain Cross-VM Attacks on Xen and VMware are possible! *IACR Cryptology ePrint Archive 2014/248*.

[22] Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Wait a Minute! A fast, Cross-VM Attack on AES. In *Research in Attacks, Intrusions and Defenses Symposium – RAID* (2014), vol. 8688 of *LNCS*, Springer, pp. 299–319.

[23] Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Know Thy Neighbor: Crypto Library Detection in Cloud. *Privacy Enhancing Technologies 1*, 1 (2015), 25–40.

[24] Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Lucky 13 Strikes Back. In *ACM ASIA CCS* (2015), pp. 85–96.

[25] KÄSPER, E., AND SCHWABE, P. Faster and Timing-Attack Resistant AES-GCM. In *Cryptographic Hardware and Embedded Systems – CHES* (2009), vol. 5747 of *LNCS*, Springer, pp. 1–17.

[26] KELSEY, J., SCHNEIER, B., WAGNER, D., AND HALL, C. Side Channel Cryptanalysis of Product Ciphers. *Journal of Computer Security 8*, 2/3 (2000), 141–158.

[27] KIM, Y., DALY, R., KIM, J., FALLIN, C., LEE, J., LEE, D., WILKERSON, C., LAI, K., AND MUTLU, O. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *ACM/IEEE International Symposium on Computer Architecture – ISCA* (2014), IEEE Computer Society, pp. 361–372.

[28] KOCHER, P. C. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology – CRYPTO* (1996), vol. 1109 of *LNCS*, Springer, pp. 104–113.

[29] KONG, J., ACIIÇMEZ, O., SEIFERT, J., AND ZHOU, H. Deconstructing New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *ACM Workshop on Computer Security Architecture – CSAW* (2008), pp. 25–34.

[30] KÖNIGHOFER, R. A Fast and Cache-Timing Resistant Implementation of the AES. In *Topics in Cryptology – CT-RSA* (2008), vol. 4964 of *LNCS*, Springer, pp. 187–202.

[31] LIU, F., AND LEE, R. B. Random Fill Cache Architecture. In *International Symposium on Microarchitecture – MICRO* (2014), IEEE, pp. 203–215.

[32] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. B. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy – S&P* (2015).

[33] MAURICE, C., NEUMANN, C., HEEN, O., AND FRANCILLON, A. C5: Cross-Cores Cache Covert Channel. In *DIMVA* (2015). In press.

[34] MOWERY, K., KEELVEEDHI, S., AND SHACHAM, H. Are AES x86 Cache Timing Attacks Still Feasible? In *Workshop on Cloud Computing Security – CCSW* (2012), ACM, pp. 19–24.

[35] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. Advanced Encryption Standard. NIST FIPS PUB 197, 2001.

[36] NEVE, M. *Cache-based Vulnerabilities and SPAM Analysis*. PhD thesis, UCL, 2006.

[37] OPENSSL SOFTWARE FOUNDATION. OpenSSL Project, 2014. URL: `http://www.openssl.org/`.

[38] OREN, Y., KEMERLIS, V. P., SETHUMADHAVAN, S., AND KEROMYTIS, A. D. The Spy in the Sandbox - Practical Cache Attacks in Javascript. *CoRR abs/1502.07373* (2015).

[39] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology – CT-RSA* (2006), vol. 3860 of *LNCS*, Springer, pp. 1–20.

[40] OWENS, R., AND WANG, W. Non-Interactive OS Fingerprinting Through Memory De-Duplication Technique in Virtual Machines. In *International Performance Computing and Communications Conference – IPCCC* (2011), IEEE, pp. 1–8.

[41] PAGE, D. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. *IACR Cryptology ePrint Archive 2002/169*.

[42] PERCIVAL, C. Cache Missing for Fun and Profit, 2005. URL: `http://www.daemonology.net/hyperthreading-considered-harmful/`.

[43] REBEIRO, C., SELVAKUMAR, A. D., AND DEVI, A. S. L. Bitslice Implementation of AES. In *Cryptology and Network Security – CANS* (2006), vol. 4301 of *LNCS*, Springer, pp. 203–212.

[44] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. 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), ACM, pp. 199–212.

[45] SEABORN, M., AND DULLIEN, T. Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges, 2015. URL: `http://googleprojectzero.blogspot.co.at/2015/03/exploiting-dram-rowhammer-bug-to-gain.html`.

[46] SPREITZER, R., AND PLOS, T. Cache-Access Pattern Attack on Disaligned AES T-Tables. In *Constructive Side-Channel Analysis and Secure Design – COSADE* (2013), vol. 7864 of *LNCS*, Springer, pp. 200–214.

[47] SUZAKI, K., IIJIMA, K., YAGI, T., AND ARTHO, C. Memory Deduplication as a Threat to the Guest OS. In *European Workshop on System Security – EUROSEC* (2011), ACM, pp. 1–6.

[48] TANNOUS, A., TROSTLE, J. T., HASSAN, M., MCLAUGHLIN, S. E., AND JAEGER, T. New Side Channels Targeted at Passwords. In *Annual Computer Security Applications Conference – ACSAC* (2008), pp. 45–54.

[49] TROMER, E., OSVIK, D. A., AND SHAMIR, A. Efficient Cache Attacks on AES, and Countermeasures. *Journal Cryptology 23*, 1 (2010), 37–71.

[50] TSUNOO, Y., SAITO, T., SUZAKI, T., SHIGERI, M., AND MIYAUCHI, H. Cryptanalysis of DES Implemented on Computers with Cache. In *Cryptographic Hardware and Embedded Systems – CHES* (2003), vol. 2779 of *LNCS*, Springer, pp. 62–76.

[51] WANG, Z., AND LEE, R. B. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *International Symposium on Computer Architecture – ISCA* (2007), pp. 494–505.

[52] WANG, Z., AND LEE, R. B. A Novel Cache Architecture with Enhanced Performance and Security. In *IEEE/ACM International Symposium on Microarchitecture – MICRO* (2008), pp. 83–93.

[53] WEISS, M., HEINZ, B., AND STUMPF, F. A Cache Timing Attack on AES in Virtualization Environments. In *Financial Cryptography and Data Security – FC* (2012), vol. 7397 of *LNCS*, Springer, pp. 314–328.

[54] YAROM, Y., AND BENGER, N. Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack. *IACR Cryptology ePrint Archive 2014/140*.

[55] YAROM, Y., AND FALKNER, K. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium* (2014), USENIX Association, pp. 719–732.

[56] ZHANG, K., AND WANG, X. Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems. In *USENIX Security Symposium* (2009), USENIX Association, pp. 17–32.

[57] ZHANG, Y., JUELS, A., OPREA, A., AND REITER, M. K. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In *IEEE Symposium on Security and Privacy – S&P* (2011), IEEE Computer Society, pp. 313–328.

[58] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *ACM Conference on Computer and Communications Security – CCS* (2014), ACM, pp. 990–1003.

[59] ZHANG, Y., AND REITER, M. K. Düppel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud. In *ACM Conference on Computer and Communications Security – CCS* (2013), ACM, pp. 827–838.

A Placement Vulnerability Study in Multi-Tenant Public Clouds

Venkatanathan Varadarajan[†], Yinqian Zhang[‡], Thomas Ristenpart[*], and Michael Swift[†]

[†]*University of Wisconsin-Madison*, [‡]*The Ohio State University*, [*]*Cornell Tech*,
{venkatv,swift}@cs.wisc.edu, yinqian@cse.ohio-state.edu, ristenpart@cornell.edu

Abstract

Public infrastructure-as-a-service clouds, such as Amazon EC2, Google Compute Engine (GCE) and Microsoft Azure allow clients to run virtual machines (VMs) on shared physical infrastructure. This practice of multi-tenancy brings economies of scale, but also introduces the risk of sharing a physical server with an arbitrary and potentially malicious VM. Past works have demonstrated how to place a VM alongside a target victim (co-location) in early-generation clouds and how to extract secret information via side-channels. Although there have been numerous works on side-channel attacks, there have been no studies on placement vulnerabilities in public clouds since the adoption of stronger isolation technologies such as Virtual Private Clouds (VPCs).

We investigate this problem of placement vulnerabilities and quantitatively evaluate three popular public clouds for their susceptibility to co-location attacks. We find that adoption of new technologies (e.g., VPC) makes many prior attacks, such as cloud cartography, ineffective. We find new ways to reliably test for co-location across Amazon EC2, Google GCE, and Microsoft Azure. We also find ways to detect co-location with victim web servers in a multi-tiered cloud application located behind a load balancer.

We use our new co-residence tests and multiple customer accounts to launch VM instances under different strategies that seek to maximize the likelihood of co-residency. We find that it is much easier (10× higher success rate) and cheaper (up to \$114 less) to achieve co-location in these three clouds when compared to a secure reference placement policy.

Keywords: co-location detection, multi-tenancy, cloud security

1 Introduction

Public cloud computing offers easy access to relatively cheap compute and storage resources. Cloud providers are able to sustain this cost-effective solution through *multi-tenancy*, where the infrastructure is shared between computations run by arbitrary customers over the Internet. This increases utilization compared to dedicated infrastructure, allowing lower prices.

However, this practice of multi-tenancy also enables various security attacks in the public cloud. Should an adversary be able to launch a virtual machine on the same physical host as a victim, making the two VMs co-resident (sometimes the term co-located is used), there exist attacks that break the logical isolation provided by virtualization to breach confidentiality [29, 32, 33, 35, 37, 38] or degrade the performance [30, 39] of the victim. Perhaps most notable are the side-channel attacks that steal private keys across the virtual-machine isolation boundary by cleverly monitoring shared resource usage [35, 37, 38].

Less understood is the ability of adversaries to arrange for co-residency in the first place. In general, doing so consists of using a *launch strategy* together with a mechanism for *co-residency detection*. The only prior work on obtaining co-residency [29] showed simple network-topology-based co-residency checks along with low-cost launch strategies that obtain a high probability of achieving co-residency compared to simply launching as many VM instances as possible. When such advantageous strategies exist, we say the cloud suffers from a placement vulnerability. Since then, Amazon has made several changes to their architecture, including removing the ability to do the simplest co-residency check. Whether placement vulnerabilities exist in other public clouds has, to the best of our knowledge, never been explored.

In this work, we provide a framework to systematically evaluate public clouds for placement vulnerabilities and show that three popular public cloud providers may be vulnerable to co-location attacks. More specifically, we set out to answer four questions:

- Can co-residency be effectively detected in modern public clouds?
- Are known launch strategies [29] still effective in modern clouds?

[*] Work primarily done while at the University of Wisconsin-Madison.

- Are there any new exploitable placement vulnerabilities?
- Can we quantify the money and time required of an adversary to achieve a certain probability of success?

We start by exploring the efficacy of prior co-residency tests (§ 4) and develop more reliable tests for our placement study (§ 4.1). We also find a novel test to detect co-residency with VMs uncontrolled by the attacker by just using their public interface even when they are behind a load balancer (§ 4.3).

We use multiple customer accounts across three popular cloud providers, launch VM instances under different scenarios that may affect the placement algorithm, and test for co-residency between all launched instances. We analyze three popular cloud providers, Amazon Elastic Compute Cloud (EC2) [2], Google Compute Engine (GCE) [6] and Microsoft Azure (Azure) [15], for vulnerabilities in their placement algorithm. After exhaustive experimentation with each of these cloud providers and at least 190 runs per cloud provider, we show that an attacker can still successfully arrange for co-location (§ 5). We find new launch strategies in these three clouds that obtain co-location faster (10x higher success rate) and cheaper (up to $114 less) when compared to a secure reference placement policy.

Next, we start by giving some background on public clouds (§ 2) and then define our threat model (§ 3). We conclude the paper with related and future work (§ 6 and § 7, respectively).

2 Background

Public clouds. Infrastructure-as-a-service (IaaS) public clouds, such as Amazon EC2, Google Compute Engine and Microsoft Azure, provide a management interface for customers to launch and terminate VM instances with a user-specified configuration. Typically, users register with the cloud provider for an account and use the cloud interface to specify VM configuration, which includes instance type, disk image, data center or region to host the VMs, and then launch VM instances. In addition, public clouds also provide many higher-level services that monitor load and automatically launch or terminate instances based on the workload [4,8,13]. These services internally use the same mechanisms as users to configure, launch and terminate VMs.

The provider's VM launch service receives from a client a desired set of parameters describing the configuration of the VM. The service then allocates resources for the new VM; this process is called *VM provisioning*. We are most interested in the portion of VM provisioning that selects the physical host to run a VM, which we call the *VM placement algorithms*. The resulting VM-to-host mapping we call the *VM placement*. The placement for a specific virtual machine may depend on many factors: the load on each machine, the number of machines in the data center, the number of concurrent VM launch requests, etc.

Type	Variable
Placement Parameters	# of customers
	# of instances launched per customer
	Instance type
	Data Center (DC) or Region
	Time launched
	Cloud provider
Environment Variable	Time of the day
	Days of the week
	Number of in-use VMs
	Number of machines in DC

Figure 1: **List of placement variables.**

While cloud providers do not generally publish their VM placement algorithms, there are several variables under the control of the user that could affect the VM placement, such as time-of-day, requested data center, and number of instances. A list of some notable parameters are given in Figure 1. By controlling these variables, an adversary can partially influence the placement of VMs on physical machines that may also host a target set of VMs. We call these variables *placement variables* and the set of values for these variables form a *launch strategy*. An example launch strategy is to launch 20 instances 10 minutes after triggering an auto-scale event on a victim application. This is, in fact, a launch strategy suggested by prior work [29].

Placement policies. VM placement algorithms used in public clouds often aim to increase data center efficiency, quality of service, or both by realizing some *placement policy*. For instance, a policy that aims to increase data center utilization may pack launched VMs on a single machine. Similarly policies that optimize the time to provision a VM, which involves fetching an image over the network to the physical machine and booting, may choose the last machine that used the same VM image, as it may already have the VM image cached on local disks. Policies may vary across cloud providers, and even within a provider.

Public cloud placement policies, although undocumented, often exhibit behavior that is externally observable. One example is *parallel placement locality* [29], in which VMs launched from different accounts within a short time window are often placed on the same physical machine. Two instances launched sequentially, where the first instance is terminated before the launch of the second one, are often placed on the same physical machine, a phenomenon called *sequential placement locality* [29].

These placement behaviors are artifacts of the two placement policies described earlier, respectively. Other examples of policies and resulting behaviors exist as well. VMs launched from the same accounts may either be packed on the same physical machine to maximize locality (and hence co-resident with themselves) or striped across different physical machines to maximize redundancy (and hence never co-resident with themselves). In the course of normal usage, such behaviors are unlikely to be noticed, but they can be measured with careful experiments.

Launch strategies. An adversary can exploit placement behaviors to increase the likelihood of co-locating with target victims. As pointed out by Ristenpart et al. [29], parallel placement locality can be exploited by triggering a scale-up event on target victim by increasing their load, which will cause more victim VMs to launch. The adversary can then simultaneously (or after a time lag) launch multiple VMs some of which may be co-located with the newly launched victim VM(s).

In this study, we develop a framework to systematically evaluate public clouds against launch strategies and uncover previously unknown placement behaviors. We approach this study by (i) identifying a set of placement variables that characterize a VM, (ii) enumerating the most interesting values for these variables, and (iii) quantifying the cost of such a strategy, if it in fact exposes a co-residency vulnerability. We repeat this for three major public cloud providers: Amazon EC2, Google Compute Engine, and Microsoft Azure. Note that the goal of this study is not to *reverse engineer* the exact details of the placement policies, but rather to identify launch strategies that can be exploited by an adversary.

Co-residency detection. A key technique for understanding placement vulnerabilities is detecting when VMs are co-resident on the same physical machine (also termed *co-locate*). In 2009, Ristenpart et al. [29] proposed several co-residency detection techniques and used them to identify several placement vulnerabilities in Amazon EC2. As co-resident status is not reported directly by the cloud provider, these detection methods are usually referred to as side-channel based techniques, which can be further classified into two categories: *logical side-channels* or *performance side-channels*.

Logical side-channels: Logical side-channels allow information leakage via logical resources that are observable to a software program, e.g., IP addresses, timestamp counter values. Particularly in Amazon EC2, each VM is assigned two IP addresses, a *public* IP address for communication over the Internet, and a private or *internal* IP address for intra-datacenter communications. The EC2 cloud infrastructure allowed translation of public IP addresses to their internal counterparts. This translation revealed the topology of the internal data center network, which allowed a remote adversary to map the entire public cloud infrastructure and determine, for example, the availability zone and instance type of a victim. Furthermore, co-resident VMs tended to have adjacent internal IP addresses.

Logical side-channels can also be established via shared timestamp counters. In prior work, skew in timestamp counters were used to fingerprint a physical machine [27], although this technique has not yet been explored for co-residency detection. Co-residency detection via shared state like interrupt counts and process statistics reported in `procfs` also come under this category, but are only applicable to container-based platform-as-a-service clouds.

Performance side-channels: Performance side-channels are created when performance variations due to resource contention are observable. Such variations can be used as an indicator of co-residency. For instance, network performance has been used for detecting co-residence [29, 30]. This is because hypervisors often directly relay network traffic between VMs on the same host, providing detectably shorter round-trip times than between VMs on different hosts.

Covert channels, as a special case of side-channels, can be established between two VMs that are cooperating in order to detect co-residency. For purposes of co-residency detection, covert channels based on shared hardware resources, such as last level caches (LLCs) or local storage disks, can be exploited by one VM to detect performance degradation caused by a co-resident VM. Covert channel detection techniques require control over both VMs, and are usually used in experimentation rather than in practical attacks. We later refer to such approaches as *cooperative co-residency detection*.

Placement study in PaaS. While we mainly studied placement vulnerabilities in the context of IaaS, we also experimented with Platform-as-a-Service (PaaS) clouds. PaaS clouds offer elastic application hosting services. Unlike IaaS where users are granted full control of a VM, PaaS provides managed compute tasks (or instances) for the execution of hosted web applications, and allow multiple such instances to share the same operating system. These clouds use either process-level isolation via file system access controls, or increasingly Linux-style containers (see [38] for a more detailed description). As such, logical side-channels alone are usually sufficient for co-residency detection purposes. For instance, in PaaS clouds, co-resident instances often share the same public IP address as the host machine. This is because the host-to-instance network is often configured using Network Address Translation (NAT) and each instance is assigned a unique port under the host IP address for incoming connections.

We found that many such logical side-channel-based co-residency detection approaches worked on PaaS clouds, even on those using containers. Specifically, we used both system-level interrupt statistics via `/proc/interrupts` and shared public IP addresses of the instances to detect co-location in Heroku [10].

Our brief investigation of co-location attacks in Heroku showed that naïve strategies like scaling two PaaS web applications to 30 instances with a time interval of 5 minutes between them, resulted in co-location in 6 out of 10 attempts. Moreover, since the co-location detection was simple and quick including the time taken for application scaling, we were able to do these experiments free of cost.

This result reinforces prior findings on PaaS co-location attacks [38] and confirms the existence of cheap launch strategies to achieve co-location and easy detection mechanisms to verify it. We do not investigate PaaS clouds further in the rest of this paper.

3 Threat Model

Co-residency attacks in public clouds, as mentioned earlier, involve two steps: a launch strategy and co-residency detection. We assume that the adversary has access to tools to identify a set of target victims, and either knows victim VMs' launch characteristics or can directly trigger their launches. The latter is possible by increasing load in order to cause the victim to scale up by launching more instances. The focus of this study is to identify if there exists any launch strategy that an adversary can devise to increase the chance of co-residency with a set of targeted victim VMs.

In our threat model, we assume that the cloud provider is trusted and the attacker has no affiliation of any form with the cloud provider. This also means that the adversary has no internal knowledge of the placement policies that are responsible for the VM placements in the public cloud. An adversary also has the same interface for launching and terminated VMs as regular customers, and no other special interfaces. Even though there may be per-account limits on the number of VMs that a cloud provider imposes, an adversary has access to an unlimited number of accounts and hence has no limit on the number of VMs he could launch at any given time.

No resource-limited cloud provider is a match to an adversary with limitless resources and hence realistically we assume that the adversary is resource-limited. For the same reason, a cloud provider is vulnerable to a launch strategy only when it is trivial or cost-effective for an adversary. As such, we aim to (i) quantify the cost of executing a launch strategy by an adversary, (ii) define a reference placement policy with which the placement policies of real clouds can be compared, and (iii) define metrics to quantify placement vulnerability.

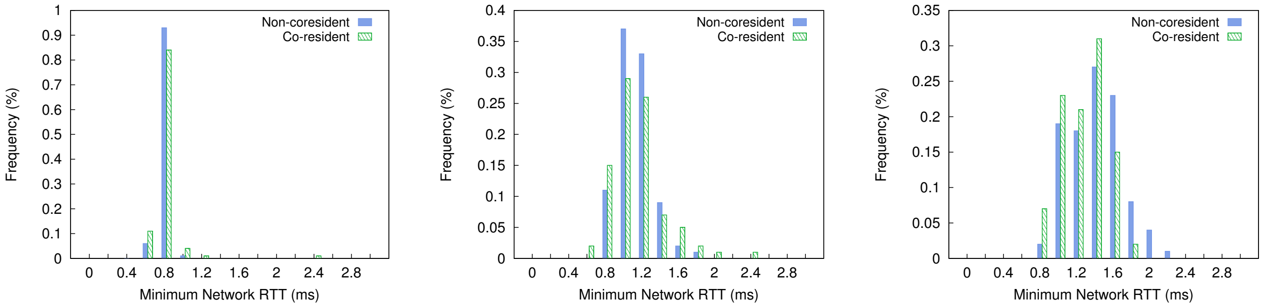
Cost of a launch strategy. Quantifying the cost of a launch strategy is straightforward: it is the cost of launching a number of VMs and running tests to detect co-residency with one or more target victim VMs. To be precise, the cost of a launch strategy S is given by $C_S = a * P(a_{type}) * T_d(v,a)$, where a is the number of attacker VMs of type a_{type} launched to get co-located with one of the v victim VMs. $P(a_{type})$ is the price of running one VM of type a_{type} for a unit time. $T_d(a,v)$ is the time to detect (in billing units) co-residency between all pairs of a attackers and v victim VMs, excluding pairs within each group. For simplicity, we assume that the attacker is running all instances till the last co-residency check is completed.

Reference placement policy. In order to define placement vulnerability, we need a yardstick to compare various placement policies and the launch strategies that they may be vulnerable to. To aid this purpose, we define a simple reference placement policy that has good security properties against co-residency attacks and use it to gauge the placement policies used in public clouds. Let there be N machines in a data center and let each machine have unlimited capacity. Given a set of unordered VM launch requests, the mapping of each VM to a machine follows a uniform random distribution. Let there be v victim VMs assigned to v unique machines among N, where $v \ll N$. The probability of *at least one* collision (i.e. co-residency) under the random placement policy and the above attack scenario when attacker launches a instances is given by $1 - \left(1 - v/N\right)^a$. We call this probability the reference probability[1]. Recall that for calculating cost of a launch strategy under this reference policy, we also need to define the price function, $P(vm_{type})$. For simplicity, we use the minimum price offered by any cloud provider as the price for the compute resource under the reference policy. For example, at the time of this study, Amazon EC2 offered t2.small instances at $0.026 per hour of instance activity, which was the cheapest price across all three clouds considered in this study.

Note that the reference placement policy makes several simplifying assumptions, but these only benefit the attacker. This is conservative as we will compare our experimental results to the best possible launch strategy under the reference policy. For instance, the assumption on unlimited capacity of the servers only benefits the attacker as it never limits the number of victim VMs an attacker could potentially co-locate with. We use a conservative value of 1000 for N, which is at least an order-of-magnitude less than the number of servers (50,000) in the smallest reported Amazon EC2 data centers [5]. Similarly, the price function of this placement policy also favors an attacker as it provides the cheapest price possible in the market even though in reality a secure placement policy may demand a higher price. Hence, it would be troubling if the state-of-the-art placement policies used in public clouds does not measure well even against such a conservative reference placement policy.

Placement Vulnerability. Putting it all together, we define two metrics to gauge any launch strategy against a placement policy: (i) *normalized success rate*, and (ii) *cost-benefit*. The normalized success rate is the success rate of the launch strategy in the cloud under test normalized to the success rate of the same strategy under the reference placement policy. The cost-benefit of a strategy is the additional cost that is incurred by the adversary in the reference placement policy to achieve the same success rate as the strategy in the placement policy under test. We define that a placement policy has a *placement vulnerability* if and only if there exists a launch strategy with a normalized success rate that is greater than 1.

[1] This probability event follows a hypergeometric distribution.



(a) GCE (b) EC2 (c) Azure

Figure 2: **Histogram of minimum network round trip times between pairs of VMs.** The frequency is represented as a fraction of total number of pairs in each category. The figure does not show the tail of the histogram.

Note that the normalized success rate quantifies how easy it is to get co-location. On the other hand, the cost benefit metric helps to quantify how cheap it is to get co-location compared to a more secure placement policy. These metrics can be used to compare launch strategies under different placement policies, where a higher value for any of these metrics indicate that the placement policy is relatively more vulnerable to that launch strategy. An ideal placement policy should aim to reduce both the success rate and the cost benefit of any strategy.

4 Detecting Co-Residence

An essential prerequisite for the placement vulnerability study is access to a co-residency detection technique that identifies whether two VMs are resident on the same physical machine in a third-party public cloud.

Challenges in modern clouds. Applying the detection techniques mentioned in Section 2 is no longer feasible in modern clouds. In part due to the vulnerability disclosure by Ristenpart et al. [29], modern public clouds have adopted new technologies that enhance the isolation between cloud tenants and thwart known co-residence detection techniques. In the network layer, virtual private clouds (VPCs) have been broadly employed for data center management [17, 20]. With VPCs, internal IP addresses are private to a cloud tenant, and can no longer be used for cloud cartography. Although EC2 allowed this in older generation instances (called EC2-classic), this is no longer possible under Amazon VPC setting. In addition, VPCs require communication between tenants to use public IP addresses for communication. As shown in Figure 2, the network timing test is also defeated, as using public IP addresses seems to involve routing in the data center network rather than short-circuiting through the hypervisor. Here, the ground-truth of co-residency is detected using memory-based covert-channel (described later in this section). Notice that there is *no* clear distinction between the frequency distribution of the network round trip times of co-resident and non-coresident pairs on all three clouds.

In the system layer, persistent storage using local disks is no longer the default. For instance, many Amazon EC2 instance types do not support local storage [1]; GCE and Azure provide only local Solid State Drives (SSD) [7, 14], which are less susceptible to detectable delays from long seeks. In addition, covert channels based on last-level caches [29, 30, 33, 36] are less reliable in modern clouds that use multiple CPU packages. Two VMs sharing the same machine may not share LLCs to establish the covert channel. Hence, these LLC-based covert-channels can only capture a subset of co-resident instances.

As a result of these technology changes, none of the prior techniques for detecting co-residency reliably work in modern clouds, compelling us to develop new approaches for our study.

4.1 Co-residency Tests

We describe in this subsection a pair of tools for co-residency tests, with the following design goals:

- Applicable to a variety of *heterogeneous* software and hardware stacks used in public clouds.
- Detect co-residency with *high confidence*: the false detection rate should be low even in the presence of background noise from other neighboring VMs.
- Detect co-residency *fast* enough to facilitate experimentation among large sets of VMs.

We chose a performance covert-channel based detection technique that exploits shared hardware resources, as this type of covert-channels are often hard to remove and most clouds are very likely to be vulnerable to it.

A covert-channel consists of a *sender* and a *receiver*. The sender creates contention for a shared resource and uses it to signal another tenant that potentially share the same resource. The receiver, on the other hand, senses this contention by periodically measuring the performance of that shared resource. A significant performance degradation measured at the receiver results in a successful detection of a sender's signal. Here the reliability of the covert-channel is highly dependent on the choice of the shared resource and the level of contention created by the sender. The sender is the key component of the co-residency detection techniques we developed as part of this study.

```
// allocate memory multiples of 64 bits
char_ptr = allocate_memory((N+1)*8)
//move half word up
unaligned_addr = char_ptr + 2
loop forever:
   loop i from (1..N):
     atomic_op(unaligned_addr + i, some_value)
   end loop
```

Figure 3: **Memory-locking – Sender.**

Memory-locking sender. Modern x86 processors support atomic memory operations, such as `XADD` for atomic addition, and maintain their atomicity using cache coherence protocols. However, when a locked operation extends across a cache-line boundary, the processor may lock the memory bus temporarily [32]. This locking of the bus can be detected as it slows down other uses of the bus, such as fetching data from DRAM. Hence, when used properly, it provides a timing covert channel to send a signal to another VM. Unlike cache-based covert channels, this technique works regardless of whether VMs share a CPU core or package.

We developed a sender exploiting this shared memory-bus covert-channel. The psuedocode for the sender is shown in Figure 3. The sender creates a memory buffer and uses pointer arithmetic to force atomic operations on unaligned memory addresses. This indirectly locks the memory bus even on all modern processor architectures [32].

```
size = LLC_size * (LLC_ways +1)
stride = LLC_sets * cacheline_sz)
buffer = alloc_ptr_chasing_buff(size, stride)
loop sample from (1..10): //number of samples
     start_rdtsc = rdtsc()
     loop probes from (1..10000):
         probe(buffer); //always hits memory
     end loop
     time_taken[sample] = (rdtsc() - start_rdtsc)
end loop
```

Figure 4: **Memory-probing – Receiver.**

Receivers. With the aforementioned memory-locking sender, there are several ways to sense the memory-locking contention induced by the sender in another co-resident tenant instance. All the receivers measure the memory bandwidth of the shared system. We present two types of receivers that we used in this study that works on heterogeneous hardware configurations.

Memory-probing receiver uses carefully crafted memory requests that always miss in the cache hierarchy and always hit memory. Constricting the data accesses of the receiver into a *single LLC set* ensures this. In order to evade hardware prefetching, we use a pointer-chasing buffer to randomly access a list of memory addresses (pseudocode shown in Figure 4). The time needed to complete a fixed number of probes (e.g., 10,000) provides a signal of co-residence: when the sender is performing locked operations, loads from memory proceed slowly.

Memory-locking receiver is similar to the sender but measures the number of unaligned atomic operations that could be completed per unit time. Although it also measures the memory bandwidth, unlike the memory-probing receiver, it works even when the cache architecture of the machine is unknown.

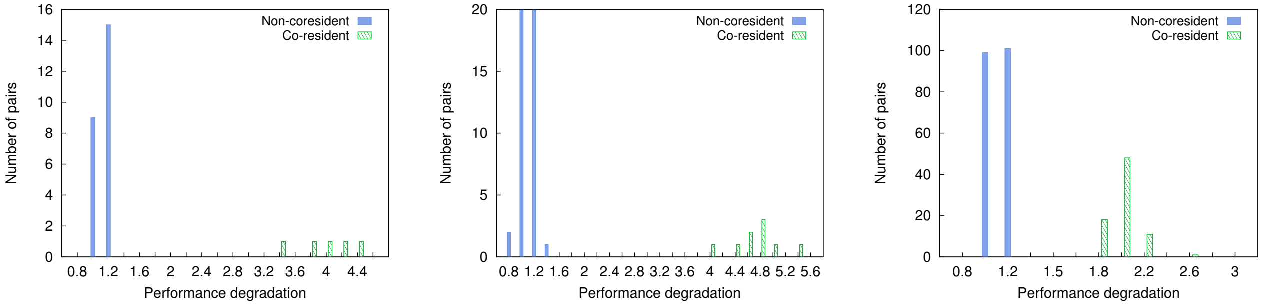
The sender along with these two receivers form our two novel co-residency detection methods that we use in this study: *memory-probing test* and *memory-locking test* (named after their respective receivers). These comprise our co-residency test suite. Each test in the suite starts by running the receiver on one VM while keeping the other idle. The performance measured by this run is the baseline performance without contention. Then the receiver and the sender are run together. If the receiver detects decreased performance, the tests conclude that the two VMs are co-resident. We use a slowdown threshold to detect when the change in receiver performance indicates co-residence (discussed later in the section).

Machine	Cores	Memory	Memory	Socket
Architecture		Probing	Locking	
Xeon E5645	6	3.51	1.79	Same
Xeon X5650	12	3.61	1.77	Same
Xeon X5650	12	3.46	1.55	Diff.

Figure 5: **Memory-probing and -locking on testbed machines.** Slowdown relative to the baseline performance observed by the receiver averaged across 10 samples. Same – sender and receiver on different cores on the same socket, Diff. – sender and receiver on different cores on different sockets. Xeon E5645 machine had a single socket.

Evaluation on local testbed. In order to measure the efficacy of this covert-channel we ran tests in our local testbed. Results of running memory-probing and -locking tests under various configurations are shown in Figure 5. The hardware architectures of these machines are similar to what is observed in the cloud [21]. Across these hardware configurations, we observed a performance degradation of at least 3.4× compared to not running memory-locking sender on a non-coresident instance (i.e. a baseline run with idle sender), indicating reliability. Note that this works even when the co-resident instances are running on cores on different sockets, which does not share the same LLC (works on *heterogeneous* hardware). Further, a single run takes one tenth of a second to complete and hence is also *quick*.

Note that for this test suite to work in the real world, an attacker requires control over both the VMs under test, which includes the victim. We call this scenario as co-residency detection under cooperative victims (in short, *co-operative co-residency detection*). Such a mechanism is sufficient to observe placement behavior in public clouds (Section 4.2). We further investigated approaches to detect co-residency under a realistic setting with an uncooperative victim. In Section 4.3 we show how to adapt the memory-probing test to detect co-location with one of the many web-servers behind a load balancer.



(a) GCE (b) EC2 (c) Azure – Intel machines

Figure 6: **Distribution of performance degradation of memory-probing test.** For varying number of pairs on each cloud (GCE:29, EC2:300, Azure:278). Note the x-axis plots performance degradation. Also for EC2 x-axis range is cut short at 20 pairs for clarity.

4.2 Cooperative Co-residency Detection

In this section, we describe the methodology we used to detect co-residency in public clouds. For the purposes of studying placement policies, we had the flexibility to control both VMs that we test for co-residence. We did this by launching VMs from two separate accounts and test them for pairwise co-residence. We encountered several challenges when running the co-residency test suite on three different public clouds - Google Computer Engine, Amazon EC2 and Microsoft Azure.

First, we had to handle noise from neighboring VMs sharing the same host. Second, hardware and software heterogeneity in the three different public clouds required special tuning process for the co-residency detection tests. Finally, testing co-residency for a large set of VMs demanded a scalable implementation. We elaborate on our solution to these challenges below.

Handling noise. Any noise from neighboring VMs could affect the performance of the receiver with and without the signal (or baseline) and result in misdetection. To handle such noise, we alternate between measuring the performance with and without the sender's signal, such that any noise equally affects both the measurements. Secondly, we take ten samples of each measurement and only detect co-residence if the ratios of *both* the mean and median of these samples exceed the threshold. As each run takes a fraction of a second to complete, repeating 10 times is still fast enough.

Cloud	**Machine**	**Clock**	**LLC**
Provider	**Architecture**	(GHz)	(Ways × Set)
EC2	Intel Xeon E5-2670	2.50	20 × 20480
GCE	Generic Xeon*	2.60*	20 × 16384
Azure	Intel E5-2660	2.20	20 × 16384
Azure	AMD Opteron 4171 HE	2.10	48 × 1706

Figure 7: **Machine configuration in public clouds.** The machine configurations observed over all runs with small instance types. GCE did not reveal the exact microarchitecture of the physical host (*). Ways × Sets × Word Size gives the LLC size. The word size is 64 bytes on all these x86-64 machines.

Tuning thresholds. As expected, we encountered different machine configurations on the three different public clouds (shown in Figure 7) with heterogeneous cache dimensions, organizations and replacement policies [11, 26]. This affects the performance degradation observed by the receivers with respect to the baseline and the ideal threshold for detecting co-residency. This is important because the thresholds we use to detect co-residence yield false positives, if set too low, and false negatives if set too high. Hence, we tuned the threshold to each hardware we found on all three clouds.

We started with a conservative threshold of 1.5x and tuned to a final threshold of 2x for GCE and EC2 and 1.5x for Azure for both the memory-probing and -locking tests. Figure 6 shows the distribution of performance degradation under the memory-probing tests across Intel machines in EC2, GCE, and Azure. For GCE and EC2, a performance degradation threshold of 2 clearly separates co-resident from non-coresident instances. For all Intel machines we encountered, although we ran both memory-locking and -probing tests, memory-probing was sufficient to detect co-residency. For Azure, overall we observe lower performance degradation and the initial threshold of 1.5 was sufficient to detect co-location on Intel machines.

The picture for AMD machines in Azure differs significantly as shown in Figure 8. The distribution of perfor-

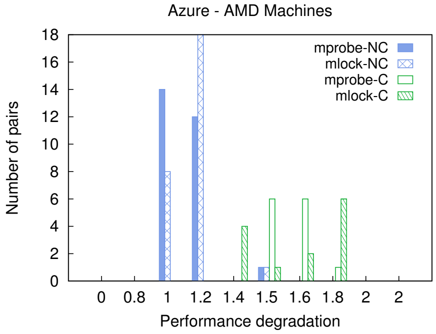


Figure 8: **Distribution of performance degradation of memory-probing and -locking tests.** On AMD machines in Azure with 40 pairs of nodes. Here NC stands for non-coresident and C, co-resident pairs. Note that the x-axis plots performance degradation.

mance degradation for both memory-locking and memory-probing shows that, unlike for other architectures, co-residency detection is highly sensitive to the choice of the threshold for AMD machines. This may be due to the more associative cache (48 ways vs. 20 for Intel), or different handling of locked instructions. For these machines, a threshold of 1.5 was high enough to have no false positives, which we verified by hand checking the instances using the two covert-channels and observed consistent performance degradation of at least 50%. We determine a pair of VMs as co-resident if the degradation in *either* of the tests is above this threshold. We did not detect any cross-architecture (false) co-residency detection in any of the runs.

Scaling co-residency detection tests. Testing co-residency at scale is time-consuming and increases quadratically with the number of instances: checking 40 VM instances, involves 780 pair-wise tests. Even if each run of the entire co-residency test suite takes only 10 seconds, a naïve sequential execution of the tests on all the pairs will take 2 hours. Parallel co-residency checks can speed checking, but concurrent tests may interfere with each other.

To parallelize the test, we partition the set of all VM pairs ($\binom{v+a}{2}$) into sets of pairs with no VMs twice; we run one of these sets at a time and record which pairs detected possible co-residence. After running all sets, we have a set of candidate co-resident pairs, which we test sequentially. Parallelizing co-residency tests significantly decreased the time taken to test all co-residency pairs. For instance, the parallel version of the test on one of the cloud providers took 2.4 seconds per pair whereas the serial version took almost 46.3 seconds per pair (a speedup of 20x). While there are faster ways to parallelize co-residency detection, we chose this approach for simplicity.

Veracity of our tests. Notice that a performance degradation of 1.5x, 2x and 4x corresponds to 50%, 100% and 300% performance degradation. Such high performance degradation (even 50%) is clear enough signal to declare co-residency due to resource sharing. Furthermore, we hand checked by running the two tests in isolation on the detected instance-pairs for a significant fraction of the runs for all clouds and observed a consistent covert-channel signal. Thus our methodology did not detect any false positives, which are more detrimental to our study than false negatives. Although *co-residency* here implies sharing of memory channel, which may not always mean sharing of cores or other per-core hardware resources.

4.3 Co-residency Detection on Uncooperative Victims

Until now, we described a method to detect co-residency with a cooperative victim. In this section, we look at a more realistic setting where an adversary wishes to detect co-residency with a victim VM with accesses limited to only public interfaces like HTTP or a key-value (KV) store's put-get interface. We show that the basic cooperative co-residency detection can also be employed to detect co-residency with an uncooperative victim in the wild.

Attack setting. Unlike previous attack scenarios, we assume the attacker has no access to the victim VMs or its application other than what is permitted to any user on the Internet. That is, the victim application exposes a well-known public interface (e.g., HTTP, FTP, KV-store protocol) that allows incoming requests, which is also the only access point for the attacker to the victim. The front end of this victim application can range from caching or data storage services (e.g., memcached, cassandra) to generic webservers. We also assume that there may be multiple instances of this front-end service running behind a load balancer. Under this scenario, the attacker wishes to detect co-location with one or more of the front-facing victim VMs.

Co-residency test. We adapt the memory tests used in previous section by running the memory-locking sender in the attacker instance. For a receiver, we use the public interface exposed by the victim by generating a set of requests that potentially makes the victim VMs hit the memory bus. This can be achieved by looping through a large number of requests of sizes approximately equal or greater than the size of the LLC. This creates a performance side-channel that leaks co-residency information. This receiver runs in an independent VM under the adversary's control, which we call the co-residency detector.

Experiment setup. To evaluate the efficacy of this method, we used the Olio multi-tier web application [12] that is designed to mimic a social-networking application. We used an instance of this workload from CloudSuite [22]. Although Olio supports several tiers (e.g., memcached to cache results of database queries), we configured it with two tiers, with each webserver and the database server running in a separate VM of type t2.small on Amazon EC2. Multiple of these webserver VMs are configured behind a HAProxy-based load balancer [9] running in an m3.medium instance (for better networking performance). The load balancer follows the standard configuration of using round-robin load balancing algorithm with sticky client sessions using cookies. We believe such a victim web application and its configuration is a reasonable generalization of real world applications running in the cloud.

For the attacker, we use an off-the-shelf HTTP performance measurement utility called `HTTPerf` [28] as the receiver in the co-residency detection test. This receiver is run inside a t2.micro instance (for free of charge). We used a set of 212 requests that included web pages and web objects (images, PDF files). We gathered these requests using the access log of manual navigation around the web application from a web browser.

Evaluation methodology. We start with a known co-resident VM pair using the cooperative co-residency detection method. We configure one of the VMs as a victim webserver VM and launch four more VMs: two webservers, one database server and a load balancer, all of which are not co-resident with the attacker VM.

Co-residency detection starts by measuring the average request latency at the receiver inside the co-residency detector for the baseline (with idle attacker) and contended case with the attacker running the memory-locking sender. A significant performance degradation between the baseline and the contended case across multiple samples reveal co-residency of one of the victim VMs with the attacker VM. On Amazon EC2, with the above setup we observed an average request latency of 4.66ms in the baseline case and a 10.6ms in the memory-locked case, i.e., a performance degradation of $\approx 2.3\times$.

Background noise. The above test was performed when the victim web application was idle. In reality, any victim in the cloud might experience constant or varying background load on the system. False positives or negatives may occur when there is spike in load on the victim servers. In such case, we use the same solution as in Section 4.2 — alternating between measuring the idle and the contended case.

In order to gauge the efficacy of the test under constant background load, we repeated the above experiment with varying load on the victim. The result of this experiment is summarized in Figure 9. Counterintuitively, we found that a constant load on the background server exacerbates the performance degradation gap, hence resulting in a clearer signal of co-residency. This is because running memory-locking on the co-resident attacker increases the service time of all requests as majority of the requests rely on memory bandwidth. This increases the queuing delay in the system and in turn increasing the overall request latency. Interestingly, this aforementioned performance gap stops widening at higher loads of 750 to 1000 concurrent users as the system hits a bottleneck (in our case a network bottleneck at the load balancer) even without running the memory-locking sender. Thus, detecting co-residency with a victim VM that is part of a highly loaded and bottlenecked application would be hard using this test.

We also experimented with increasing the number of victim webservers behind the load balancer beyond 3 (Figure 10). As expected, the co-residency signal grew weaker with increasing victims, and at 9 webservers, the performance degradation was too low to be useful for detecting co-residency.

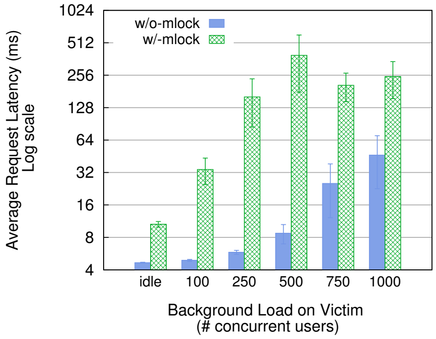


Figure 9: **Co-residency detection on an uncooperative victim.** The graph shows the average request latency at the co-residency detector without and with memory-locking sender running on the co-resident attacker VM under varying background load on the victim. Note that the y-axis is in log scale. The load is in the number of concurrent users, where each user on average generates 20 HTTP requests per second to the webserver.

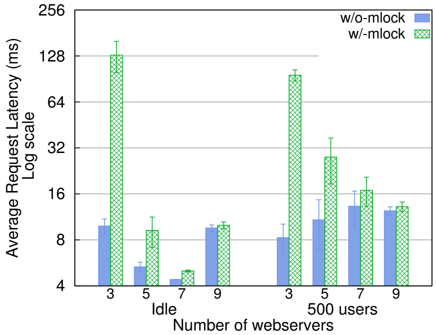


Figure 10: **Varying number of webservers behind the load balancer.** The graph shows the average request latency at the co-residency detector without and with memory-locking sender running on the co-residency attacker VM under varying background load on the victim. Note that the y-axis is in log scale. The error bars show the standard deviation over 5 samples.

5 Placement Vulnerability Study

In this section, we evaluate three public clouds, Amazon EC2, Google Compute Engine and Microsoft Azure, for placement vulnerabilities and answer the following questions: (i) what are all the strategies that an adversary can employ to increase the chance of co-location with one or more victim VMs? (ii) what are the chances of success and cost of each strategy? and (iii) how do these strategies compare against the reference placement policy introduced in Section 3?

5.1 Experimental Methodology

Before presenting the results, we first describe the experiment setting and methodology that we employed for this placement vulnerability study.

Experiment settings. Recall VM placement depends on several placement variables (shown in Figure 1). We assigned reasonable values to these placement variables and enumerated through several launch strategies. A *run* corresponds to one launch strategy and involves launching multiple VMs from two distinct accounts (i.e., subscriptions in Azure and projects in GCE) and checking for co-residency between all pairs of VMs launched. One account was designated as a proxy for the victim and the other for the adversary. We denote a *run configuration* by $v \times a$, where v is the number of victim instances and a is the number of attacker instances launched in that run. We varied v and a for all v, $a \in \{10, 20, 30\}$ and restricted them to the inequality, $v \leq a$, as it increases the likelihood of achieving co-residency.

Other placement variables that are part of the run configuration include: victim launch time (including time of the day, day of the week), delay between victim and attacker VM launches, victim and attacker instance types and data center location or region where the VMs are launched. We repeat each run multiple times across all three cloud providers. The repetition of experiments is especially required to control the effect of certain environment variables like time of day. We repeat experiments for each run configuration over various times of the day and days of the week. We fix the instance type of VMs to small instances (t2.small on EC2, g1.small on GCE and small or Standard-A1 on Azure) and data center regions to us-east for EC2, us-central1-a for GCE and east-us for Azure, unless otherwise noted. All experiments were conducted over 3 months between December 2014 to February 2015.

We used a single, local Intel Core i7-2600 machine with 8 SMT cores to launch VM instances, log instance information and run the co-residency detection test suite.

Implementation and the Cloud APIs. In order to automate our experiments, we used Python and the libcloud[2] library [3] to interface with EC2 and GCE. Unfortunately, libcloud did not support Azure. The only Azure cloud API on Linux platform was a node.js library and a cross-platform command-line interface (CLI). We built a wrapper around the CLI. There were no significant differences across different cloud APIs except that Azure did not have any explicit interface to launch multiple VMs simultaneously.

As mentioned in the experiment settings, we experimented with various delays between the victim and attacker VM launches (0, 1, 2, 4 ... hours). To save money, we reused the same set of victim instances for each of the longer runs. That is, for the run configuration of 10x10 with 0, 1, 2, and 4 hours of delay between victim and attacker VM launches, we launched the victim VMs only once at the start of the experiment. After running co-residency tests on the first set of VM pairs, we terminated all the attacker instances and relaunched attacker VM instances after appropriate delays (say 1 hour) and rerun the tests with *the same set* of victim VMs. We repeat this until we experiment with all delays for this configuration. We call this methodology the *leap-frog method*. It is also important to note that zero delay here means parallel launch of VMs from our test machine (and not sequential launch of VMs from one account after another), unless otherwise noted.

In the sections below, we take a closer look at the effect of varying one placement variable while keeping other variables fixed across all the cloud providers. In each case, we use three metrics to measure the degree of co-residency: chances of getting at least one co-resident instance across a number of runs (or success rate), average number of co-resident instances over multiple runs and average coverage (i.e., fraction of victim VMs with which attacker VMs were co-resident). Although these experiments were done with victim VMs under our control, the results can be extrapolated to guide an attacker's launch strategy for an uncooperative victim. We also discuss a set of such strategic questions that the results help answer. At the end of this section, we summarize and calculate the cost of several interesting launch strategies and evaluate the public clouds against our reference placement policy.

Delay (hr.)	**Config.**	**Mean**	**S.D.**	**Min**	**Median**	**Max**
0	10x10	0.11	0.33	0	0	1
0	10x20	0.2	0.42	0	0	1
0	10x30	0.5	0.71	0	0	2
0	20x20	0.43	0.65	0	0	2
0	20x30	1.67	1.22	0	2	4
0	30x30	1.6	1.65	0	1	5
1	10x10	0.25	0.46	0	0	1
1	10x20	0.33	0.5	0	0	1
1	10x30	1.6	1.07	0	2	3
1	20x20	1.27	1.22	0	1	4
1	20x30	2.44	1.51	0	3	4
1	30x30	3	1.12	1	3	5

Figure 11: **Distribution of number of co-resident pairs on GCE.** Region: us-central1-a.

Delay (hr.)	**Config.**	**Mean**	**S.D.**	**Min**	**Median**	**Max**
0	*	0	0	0	0	0
1	10x10	0.44	0.73	0	0	2
1	10x20	1.11	1.17	0	1	3
1	10x30	1.4	1.43	0	1.5	4
1	20x20	3.57	2.59	0	3.5	9
1	20x30	3.78	1.79	1	4	7
1	30x30	3.89	2.09	2	3	9

Figure 12: **Distribution of number of co-resident pairs on EC2.** Region: us-east.

5.2 Effect of Number of Instances

In this section, we observe the placement behavior while varying the number of victim and attacker instances. Intuitively, we expect the chances of co-residency to increase with the number of attacker and victim instances.

Varying number of attacker instances. Keeping all the placement variables constant including the number of victim instances, we measure the chance of co-residency over multiple runs. The result of this experiment helps to answer the question: How many VMs should an adversary launch to increase the chance of co-residency?

As is shown in Figure 14, the placement behavior

[2] We used libcloud version 0.15.1 for EC2, and a modified version of 0.16.0 for GCE to support the use of multiple accounts in GCE.

Delay (hr.)	**Config.**	**Mean**	**S.D.**	**Min**	**Median**	**Max**
0	10x10	15.22	19.51	0	14	64
0	10x20	3.78	4.71	0	3	14
0	10x30	4.25	6.41	0	2.5	19
0	20x20	9.67	8.43	0	8	27
0	20x30	2.38	1.51	1	2	5
0	30x30	24.57	36.54	1	6	99
1	10x10	2.78	3.87	0	1	12
1	10x20	0.78	1.2	0	0	3
1	10x30	0.75	1.39	0	0	3
1	20x20	0.67	1.66	0	0	5
1	20x30	0.86	0.9	0	1	2
1	30x30	4.71	9.89	0	1	27

Figure 13: **Distribution of number of co-resident pairs on Azure.** Region: East US 1.

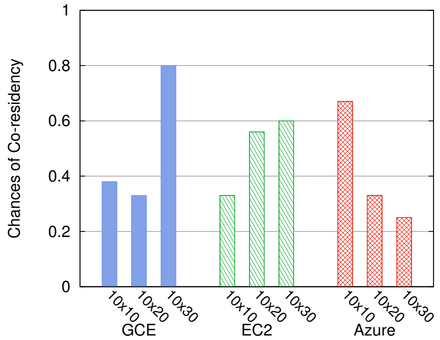


Figure 14: **Chances of co-residency of 10 victim instances with varying number of attacker instances.** All results are from one data center region (EC2: us-east, GCE: us-central1-a, Azure: East US) and the delays between victim and attacker instance launch were 1 hour. Results are over at least 9 runs per run configuration with at least 3 runs per time of day.

changes across different cloud providers. For GCE and EC2, we observe that higher the number of attacker instances relative to the victim instances, the higher the chance of co-residency is. Figure 11 and 12 show the distribution of number of co-resident VM pairs on GCE and EC2, respectively. The number of co-resident VM pairs also increases with the number of attacker instances, implying that the coverage of an attack could be increased with larger fraction of attacker instances than the target VM instances if the launch times are coordinated.

Contrary to our expectations, the placement behavior observed on Azure is the inverse. The chances of co-residency with 10 attacker instances are almost twice as high as with 30 attacker instances. This is also reflected in the distribution of number of co-residency VM pairs (shown in Figure 13). Further investigation revealed a correlation between the number of victim and attacker instances launched and the chance of co-residency. That is, for the run configuration of 10x10, 20x20 and 30x30, where number of victim and attacker instances are the same, and with 0 delay, the chance of co-residency were equally high for all these configurations (between 0.9 to 1). This suggests a possible placement policy that collates VM launch requests together based on their request size and places them on the same group of machines.

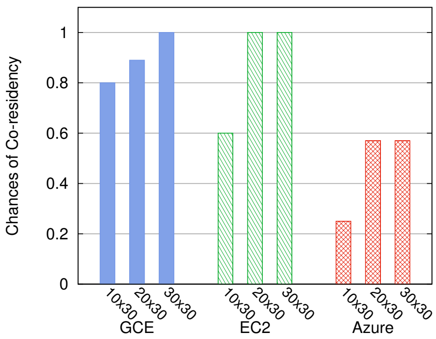


Figure 15: **Chances of co-residency of 30 attacker instances with varying number of victim instances.** All results are from one data center region (EC2: us-east, GCE: us-central1-a, Azure: East US) and the delays between victim and attacker instance launch were 1 hour. Results are over at least 9 runs per configuration with at least 3 runs per time of day.

Varying number of victim instances. Similarly, we also varied the number of victim instances by keeping the number of attacker instances and other placement variables constant (results shown in Figure 15). We expect the chance of co-residency to increase with the number of victims targeted. Hence, the results presented here help an adversary answer the question: What are the chances of co-residency with varying sizes of target victims?

As expected, we see an increase in the chances of co-residency with increasing number of victim VMs across all cloud providers. We see that the absolute value of the chance of co-residency is lower for Azure than other clouds. This may be the result of significant additional delay between victim and attacker launch times in Azure as a result of our methodology (more on this later).

5.3 Effect of Instance Launch Time

In this section, we answer two questions that aid an adversary to design better launch strategies: How quickly should an attacker launch VMs after the victim VMs are launched? Is there any increase in chance associated with the time of day of the launch?

Varying delay between attacker and victim launches. The result of varying the delay between 0 (i.e., parallel launch) and 1 hour delay is shown in Figure 16. We can make two immediate observations from this result.

The first observation reveals a significant artifact of EC2's placement policy: VMs launched within a short time window are never co-resident on the same machine. This observation helps an adversary to avoid such a strategy. We further investigated placement behaviors on EC2 with

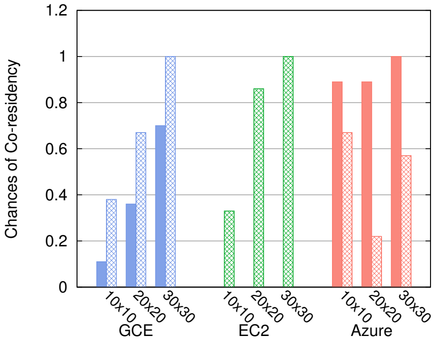


Figure 16: **Chances of co-residency with varying delays between victim and attacker launches.** *Solid* boxes correspond to zero delay (simultaneous launches) and *gauze*-like boxes correspond to 1 hour delay between victim and attacker launches. We did not observe any co-resident instances for runs with zero delay on EC2. All results are from one data center region (EC2: us-east, GCE: us-central1-a, Azure: East US). Results are over at least 9 runs per run configuration with at least 3 runs per time of day.

Delay	Mean	S.D.	Min	Median	Max	Success rate
0+	0.6	1.07	0	0	3	0.30
5 min	1.38	0.92	0	1	3	0.88
1 hr	3.57	2.59	0	3.5	9	0.86

Figure 17: **Distribution of number of co-resident pairs and success rate or chances of co-residency for shorter delays under 20x20 run configuration in EC2.** A delay with 0+ means victim and attacker instances were launched sequentially, i.e. attacker instances were not launched until all victim instances were running. The results averaged are over 9 runs with 3 runs per time of day.

shorter non-zero delays in order to find the duration of this time window in which there are zero co-residency (results shown in Figure 17). We found that this time window is very short and that even a sequential launch of instances (denoted by 0+) could result in co-residency.

The second observation shows that non-zero delay on GCE and zero delay on Azure increases the chance of co-residency and hence directly benefits an attacker. It should be noted that on Azure, the launch delays between victim and attacker instances were longer than 1 hour due to our leap-frog experimental methodology; the actual delays between the VM launches were, on average, 3 hours (with a maximum delay of 10 hours for few runs). This higher delay was more common in runs with larger number of instances as there were significantly more false positives, which required a separate sequential phase to resolve (see Section 4.2).

We also experimented with longer delays on EC2 and GCE to understand whether and how quickly the chance of co-residency drops with increasing delay (results shown in Figure 18). Contrary to our expectation, we did not find the chance of co-residency to drop to zero even for delays as high as 16 and 32 hours. We speculate that the reason for this observation could be that the system was under constant churn where some neighboring VMs on the victim's machine were terminated. Note that our leap-frog methodology may, in theory, interfere with the VM placement. But it is noteworthy that we observed increased number of unique co-resident pairs with increasing delays, suggesting fresh co-residency with victim VMs over longer delays.

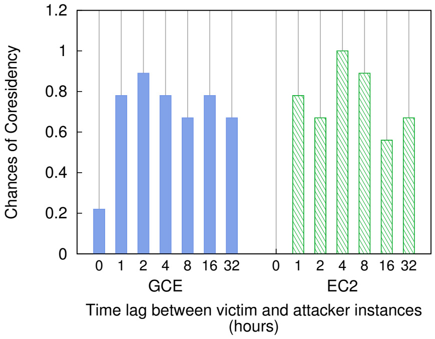


Figure 18: **Chances of co-residency over long periods.** Results include 9 runs over two weeks with 3 runs per time of day under the 20x20 run configuration. Note that we only conducted 3 runs for 32 hour delay as opposed to 9 runs for all other delays.

Chances of Co-residency			
Cloud	**Morning**	**Afternoon**	**Night**
	02:00 - 10:00	10:00 - 18:00	18:00 - 02:00
GCE	0.68	0.61	0.78
EC2	0.89	0.73	0.6

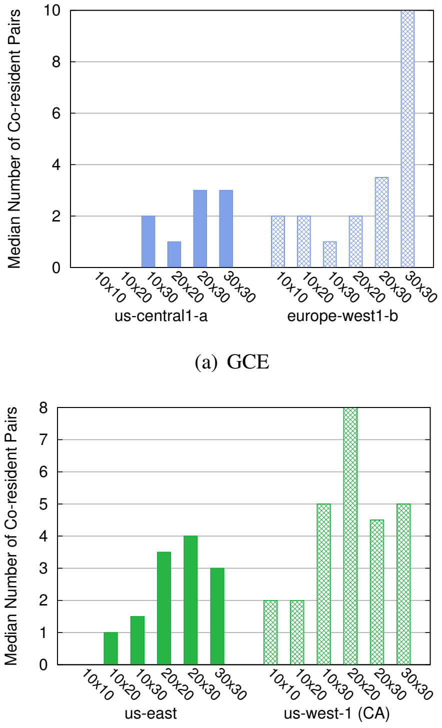
Figure 19: **Effect of time of day.** Chances of co-residency when an attacker changes the launch time of his instances. The results were aggregated across all run configurations with 1 hour delay between victim and attacker launch times. All times are in PT.

Effect of time of day. Prior works have shown that churn or load is often correlated with the time of day [31]. Our simple reference placement policy does not have a notion of load and hence have no effect on time of day. In reality, with limited number of servers in datacenters and limited number of capacity per host, load on the system has direct effect on the placement behavior of any placement policy.

As expected, we observe small effect on VM placement based on the time of day when attacker instances are launched (results shown in Figure 19). Specifically, there is a slightly higher chance of co-residency if the attacker instances are launched in the early morning for EC2 and at night for GCE.

5.4 Effect of Data Center Location

All the above experiments were conducted on relatively popular regions in each cloud (especially true for EC2 [31]). In this section, we report the results on other smaller and less popular regions. As the regions are less popular and have relatively fewer machines, we expect higher co-residency rates and more co-resident instances. Figure 20 shows the median number of co-resident VM



(a) GCE

(b) EC2

Figure 20: **Median number of co-resident pairs across two regions.** The box plot shows the median number of co-resident pairs excluding co-residency within the same account. Results are over at least 3 run per run configuration (x-axis).

pairs placed in these regions alongside the results for popular regions. The distribution of number of co-resident instances is not shown here in the interest of space.

The main observation from these experiments is that there is a higher chance of co-residency in these smaller regions than the larger, more popular regions. Note that we placed at least one co-resident pair in all the runs in these regions. Also the higher number of co-resident pairs also suggests a larger coverage over victim VMs in these smaller regions.

One anomaly that we found during two 20x20 runs on EC2 between 30^{th} and 31^{st} of January 2015, when we observed an unusually large number of co-resident instances (including three VMs from the same account). We believe this anomaly may be a result of an internal management incident in the Amazon EC2 us-west-1 region.

5.5 Other Observations

We report several other interesting observations in this section. First, we found more than two VMs can be co-resident on the same host on both Azure and GCE, but not on EC2. Figure 21 shows the distribution of number of co-resident instances per host. Particularly, in one of the runs, we placed 16 VMs on a single host.

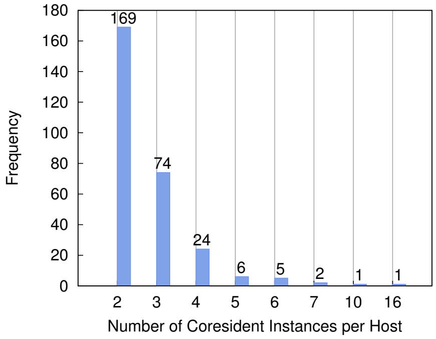


Figure 21: **Distribution of number of co-resident instances per host on Azure.** The results shown are across all the runs. We saw at most 2 instances per host in EC2 and at most 3 instances per host in GCE.

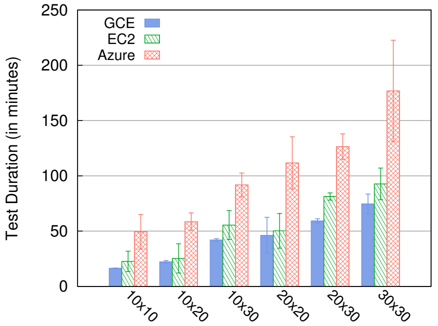


Figure 22: **Launch strategy and co-residency detection execution times.** The run configurations $v \times a$ indicates the number of victims vs. number of attackers launched. The error bars show the standard deviation across at least 7 runs.

Another interesting observation is related to co-resident instances from the same account. We term them as *self-co-resident instances*. We observed many self-co-resident pairs on GCE and Azure (not shown). On the other hand, we never noticed any self co-resident pair on EC2 except for the anomaly in us-west-1. Although we did not notice any effect on the actual chance of co-residence, we believe such placement behaviors (or the lack of) may affect VM placement.

We also experimented with medium instances and successfully placed few co-located VMs on both EC2 and GCE by employing similar successful strategies learned with small instances.

5.6 Cost of Launch Strategies

Recall that the cost of a launch strategy from Section 3, $C_S = a * P(a_{type}) * T_d(v,a)$. In order to calculate this cost, we need $T_d(v,a)$ which is the time taken to detect co-location with a attackers and v victims. Figure 22 shows the average time taken to complete launching attacker instances and complete co-residency detection for each run configuration. Here the measured co-residency detection is the parallelized version discussed in Section 4.2 and also includes time taken to detect co-residency within each tenant account. Hence, for these reasons the time to detect

Run	**Average Cost ($)**			**Maximum Cost ($)**		
config.	GCE	EC2	Azure	GCE	EC2	Azure
10x10	**0.137**	0.260	0.494	0.140	0.260	0.819
10x20	0.370	0.520	1.171	0.412	0.520	1.358
10x30	1.049	0.780	2.754	1.088	1.560	3.257
20x20	0.770	0.520	2.235	1.595	1.040	3.255
20x30	1.482	1.560	3.792	1.581	1.560	4.420
30x30	1.866	1.560	5.304	2.433	1.560	**7.965**

Figure 23: **Cost of running a launch strategy** (in dollars). Maximum cost column refers to the maximum cost we incurred out of all the runs for that particular configuration and cloud provider. The cost per hour of small instances at the time of this study were: 0.05, 0.026 and 0.06 dollars for GCE, EC2 and Azure, respectively. The minimum and maximum costs are in **bold**.

Run Config.	10x10	10x20	10x30	20x20	20x30	30x30
$\Pr[\mathbf{E}_a^v > 0]$	0.10	0.18	0.26	0.33	0.45	0.60

Figure 24: **Probability of co-residency under the reference placement policy.**

co-location is an upper bound for a realistic and highly optimized co-residency detection mechanism.

We calculate the cost of executing each launch strategy under the three public clouds. The result is summarized in Figure 23. Note that we only consider the cost incurred by the compute instances because the cost for other resources such as network and storage, was insignificant. Also note that EC2 bills every hour even if an instance runs less than an hour [16], whereas GCE and Azure charge per minute of instance activity. This difference is considered in our cost calculation. Overall, the maximum cost we incurred was about $8 for running 30 VMs for 4 hours 25 minutes on Azure and a minimum of 14 cents on GCE for running 10 VMs for 17 minutes. We incurred the highest cost for all the launch strategies in Azure because of overall higher cost per hour and partly due to longer tests due to our co-residency detection methodology.

5.7 Summary of Placement Vulnerabilities

In this section, we return to the secure reference placement policy introduced in Section 3 and use it to identify placement vulnerabilities across all the three clouds. Recall that the probability of at least one pair of co-residency under this random placement policy is given by $\Pr[\mathbf{E}_a^v > 0] = 1 - (1 - v/N)^a$, where $\mathbf{E}_a^v$ is the random variable denoting the number of co-location observed when placing a attacker VMs among $N = 1000$ total machines where v machines are already picked for the v victim VMs. First, we evaluate this probability for various run configurations that we experimented with in the public clouds. The probabilities are shown in Figure 24.

Strategy	v & a	a'	**Cost benefit ($)**	**Normalized Success**
S1 & S2	10	688	113.87	10
S3	30	227	32.75	1.67
S4(i)	20	105	4.36	2.67
S4(ii)	20	342	53.76	3.03
S5	20	110	4.83	1.48

Figure 25: **Cost benefit analysis.** $N = 1000$, $P(a_{type}) = 0.026$, which is the cost per instance-hour on EC2 (the cheapest). For simplicity $T_d(v,a) = (v * a) * 3.85$, where 3.85 is fastest average time to detect co-residency per instance-pair. Here, $v \times a$ is the run configuration of the strategy under test. Note that the cost benefit is the additional cost incurred under the reference policy, hence is equal to cost incurred by $a' - a$ additional VMs.

Recall that a launch strategy in a cloud implies a placement vulnerability in that cloud's placement policy if its normalized success rate is greater than 1. The normalized success rate of the strategy is the ratio of the chance of co-location under that launch strategy to the probability of co-location in the reference policy ($\Pr[\mathbf{E}_a^v > 0]$). Below is a list of selected launch strategies that escalate to placement vulnerabilities using our reference policy with their normalized success rate in parenthesis.

(S1) In Azure, launch ten attacker VMs closely after the victim VMs are launched (1.0/0.10).

(S2) In EC2 and GCE, if there are known victims in any of the smaller datacenters, launch at least ten attacker VMs with a non-zero delay (1.0/0.10).

(S3) In all three clouds, launch 30 attacker instances, either with no delay (Azure) or one hour delay (EC2, GCE) from victim launch, to get co-located with one of the 30 victim instances (1.00/0.60).

(S4) (i) In Amazon EC2, launch 20 attacker VMs with a delay of 5 minutes or more after the victims are launched (0.88/0.33). (ii) The optimal delay between victim and attacker VM launches is around 4 hours for a 20x20 run (1.00/0.33).

(S5) In Amazon EC2, launch the attacker VMs with 1 hour after the victim VMs are launched where the time of day falls in the early morning, i.e., 02:00 to 10:00hrs PST (0.89/0.60).

Cost benefit. Next, we quantify the cost benefit of each of these strategies over the reference policy. As the success rate of any launch strategy on a vulnerable placement policy is greater than what is possible in the reference policy, we need more attacker instances in the reference policy to achieve the same success rate. We calculate this number of attacker instances a' using: $a' = \ln(1 - S_a^v)/\ln(1 - v/N)$, where, S_a^v is the success rate of a strategy with run configuration of $v \times a$. The result of this calculation is presented in Figure 25. The result shows that the best strategy, S1 and S2, on all three cloud providers is $114 cheaper than what is possible in the reference policy.

It is also evident that these metrics enable evaluating and comparing various launch strategies and their efficacy on various placement policies both on robust placements and attack cost. For example, note that although the normalized success rate of *S3* is lower than *S4*, it has a higher cost benefit for the attacker.

5.8 Limitations

Although we exhaustively experimented with a variety of placement variables, the results have limitations. One major limitation of this study is the number of placement variables and the set of values for the variables that we used to experiment. For example, we limited our experiments with only one instance type, one availability zone per region and used only one account for the victim VMs. Although different instance types may exhibit different placement behavior, the presented results hold strong for the chosen instance type. The only caveat that may affect the results is if the placement policy uses account ID for VM placement decisions. Since, we experimented with only one victim account (separate from the designated attacker account) across all providers, these results, in the worst case, may have captured the placement behavior of an unlucky victim account that was subject to similar placement decisions (and hence co-resident) as that of the VMs from the designated attacker account.

Even though we ran at least 190 runs per cloud provider over a period of 3 months to increase statistical significant of our results, we were still limited to at most 9 runs per run configuration (with 3 runs per time of day). These limitations have only minor bearing on the results presented, if any, and the reported results are significant and impactful for cloud computing security research.

6 Related Work

VM placement vulnerability studies. Ristenpart et al. [29] first studied the placement vulnerability in public clouds, which showed that a malicious cloud tenant could place one of his VMs on the same machine as a target VM with high probability. Placement vulnerabilities exploited in their study include publicly available mapping of VM's public/internal IP addresses, disclosure of Dom0 IP addresses, and a shortcut communication path between co-resident VMs. Their study was followed by Xu et al. [33] and further extended by Herzberg et al. [25]. However, the results of these studies have been outdated by the recent development of cloud technologies, which is the main motivation of our work.

Concurrent with our work, Xu et al. [34] conducted a systematic measurement study of co-resident threats in Amazon EC2. Their focus, however, is in-depth evaluation of co-residency detection using network route traces and quantification of co-residence threats on older generation instances with EC2's classic networking (prior to Amazon VPC). In contrast, we study placement vulnerabilities in the context of VPC on EC2, as well as on Azure and GCE. The two studies are mostly complementary and strengthen the arguments made by each other.

New VM placement policies to defend against placement attacks have been studied by Han et al. [23, 24] and Azar et al. [18]. It is unclear, however, whether their proposed policies work against the performance and reliability goals of public cloud providers.

Co-residency detection techniques. Techniques for co-residency detection have been studied in various contexts. We categorize these techniques into one of the two classes: side-channel approaches to detecting co-residency with *uncooperative* VMs and covert-channel approaches to detecting co-residency with *cooperative* VMs.

Side-channels allow one party to exfiltrate secret information from another; therefore these approaches may be adapted in practical placement attack scenarios with targets not controlled by the attackers. Network round-trip timing side-channel was used by Ristenpart et al. [29] to detect co-residency. Zhang et al. [36] developed a system called *HomeAlone* to enable VMs to detect third-party VMs using timing side-channels in the last level caches. Bates et al. [19] proposed a side-channel for co-residency detection by causing network traffic congestion in the host NICs from attacker-controlled VMs; the interference of target VM's performance, if the two VMs are co-resident, should be detectable by remote clients. Kohno et al. [27] explored techniques to fingerprint remote machines using timestamps in TCP or ICMP based network probes, although their approach was not designed for co-residency detection. However, none of these approaches works effectively in modern cloud infrastructures.

Covert-channels on shared hardware components can be used for co-residency detection when both VMs under test are cooperative. Coarse-grained covert-channels in CPU caches and hard disk drives were used in Ristenpart et al. [29] for co-residency confirmation. Xu et al. [33] established covert-channels in shared last level caches between two colluding VMs in the public clouds. Wu el al. [32] exploited memory bus as a covert-channel on modern x86 processors, in which the sender issues atomic operations on memory blocks spanning multiple cache lines to cause memory bus locking or similar effects on recent processors. However, covert-channels proposed in the latter two studies were not designed for co-residency detection, while those developed in our work are tuned for this purpose.

7 Conclusion and Future Work

Multi-tenancy in public clouds enable co-residency attacks. In this paper, we revisited the problem of placement — can an attacker achieve co-location? — in modern public clouds. We find that while past techniques for verifying co-location no longer work, insufficient performance isolation in hardware still allows detection of co-location. Furthermore, we show that in the three popular cloud providers (EC2, GCE and Azure), achieving co-location is surprisingly simple and cheap. It is even simpler and costs nothing to achieve co-location in some PaaS clouds. Our results demonstrate that even though cloud providers have massive datacenters with numerous physical servers, the chances of co-location are far higher than expected. More work is needed to achieve a better balance of efficiency and security using smarter co-location-aware placement policies.

Acknowledgments

This work was funded by the National Science Foundation under grants CNS-1330308, CNS-1546033 and CNS-1065134. Swift has a significant financial interest in Microsoft Corp.

References

[1] Amazon ec2 instance store. `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/InstanceStorage.html`.

[2] Amazon elastic compute cloud. `http://aws.amazon.com/ec2/`.

[3] Apache libcloud. `http://libcloud.apache.org/`.

[4] Aws elastic beanstalk. `http://aws.amazon.com/elasticbeanstalk/`.

[5] Aws innovation at scale, re:invent 2014, slide 9-10. `http://www.slideshare.net/AmazonWebServices/spot301-aws-innovation-at-scale-aws-reinvent-2014`.

[6] Google compute engine. `https://cloud.google.com/compute/`.

[7] Google compute engine – disks. `https://cloud.google.com/compute/docs/disks/`.

[8] Google compute enginer autoscaler. `http://cloud.google.com/compute/docs/autoscaler/`.

[9] Haproxy: The reliable, high performance tcp/http load balancer. `http://www.haproxy.org/`.

[10] Heroku PaaS system. `https://www.heroku.com/`.

[11] Intel Ivy Bridge cache replacement policy. `http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/`.

[12] Olio workload. `https://cwiki.apache.org/confluence/display/OLIO/The+Workload`.

[13] Rightscale. `http://www.rightscale.com`.

[14] Virtual machine and cloud service sizes for azure. `https://msdn.microsoft.com/en-us/library/azure/dn197896.aspx`.

[15] Windows azure. `http://www.windowsazure.com/`.

[16] Amazon ec2 pricing, 2015. `http://aws.amazon.com/ec2/pricing/`.

[17] Amazon Web Services. Extend your it infrastructure with amazon virtual private cloud. Technical report, Amazon, 2013.

[18] Y. Azar, S. Kamara, I. Menache, M. Raykova, and B. Shepard. Co-location-resistant clouds. In *In Proceedings of the ACM Workshop on Cloud Computing Security*, pages 9–20, 2014.

[19] A. Bates, B. Mood, J. Pletcher, H. Pruse, M. Valafar, and K. Butler. Detecting co-residency with active traffic analysis techniques. In *Proceedings of the 2012 ACM Workshop on Cloud Computing Security Workshop*, pages 1–12. ACM, 2012.

[20] B. Beach. Virtual private cloud. In *Pro Powershell for Amazon Web Services*, pages 67–88. Springer, 2014.

[21] 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*. ACM, 2012.

[22] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2012.

[23] Y. Han, T. Alpcan, J. Chan, and C. Leckie. Security games for virtual machine allocation in cloud computing. In *Decision and Game Theory for Security*. Springer International Publishing, 2013.

[24] Y. Han, J. Chan, T. Alpcan, and C. Leckie. Virtual machine allocation policies against co-resident attacks in cloud computing. In *IEEE International Conference on Communications,*, 2014.

[25] A. Herzberg, H. Shulman, J. Ullrich, and E. Weippl. Cloudoscopy: Services discovery and topology mapping. In *2013 ACM Workshop on Cloud Computing Security Workshop*, pages 113–122, 2013.

[26] D. Kanter. L3 cache and ring interconnect. `http://www.realworldtech.com/sandy-bridge/8/`.

[27] T. Kohno, A. Broido, and K. Claffy. Remote physical device fingerprinting. In *Security and Privacy, 2005 IEEE Symposium on*, pages 211–225. IEEE, 2005.

[28] D. Mosberger and T. Jin. httperfa tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):31–37, 1998.

[29] T. Ristenpart, E. Tromer, H. Shacham, and S. 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*, pages 199–212. ACM, 2009.

[30] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift. Resource-freeing attacks: Improve your cloud performance (at your neighbor's expense). In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 281–292. ACM, 2012.

[31] L. Wang, A. Nappa, J. Caballero, T. Ristenpart, and A. Akella. Whowas: A platform for measuring web deployments on IaaS clouds. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 101–114. ACM, 2014.

[32] Z. Wu, Z. Xu, and H. Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *USENIX Security symposium*, pages 159–173, 2012.

[33] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting. An exploration of L2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 29–40. ACM, 2011.

[34] Z. Xu, H. Wang, and Z. Wu. A measurement study on co-residence threat inside the cloud. In *USENIX Security Symposium*, 2015.

[35] Y. Yarom and K. Falkner. Flush+reload: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium*, pages 719–732. USENIX Association, 2014.

[36] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, pages 313–328. IEEE Computer Society, 2011.

[37] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 305–316. ACM, 2012.

[38] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 990–1003. ACM, 2014.

[39] F. Zhou, M. Goel, P. Desnoyers, and R. Sundaram. Scheduler vulnerabilities and attacks in cloud computing. *CoRR*, abs/1103.0759, 2011.

A Measurement Study on Co-residence Threat inside the Cloud

Zhang Xu
The College of William and Mary
zxu@cs.wm.edu

Haining Wang
University of Delaware
hnw@udel.edu

Zhenyu Wu
NEC Laboratories America
adamwu@nec-labs.com

Abstract

As the most basic cloud service model, Infrastructure as a Service (IaaS) has been widely used for serving the ever-growing computing demand due to the prevalence of the cloud. Using pools of hypervisors within the cloud, IaaS can support a large number of Virtual Machines (VMs) and scale services in a highly dynamic manner. However, it is well-known that the VMs in IaaS are vulnerable to co-residence threat, which can be easily exploited to launch different malicious attacks. In this measurement study, we investigate how IaaS evolves in VM placement, network management, and Virtual Private Cloud (VPC), as well as the impact upon co-residence. Specifically, through intensive measurement probing, we first profile the dynamic environment of cloud instances inside the cloud. Then using real experiments, we quantify the impacts of VM placement and network management upon co-residence. Moreover, we explore VPC, which is a defensive network-based service of Amazon EC2 for security enhancement, from the routing perspective. On one hand, our measurement shows that VPC is widely used and can indeed suppress co-residence threat. On the other hand, we demonstrate a new approach to achieving co-residence in VPC, indicating that co-residence threat still exists in the cloud.

1 Introduction

Entering the era of cloud computing, Infrastructure as a Service(IaaS) has become prevalent in providing Information Technology (IT) support. IT giants such as Amazon [1], Microsoft [4], and Google [2] have deployed large-scale IaaS services for public usage. Employing IaaS, individual IT service providers can achieve high reliability with low operation cost and no longer need to maintain their own computing infrastructures. However, IaaS groups multiple third-party services together into one physical pool, and sharing physical resources with other customers could lead to unexpected security breaches such as side-channel [25] and covert channel [19] attacks. It is well-known that IaaS is vulnerable to the co-residence threat, in which two cloud instances (i.e., VMs) from different organizations share the same physical machine. Co-residence with the victim is the prerequisite for mounting a side-channel or covert-channel attack.

The security issues induced by co-residence threat have been studied in previous research. However, most previous works focus on "what an attacker can do" [14, 19, 25], "what a victim user should do" [24], and "what a cloud vendor would do" [12, 15, 26]. In contrast, to the best of our knowledge, this measurement work initiates one of the first attempts to understand how cloud service vendors have potentially reacted to co-residence threat in the past few years and explore potential new vulnerabilities of co-residence inside the cloud. While Amazon Elastic Compute Cloud (EC2) is the pioneer of IaaS, it has the largest business scale among mainstream IaaS vendors [11, 18]. Therefore, we focus our study on Amazon EC2. More specifically, our measurement is mainly conducted in the largest data center hosting EC2 services: the northern Virginia data center, widely known as US-East region.

In our measurement study, we first perform a 15-day continuous measurement on the data center using ZMap [10] to investigate the data center's business scale and some basic management policies. With the basic knowledge of the cloud, we explore how EC2 has adjusted VM placement along with its impact on security. We further evaluate how much effort an attacker needs to expend to achieve co-residence in different circumstances. Comparing our evaluation results with those from 2008 [14], we demonstrate that the VM placement adjustment made by EC2 during the past few years has mitigated the co-residence threat.

As network management plays a critical role in cloud performance and security, we also investigate how the networking management in EC2 has been calibrated to suppress co-residence threat. We conduct large scale trace-routing from multiple sources. Based on our measurements, we highlight how the current networking configuration of EC2 is different from what it was and demonstrate how such evolution impacts co-residence inside the cloud. In particular, we measure the change of routing configuration made by EC2 to increase the difficulty of cloud cartography. We also propose a new algorithm to identify whether a rack is connected with Top of Rack switch or End of Row switch. With this algorithm, we are able to derive the network topology of EC2, which is useful for achieving co-residence inside the cloud.

To provide tenants an isolated networking environment, EC2 has introduced the service of Virtual Private Cloud (VPC). While VPC can isolate the instances from the large networking pool of EC2, it does not physically isolate the instances. After profiling the VPC usage and the routing configurations in VPC, we propose a novel approach to speculating the physical location of an instance in VPC based on trace-routing information. Our experiments show that even if a cloud instance is hidden behind VPC, an adversary can still gain co-residence with the victim with some extra effort.

The remainder of the paper is organized as follows. Section 2 introduces background and related work on cloud measurement and security. Section 3 presents our measurement results on understanding the overview of Amazon EC2 and its basic management policies. Section 4 details our measurement on VM placement in EC2, including co-residence quantification. Section 5 quantifies the impact of EC2-improved network management upon co-residence. Section 6 describes VPC, the most effective defense against co-residence threat, and reveals the haunted co-residence threat in VPC. Section 7 proposes potential solutions to make the cloud environment more secure. Finally, Section 8 concludes our work.

2 Background and Related Work

To leverage physical resources efficiently and provide high flexibility, IaaS vendors place multiple VMs owned by different tenants on the same physical machine. Generally, a scenario where VMs from different tenants are located on the same physical machine is called co-residence. In this work, the definition of co-residence is further relaxed. We define two VMs located in the same physical rack as co-residence. Thus, two VMs located in the same physical machine is considered as machine-level co-residence, while two VMs located in the same rack is defined as rack-level co-residence.

2.1 Co-residence threat

The threat of co-residence in the cloud was first identified by Ristenpart et al. [14] in 2009. Their work demonstrates that an attacker can place a malicious VM co-resident with a target and then launch certain attacks such as side channel and covert channel attacks. Following Ristenpart's work, Xu et al. [20] studied the bit rate of cache-based covert channel in EC2. Wu et al. [19] constructed a new covert channel on a memory bus with a much higher bit rate, resulting in more serious threats in an IaaS cloud. Zhang et al. [25] proposed a new framework to launch side channel attacks as well as approaches to detect and mitigate co-residence threat in the cloud [24, 26]. Bates et al. [7] proposed a co-resident watermarking scheme to detect co-residence by leveraging active traffic analysis.

The reason we define different levels of co-residence is that some attacks do not require VMs to be located on the same physical machine, but rather in the same rack or in a higher level network topology. For instance, Xu et al. [23] proposed a new threat called power attack in the cloud, in which an attacker can rent many VMs under the same rack in a data center and cause a power outage. There are also some side channel and covert channel attacks that only require the co-residence in the same sub-network [5].

In parallel with our work, Varadarajan et al. [16] performed a systematical study on placement vulnerability in different clouds. While their work mainly stands at the attacker side to explore more effective launch strategies for achieving co-residence in three different clouds, our work performs an in-depth study to understand the evolution of cloud management and the impact on co-residence threat in Amazon EC2. The two complementary works both support the point that public clouds are still vulnerable to co-residence threat.

2.2 Measurement in the cloud

In contrast to the measurement on private clouds from an internal point of view[9], the measurement works on public data centers are mostly conducted from the perspective of cloud customers. Wang et al. [17] demonstrated that in a public cloud, the virtualization technique induces a negative impact on network performance of different instance types. The work of Xu et al. [21] measures network performance in Amazon EC2 and demonstrates a long tail distribution of the latency. Their work also analyzes the reason behind the long tails and proposes a new VM deployment solution to address this issue. Bermudez et al. [8] performed a large-scale measurement on Amazon AWS traffic. Their study shows that most web service traffic towards Amazon AWS goes to the data center in Virginia, U.S. Some recent studies [11, 18] measure how web services are deployed in public clouds. They found that although many top-ranked domains deploy their subdomains into the cloud, most subdomains are located in the same region or zone,

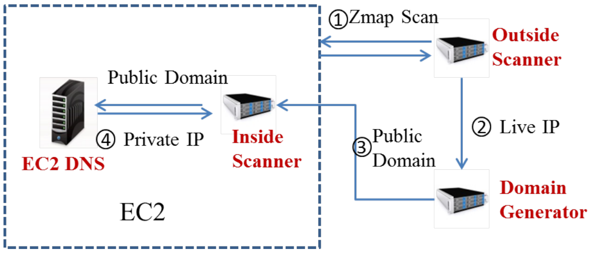


Figure 1: The system used to scan EC2.

resulting in a relatively poor fault tolerance.

In contrast to those measurement efforts, our study provides a measurement analysis from the perspective of security to reveal the management policies of a public cloud and their impact upon co-residence threat.

3 An Overview of EC2 Management

As the pioneer of IaaS, Amazon EC2 deploys its data centers all around the world, hosting the largest scale of IaaS business. In this section, we introduce some terminology in EC2 and provide an overview of the EC2 environment.

3.1 Instance type

An instance represents a virtual machine (VM) in the cloud, so we use the term "instance" and "VM" interchangeably throughout the rest of the paper. EC2 provides a list of instance types for clients to select while launching a new instance. The type of an instance indicates the configuration of the VM, determining the amount of resources the VM can use. The instance type is defined in the format *XX.XXX* such as *m1.small*. The first part of the instance type reveals the model of the physical server that will host this type of instance. The second part indicates the "size" of the VM, i.e., the amount of resources allocated to the instance. The detailed configuration of different instance types can be found at [3].

3.2 Regions and zones

Amazon EC2 has the concept of "region," which represents the physical area where the booted instance will be placed. Amazon has 9 locations around the world hosting EC2 services. Therefore, the instances in EC2 can be located in 9 regions: US east (northern Virginia), US west (Oregon), US west (northern California), South America (Sao Paulo), Asia Pacific southeast (Singapore), Asia Pacific southeast (Sydney), Asia Pacific northeast (Tokyo), EU west (Ireland), and EU central (Frankfurt). As pointed out in previous work [11], the majority of IaaS business is hosted in the US east region, e.g., in the data center located in northern Virginia. Most existing research on cloud measurement was conducted on this region [8, 13, 14]. Therefore, we also focus our study on the US east region. For the rest of the paper, we use the term "cloud" to mean the EC2 US east region and the term "data center" to mean the Amazon EC2 data center in northern Virginia, US.

In addition to regions, Amazon EC2 also allows clients to assign an instance to a certain "zone." A zone is a logical partition of the space within a region. Previous work shows that the instances in the same zone share common characters in private IP addresses, and likely instances within the same zone are physically close to each other [14, 19]. There are four availability zones in the US east region: us-east-1a, us-east-1b, us-east-1c, and us-east-1d.

3.3 Naming

The naming service is essential to cloud management. On one hand, the naming service can help customers to easily access their instances and simplify resource management. On the other hand, the naming service should help the cloud vendor to manage the cloud efficiently with high network performance.

In EC2, an instance is automatically assigned two domain names: one public and one private. The public domain name is constructed based on the public IP address of the instance, while the private domain name is constructed based on either the private IP address or the MAC address. Performing a DNS lookup outside EC2 returns the public IP of the instance, while performing a DNS lookup inside EC2 returns the private IP of the instance.

3.4 Scanning EC2 inside and outside

To better understand the environment and business scale of EC2, we performed a 15-day continuous measurement on the EC2 US east region.

Figure 1 illustrates our system to scan EC2. First we deployed a scanner outside EC2 to scan the cloud through a public IP address. Since EC2 publishes the IP range for its IaaS instances, our scanner uses ZMap [10] to scan the specified ranges of IP addresses. The ports we scanned include: ports 20 and 21 used for FTP, port 22 used for SSH, port 23 for telnet, ports 25 and 587 for SMTP, port 43 for WHOIS, port 53 for DNS, port 68 for DHCP, port 79 for Finger protocol, port 80 for HTTP, port 118 for SQL, port 443 for HTTPS, and port 3306 for MySQL. We also performed an ICMP echo scan. After scanning, our outside scanner obtained a list of live hosts in EC2 with the corresponding public IP addresses. In the next step, we performed automatic domain name generation. As mentioned above, the public domain name of an instance in EC2 can be derived using its public IP. This step produces a list of public domain names of live hosts. The generated public domain names were then sent to our inside scanner deployed inside EC2. Our inside scanner then performed DNS lookups for these domain names.

Due to the DNS lookup mechanism of EC2, the DNS server in EC2 answered the queries with the private IP addresses of the hosts. Reaching this point, our measurement system can detect live hosts in EC2 with their domain names, IP addresses, as well as the mapping between the public IP address and private IP address.

The scan interval is set to 20 minutes, which is a tradeoff between cost and accuracy. Scanning the entire EC2 US east region per port takes about 40 seconds, and we have 14 ports to scan. This means that scanning all the ports will take around 10 minutes. Note that our measurement also includes DNS lookups for all the detected live hosts. Performing these DNS lookups takes around 20 minutes, which is approximately the time for two rounds of scanning.

Our scanning measurement provides us an overview of the large business scale of EC2, the diversity of services, and the dynamic running environment. This scanning measurement also gives us the knowledge base to understand co-residence threat. The detailed results and analysis of our scanning measurement can be found in the Appendix A and B.

4 The Impact of VM Placement upon Co-residence

The VM placement policy of the cloud determines how easy or hard it is for an attacker to achieve co-residence. In this section, we present our measurement on VM placement and quantification of achieving co-residence. By comparing our measurement results with previous work, we demonstrate how the VM placement policy has been evolving in EC2 and its impact on mitigating co-residence threats.

4.1 Basic understanding of VM placement

We first launched a sufficiently large number of instances with different types in EC2. Then, we had two tasks to fulfill: (1) collecting networking (i.e., location) information of launched instances and (2) quantifying co-residence threat, i.e., given the current VM placement policy of EC2, how much effort an attacker needs to make to achieve co-residence. Since the process of achieving co-residence requires the knowledge of instance location, we can complete the two tasks together. For every instance we launched while seeking co-residence, we recorded its private IP address and public IP address. We also performed an automatic traceroute from the instance to its "neighbors" that share the /24 prefix with it. This information can provide us the basic knowledge of where the instances are placed.

During our measurement, we recorded the detailed information of 2,200 instances of type t1.micro, 1,800 instances of type m1.small, 1,000 instances of type m1.medium, 1,000 instances of type m3.medium, 80 instances of m3.large, and 40 instances of m3.xlarge. We selected some random samples from the instances we recorded to study the internal IP distribution. We investigated how private IP addresses are associated by the instance type and availability zones, i.e., whether the VM placement has type and zone locality. Our results demonstrate that currently EC2 still exhibits certain type and zone locality, i.e., instances with the same type in the same zone are more likely to be placed close to one another. However, compared with corresponding results in 2008 [14], such locality has been significantly weakened. More details of locality comparison can be found in Appendix C.

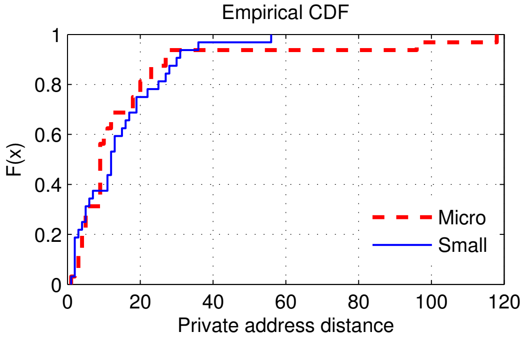


Figure 2: CDF of IP address distances between co-resident VMs.

After understanding the current VM placement in EC2, we further investigate co-residence threats in EC2.

4.2 Quantifying machine level co-residence

To understand how VM placement will affect co-residence, we assess the effort one needs to make to achieve machine level co-residence in two scenarios. The first scenario is to have a random pair of instances located on the same physical machine, and the second scenario is to have an instance co-reside with a targeted victim.

4.2.1 Random co-residence

To make our random co-residence quantification more comprehensive, we perform our measurement with different instance types and in different availability zones. Since zone us-east-1c is no longer hosting t1, m1, c1, and m3 instances, our measurement is performed in zone us-east-1a, us-east-1b, and us-east-1d. We achieve co-residence pairs with t1.micro, m1.small, m1.medium, and m3.medium. We did not achieve co-residence with large, xlarge or 2xlarge instances, because there are only 1 to 4 such large instances on one physical machine and it will be very difficult and costly to achieve co-residence with these types. Overall, we conduct 12 sets of experiments, with each set targeting a specific type of instances in a specific availability zone.

In each set of experiments, we perform rounds of co-residence probing until we find a co-residence pair.

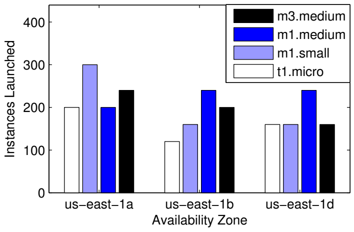


Figure 3: The service hour spent, i.e., the number of instances booted to achieve co-residence.

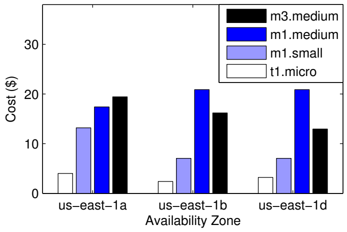


Figure 4: The financial cost (in US dollar) to achieve co-residence.

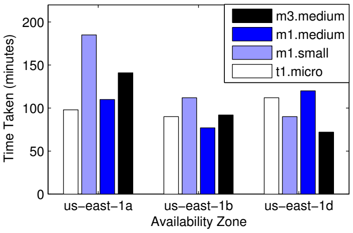


Figure 5: The time spent to achieve co-residence.

For the sake of robustness, EC2 has never placed instances from the same user on the same physical machine [14]. Therefore, we set up two accounts to launch instances simultaneously. Within one round, each account launches 20 instances, which will produce 400 pairs of co-residence candidates. Once a co-residence pair is verified, this set of experiments are terminated and the corresponding cost is recorded. If there is no co-residence pair found in this round, we move on to the next round by terminating all running instances and launching another 20 instances in each account, and then repeat the same procedure.

Given a pair of instances, verifying whether they are located on the same physical machine involves two steps: (1) pre-filtering unlikely pairs and (2) using a covert channel to justify co-residence.

For the first step, we need to screen out those pairs that are not likely to be co-resident to reduce probing space. Since the private IP address of an instance can indicate its physical location to some extent, and if the private IP addresses of two instances are not close enough, the two instances will have little chance to be co-resident. Based on this heuristic, we use the share of /24 prefix as the prerequisite of co-residence, i.e., if two instances do not share the /24 prefix, we consider them as not being co-resident and bypass the highly costly step 2. The rationale of setting the /24 prefix sharing as pre-filter is twofold:

1. First, the prerequisite of the /24 prefix sharing will not likely rule out any co-residence instance pairs. The number of instances that are hosted on the same physical machine is limited. Even for micro instances, there are no more than 32 instances running on a physical machine. For the instance type with larger size, there are even fewer instances running on a physical machine. In contrast, a /24 address space can contain 256 instances. Therefore, two co-resident instances are unlikely to be in different /24 subnets. Moreover, we obtained some co-residence pairs without any pre-filtering and recorded the private IP address distance between a pair of co-residence instances. Figure 2 illustrates the CDF of IP address distance between two co-residence instances. The distance is calculated as the difference between the two 32-bit integers of the two IP addresses. From the results we can figure out that most of these co-residence instances share the /27 prefix, which further confirms that the /24 prefix filtering will introduce very few, if any, false negatives.

2. Second, the prerequisite of sharing the /24 prefix can effectively narrow down the candidate space. Each time we use one account to launch 20 instances and use another account to launch another 20 instances, we will have 400 candidate pairs. During our measurement, we generated more than 40 rounds of such 400-pair batches. The average number of instance pairs that share the /24 prefix among 400 candidates is only 4. This means the /24 prefix sharing prerequisite can help us to screen out 99% of the candidates, which significantly accelerates the process of co-residence verification. During the 40 rounds of measurement, five co-residence pairs are observed.

The second step is to use a covert channel to verify whether two instances are actually located on the same physical machine. We use the technique introduced by Wu et al. [19] to construct a memory-bus-based covert channel between two instances. If the two instances can communicate with each other via the covert channel, then they are located on the same physical machine. This covert-channel-based verification can guarantee zero false positives.

The cost of achieving co-residence includes financial cost and time. According to the pay-as-you-go billing system, the financial cost is mainly determined by the service hours consumed during the co-residence probing. Every time an instance is launched, one billing hour is charged. Thus, the more probing instances an attacker needs to launch, the higher financial cost it will cause. In our experiments, we use only two accounts. In a real world attack, an attacker could use more accounts to launch the attack in parallel, which will result in less time required to achieve co-residence. However, under the same condition, regardless of attack process op-

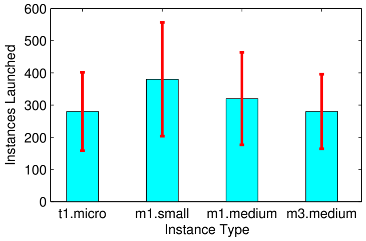


Figure 6: The service hour spent, i.e. the number of instances booted to achieve co-residence with a target.

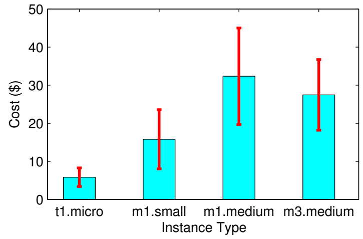


Figure 7: The financial cost (in US dollar) to achieve co-residence with a target.

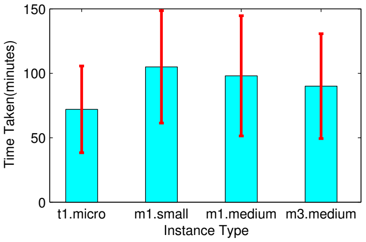


Figure 8: The time spent to achieve co-residence with a target.

timization, the time spent to achieve co-residence should have a positive correlation with the number of instances to launch, i.e., the more instances need to launch, the more time spent for detecting co-residence.

Figure 3 illustrates how many instances are required to achieve co-residence, while Figure 4 illustrates the actual financial cost. Figure 5 illustrates how much time it takes to achieve co-residence, i.e., the time cost. For each type of instance, the measurement repeats for five times and the mean value is shown in the figures. From the figures, it is evident that the cost for achieving co-residence of different types in different availability zones is quite different. Intuitively, as a larger instance has higher resource charge, it costs more money to achieve co-residence with those instances at a larger size. However, there is no such rule that the smaller size an instance is, the lower time cost we need to pay for co-residence.

4.2.2 Target co-residence

In the quantification of achieving co-residence with a particular target, we first randomly launched one instance with specific type from one account as the target. Then, from the other account, we also performed many rounds of co-residence probing until we found the instance that is co-resident with the target. The process of verifying co-residence remains the same. As demonstrated by the verification results of random co-residence above, different availability zones do not greatly impact the difficulty of achieving co-residence. Here we only show the results when our target instances are placed in zone us-east-1a.

Figures 6, 7 and 8 illustrate the number of instances to launch, the financial cost, and the time taken to achieve co-residence with a particular target, respectively. For each type of instance, the measurement is repeated for 15 times and the mean value is illustrated. The error bar with standard deviation is also shown in the figures. As is intuitive, achieving co-residence with a particular target requires launching more instances than achieving random co-residence. Getting a random co-residence pair requires launching 200 to 300 instances with two accounts (i.e., 100 to 150 instances per account), which can be done in 5 to 8 rounds. In contrast, achieving co-residence with a particular target requires launching 300 to 400 instances, which will take 15 to 20 rounds with each round launching 20 instances from one account. However, achieving co-residence with a particular target does not cost more time than achieving a random co-residence pair. The reason for this is simple: To get a random pair, we need to check 400 candidate pairs in each round, but to get a co-residence pair with a target, we only need to check 20 candidates in one round.

It is also possible that an attacker is unable to achieve co-residence with a certain target due to various reasons, e.g., the target physical machine reaches full capacity. During our study, we failed to achieve co-residence with two targets, one is m1.medium type and the other is m3.medium type. By failing to achieve co-residence we mean that after trying with more than 1,000 probing instances in two different days, we still cannot achieve co-residence with these two targets.

Overall, it is still very feasible to achieve co-residence in EC2 nowadays. However, an attacker needs to launch hundreds of instances to reach that goal, which may introduce considerable cost. In Section 4.4, we will compare our results to previous studies, demonstrating that achieving machine-level co-residence has become much more difficult than before, due to the change in cloud environments and VM placement policies.

4.3 Quantifying rack level co-residence

While covert channel and side channel attacks require an attacker to obtain an instance located exactly on the same physical machine with the victim, some malicious activities only need coarse-grained co-residence. Xu et al. [23] proposed a new attack called power attack. In their threat model, the attacker attempts to significantly increase power consumption of multiple machines connected by the same power facility simultaneously to trip the circuit breaker (CB). Since these machines located in the same rack are likely to be connected by the same CB, in a power attack the attack instances are not required to be placed on a same physical machine. Instead the attacker should place many instances within the same rack as the victim, i.e., achieving as much rack-level co-residence as possible. We performed measurement on how much effort is required to place a certain number of

Table 1: The number of co-residence pairs achieved by one round of probing in 2008 [14].

	Account A	Account B	Co-residence
Zone 1	1	20	1
	10	20	5
	20	20	7
Zone 2	1	20	0
	10	20	3
	20	20	8
Zone 3	1	20	1
	10	20	2
	20	20	8

instances under the same rack.

We first use one account to launch 20 instances, and then we check whether there are any instances in this batch that are located within the same rack. If there are no instances located in the same rack, we just randomly pick an instance and set its hosting rack as the target rack. Thanks to the Top of Rack(ToR) switch topology, verifying whether two instances are in the same rack is simple. Through a simple trace-routing, we can verify whether an instance has the same ToR switch with our target rack. This rack level co-residence can be further verified by performing trace-route from the candidate instance to the target instance. If the two instances are in the same rack, there should be only one hop in the trace, i.e., they are one hop away.

Figure 9 shows our measurement results. It is clear that an attacker can easily have multiple instances located within the same rack. The information of ToR switch helps the attacker quickly verify the rack-level co-residence. Since the malicious attack based on the rack-level co-residence is newly proposed [23], EC2 is unlikely to take any action to suppress rack-level co-residence.

4.4 Battle in VM placement

Table 1 lists the data from the original work on co-residence [14]. We can see that it was extremely easy to achieve co-residence in 2008. With two accounts each launching 20 instances, there were 7 or 8 co-residence pairs observed. In the 2012 work [19], the cost of achieving a co-residence instance pair is also briefly reported: A co-residence pair (micro) is achieved with 160 instances booted.

As we can see, nowadays it is much more difficult to achieve co-residence than in 2008 and 2012. EC2 could have adjusted its VM placement policies to suppress co-residence.

4.4.1 A larger pool

The business of EC2 is scaling fast, and thus it is intuitive that Amazon keeps deploying more servers into EC2. The measurement in 2008 [14] shows that there were three availability zones in the US east region. At present, the availability zones are expanded to four. Such expansion in availability zones also indicates that the business scale of EC2 is growing rapidly.

The measurement in 2008 [14] also shows 78 unique Domain0 IP addresses with 1785 m1.small instances, which means it only observed 78 physical machines that host m1.small service. Due to the evolution in EC2 management, we are no longer able to identify Dom0. However, we have identified at least 59 racks of servers that host m1.small instances. This suggests that the number of physical machines hosting m1.small instances is significantly larger than that in 2008. The enlarged pool provides EC2 with more flexibility to place incoming VMs, which is one of the reasons that it is now much more difficult to achieve co-residence than before.

4.4.2 Time locality

Time locality can help to achieve co-residence. Time locality means if two accounts launch instances simultaneously, it is more likely that some of these instances with time locality will be assigned to the same physical machine.

To verify whether such time locality exists in the current EC2, we performed another measurement. We set up four groups of experiments. In the first group, the two accounts always launch 20 VMs simultaneously. In the second group, the second account launches 20 VMs 10 minutes after the first account launches 20 VMs. In the third group, the launching time of the second account is one hour apart from that of the first account. In the fourth group, the second account launches VMs four hours after the first account. All instances are t1.micro type. In each group, the measurement terminates whenever a co-residence pair is observed and the number of instances required to achieve co-residence is recorded. All the experiments are repeated 5 times and the average is noted.

Figure 10 illustrates the number of instances required to achieve co-residence in each case. We can see that the efforts required to achieve co-residence do not vary significantly with the change of instance launching intervals. This implies that time locality seems to be very weak in the current EC2, which increases co-residence cost.

4.4.3 Dynamic assignment

In 2008, the IP addresses and instances in EC2 were assigned in a relatively static manner [14]. However, as we have demonstrated before, there are considerable mapping changes in our measurement, which indicates that the IP assignment has introduced a certain dynamism.

Meanwhile, in 2008, the instances were placed strictly based on the instance type, i.e., one physical machine can only host one type of instance [14]. In contrast, our measurement results show that such an assumption may not hold anymore. First, some small instances use internal IP addresses that were used by micro instances

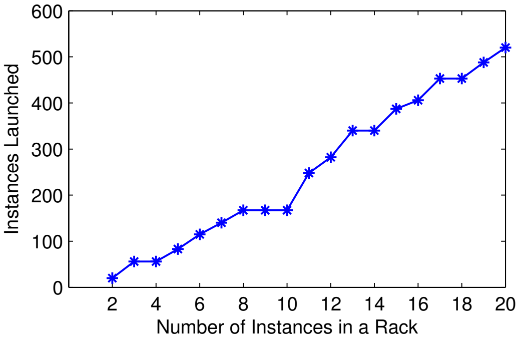


Figure 9: Instances launched to place certain number of instances within the same rack.

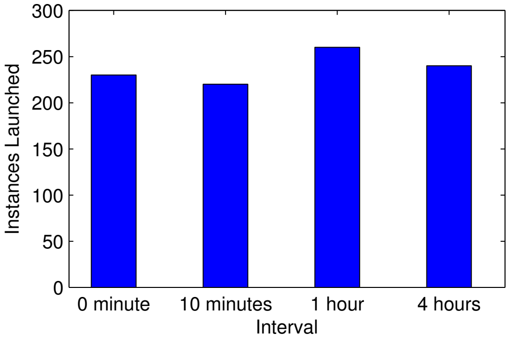


Figure 10: Effort to achieve co-residence with different time locality.

before. Second, during our measurement, by accident we observed that one live small instance has very close IP to a medium instance. We then attempted to build a covert channel between them. It turned out that the covert channel did work, which verifies that these two instances with different types are indeed located on a same physical machine. Following such an observation, in the rest of our rest measurement we also kept checking co-residence between different types of instances. Overall, five pairs of different-type co-residence instances are observed throughout our study. Our results indicate that in certain cases current VM placement policies in EC2 can mix different types of instances on one physical machine, potentially to reduce fragmentation. Such a policy also increases the difficulty of achieving co-residence.

5 The Impact of Network Management upon Co-residence

As network management plays a critical role in data center management, it has a significant impact on co-residence. On one hand, an attacker attempts to obtain as much networking information inside the cloud as possible to ease the gaining process of co-residence. On the other hand, the cloud vendors try to protect sensitive information while not degrading regular networking management and performance. In this section, we introduce the adjustments made by EC2 in network management during recent years to mitigate co-residence threat and the effectiveness of these approaches.

5.1 Methodology

To study the adjustment made by EC2 in network management, we performed large scale trace-routing. First, for the instances we booted, we performed "neighborhood trace-routing" from our instances to their "neighbors." Here we define neighbors as all those instances that share the /23 prefix of their private IP addresses with our source instances. Such trace-routing can inform us of the routing paths between an instance and other instances in the same rack and neighboring racks.

We next performed trace-routing from several of our instances (i.e., the instances we booted) to all the instances in a target list. We use the live host list from our scanning measurement (see Section 3.5 and Appendix A) as the target list. Trace-routing from our instances to over 650,000 target instances takes more than 8 days, but it can help us to understand network management in EC2 in a more comprehensive manner.

5.2 The evolution in routing configuration

The routing information has been leveraged to perform cloud cartography [14], which can further be used to launch co-residence-based attacks. However, our trace-routing results demonstrate that, as a response to cloud cartography, EC2 has adjusted its routing configurations to enhance security in the past few years. The adjustments we found are listed as follows.

5.2.1 Hidden Domain0

EC2 uses XEN as the virtualization technique in the cloud. According to the networking I/O mechanism of XEN [6], all the network traffic of guest VMs (instances) should travel through the privileged instance: Domain-0 (i.e, Dom0). Thus, Dom0 acts as the gateway of all instances on the physical machine, and all instances on this physical machine should have the same first-hop in their routing paths. Such Dom0 information provides an attacker with a very efficient probing technique: by simply checking the Dom0's IP addresses of two instances, one can know whether they are co-resident. Therefore, to prevent this Dom0 information divulgation, EC2 has hidden Dom0 in any and all routing paths, i.e. at present the Dom0 does not appear in any trace-routing results.

5.2.2 Hidden hops

To suppress cloud cartography enabled by trace-routing, EC2 has hidden certain hops in the routing paths. According to the work in May 2013 [13], traffic only needs to traverse one hop between two instances on the same physical machine and two hops between instances in

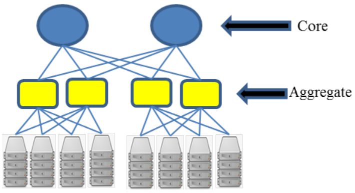


Figure 11: A common tree topology of a data center.

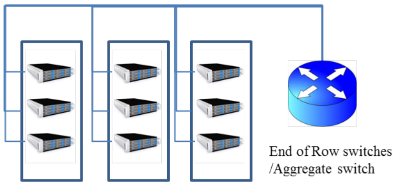


Figure 12: The topology with End of Row switch.

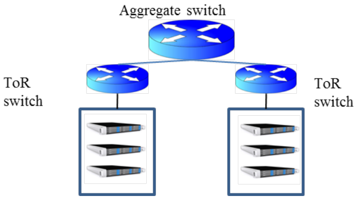


Figure 13: The topology with Top of Rack switch.

a same rack but not on the same physical machine. The paths between instances in different racks typically have 4 or 6 hops. However, our neighborhood trace-routing results show that the routing management has been changed in EC2.

First, a path of one hop does not necessarily indicate co-residence anymore. Our neighborhood trace-routing results show that an instance can have a very large number of 1-hop neighbors. For instance, one m1.small instance can have more than 60 1-hop neighbors. It is technically impractical to host so many instances on an m1 machine. To verify our hypothesis, we selected several pairs of instances with a 1-hop path and checked co-residence using covert channel construction. Our co-residence verification fails for most of these pairs, confirming that two instances with a 1-hop path do not necessarily co-locate on the same physical machine. This observation indicates that EC2 even hides the ToR switches in the routing path in some cases, leaving only one hop in the path between two instances in the same rack.

Second, we observed many odd-hop paths, accounting for 34.26% of all paths. In contrast, almost all the paths in the measurement conducted in May 2013 are even-hop [13]. This indicates that the network configuration of EC2 has changed since May 2013.

Third, the ToR switch of a source instance is shown as the first hop in the path, which indicates that the ToR switch should be an L3 router. However, we cannot observe the ToR switch of a target instance in the traces, implying that EC2 has configured the ToR switch to hide itself in the incoming traffic to the rack. Moreover, among our traces, we observed that 76.11% of paths have at least one hop filled with stars. The hops filled with stars can be a result of the configuration of certain devices such as L2 switches; it is also possible that EC2 has deliberately obscured those hops for security reasons. These paths with invisible or obscured hops significantly increase the difficulty of conducting cloud cartography.

5.3 Introducing VPC

To suppress the threat from internal networks, EC2 proposes a service called Virtual Private Cloud (VPC). VPC is a logically isolated networking environment that has a separate private IP space and routing configuration. After creating a VPC, a customer can launch instances into its VPC, instead of the large EC2 network pool. The customer can also divide a VPC into multiple subnets, where each subnet can have a preferred availability zone to place instances.

Moreover, EC2 provides instance types that are dedicated for VPC instances. These instance types include t2.micro, t2.small, and t2.medium. According to the instance type naming policy, instances with t2 type should be placed on those physical servers with the t2 model.

An instance in a VPC can only be detected through its public IP address, and its private address can never be known by any entity except the owner. Therefore, within a VPC, an attacker can no longer speculate the physical location of a target using its private IP address, which significantly reduces the threat of co-residence.

5.4 Speculating network topology

Besides routing configuration, the knowledge of network topology also helps to achieve co-residence, especially for high level co-residence such as rack-level. Figure 11 depicts the typical network topology in a data center. The core and aggregation switches construct a tree topology. Before connecting to the aggregate switches, there are two mainstream ways to connect servers in a rack/racks: End of Row (EoR) switches and Top of Rack (ToR) switches.

For EoR switches, as illustrated in Figure 12, servers of several racks are connected to the same EoR switch. To be more precise, an EoR switch can be a switch array including a group of interconnected switches. These switches can function as aggregate switches themselves. For ToR switches, as illustrated in Figure 13, all servers in a rack are first connected to a separate ToR switch, and then the ToR switch is connected to aggregate switches. Such a topology has currently become the mainstream network topology in a data center.

There are several variants of EoR topology, such as Middle of Rack (MoR) and ToR switch with EoR management. Meanwhile, there are other potential topologies such as OpenStack cluster in a data center. Therefore, we classify the network topology of a rack/racks into two classes: ToR connected and non-ToR connected. To identify whether a rack uses a ToR switch or a non-ToR switch, we analyze the neighborhood trace-routing results of multiple instances. Based on our analysis, we proposed a method to identify the network topology of a rack, ToR-connected or non-ToR-connected.

ToR-connected: a rack that deploys ToR switches must satisfy all of the following conditions:

1. For an instance A in the rack, there should be at least one instance B that is only one hop away from A.
2. For an instance A in the rack, there should be at least 8 instances that are two hops away from A.
3. For any two instances A and B, if (i) conditions 1 and 2 hold for both A and B, (ii) the trace-routing path between A and B has no more than two hops, and (iii) for any instance C, the first hop in the trace-routing path from A to C is the same as the first hop in the path from B to C, then A and B are considered as being in the same ToR rack.
4. For an instance A in the rack, for any trace-routing path with A as source and length larger than 2, the first hop in the path should share the /16 prefix with the private IP address of A.

The IP address of the first hop (i.e., ToR switch's IP address) is used to differentiate two ToR racks.

Non-ToR-connected: a rack that deploys non-ToR switches must satisfy all of the following conditions:

1. For an instance A in the rack, there should be no instance B such that the path between A and B has two hops.
2. For an instance A in the rack, for any instance B in EC2, either (i) A and B are machine-level co-resident and the path between A and B has only one hop or (ii) the path between A and B has more than two hops.
3. For two instances A and B, if (i) conditions 1 and 2 hold for both A and B, (ii) A and B share the /24 prefix of their private IP, (iii) the trace-routing path between A and B has 4 or 6 hops, and (iv) for any instance C, the first hop in the path between A and C is the same as the first hop in the path between B and C, then A and B are considered as being in the same non-ToR rack.
4. For an instance A in the rack, for any trace-routing path with A as source and length larger than 2, the first hop in the path should not share the /20 prefix with the private IP address of A.

Again, the IP address of the first hop is used to differentiate two non-ToR racks.

In EC2, there are two "generations" of instances. The old generation carries all the instances with m1 type, and the new generation covers all the instances with other types. We applied our method on m1.small, m1.medium, m3.medium, and m3.large type, which cover both old-generation instances and new-generation instances.

Overall, we identified 59 distinct racks that host m1.small instances, 18 racks that host m1.medium instances, 22 racks that host m3.medium instances, and 10 racks that host m3.large instances. Among the 109 racks, there are only 14 racks identified as non-ToR-connected while the rest are ToR-connected. Among the 14 non-ToR racks, we observed 12 old-generation racks, in which 7 racks host m1.small instances and 5 racks host m1.medium instances, and only 2 new-generation racks host m3.medium instances.

Our results demonstrate that while both ToR racks and non-ToR racks exist in EC2, ToR-connected is the dominating topology in EC2. Moreover, it is evident that new-generation machines are more likely to be located in the ToR-connected topology, indicating that the ToR-connected topology has become the main trend. While the ToR-connected topology is easy to manage, the routing information is very straightforward since the first hop reveals which rack the instance is in. Such information can be leveraged by an attacker to achieve rack-level co-residence.

6 A New Battle in VPC

Using VPC, customers can protect their instances in an isolated network environment. However, VPC only logically isolates the networks. The instances from different VPCs may still share the same physical machine, leaving the opportunity to achieve co-residence. In this section, we first take an overview on the usage of VPC in EC2, and then we introduce a new method to attack instances that are hidden behind VPCs.

6.1 The overview of VPC usage

For those instances in the default networks of EC2, our inside scanner can obtain their private addresses via DNS lookups. However, the DNS query for an instance in a VPC will only return its public IP address. Therefore, the instances in a VPC can be easily identified by checking the DNS query results of our inside scanner, i.e., any instance whose private IP address cannot be detected by our inside scanner is an instance in a VPC. Figure 14 shows the VPC usage in EC2. As we can see, all instances in VPC are assigned public IP addresses in five different ranges: 107.20.0.0/14, 184.72.64.0/18, 54.208.0.0/15, 54.236.0.0/15, and 54.80.0.0/13. This implies that all instances in a VPC are managed in a uniform manner. On average, in each round of our probing we can observe 115,801 instances in a VPC, which are around 17% of all live instances observed, demonstrating that VPC is widely used in EC2 to protect instances.

6.2 Routing paths of VPC instances

Since a VPC should be treated as a private network, the routing policies for instances inside a VPC must be different from those in the default EC2 network. This routing difference can help us further understand the management of a VPC. To connect a VPC to the public Internet, a

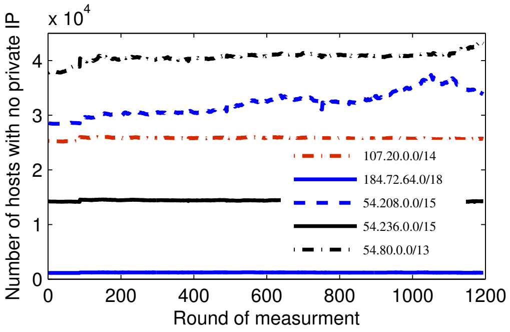


Figure 14: The live instances in VPCs.

customer must create a gateway and attach it to the VPC. The gateway must be included into the route table of the VPC. All traffic from or to the Internet must go through the gateway, but the traffic inside EC2 does not require the gateway to be involved.

Besides the basic understanding of the routing configuration of a VPC, we also need to know how a VPC is connected with the default EC2 network and other VPCs. We created several VPCs with two different accounts. The instances with different types are launched into these VPCs. Trace-routing is performed in four different ways: (1) trace-routing from an instance in a VPC to another instance in the same VPC, (2) trace-routing from an instance in a VPC to an instance in another VPC, (3) trace-routing from an instance in a VPC to an instance in the default EC2 network, and (4) trace-routing from an instance in the default EC2 network to an instance in a VPC.

6.2.1 Routing within VPC

Routing inside the same VPC is expected to be simple. We performed trace-routing between two instances in the same VPC, using both private and public IP addresses. The results show that trace-routing with private IP or public IP addresses will yield different routing paths. If trace-routing is performed with the private IP of the target instance, the result path has only one-hop, i.e., the direct connection to the destination, which is reasonable. However, if trace-routing is performed with the public IP of the target, trace-routing will return two hops with the first hop obscured with stars. Apparently, EC2 intentionally hides some routing information. The routing information between the two instances within the same VPC is made transparent to customers. Such obscuration disables a customer from speculating the physical location of the instances.

As discussed in Section V, even within the same VPC, two instances can be located in different "subnets." We also performed trace-routing between two instances in the same VPC but in different subnets. The resulting paths do not differ from the paths between two instances within the same subnet.

6.2.2 Routing between VPCs

The traffic between instances in different VPCs should traverse multiple switches and routers. Surprisingly, we found that any routing path between any two instances in any two different VPCs only has two hops: the first hop is obscured and the second hop is the destination. EC2 once again obscures the routing path between VPCs to prevent an adversary from revealing sensitive information of a VPC, e.g., the IP address of a gateway.

6.2.3 Routing from VPC to default EC2 network

Although instances in a VPC no longer share a private network with the default pool of EC2, the switches/routers that connect VPCs might still be physically connected to the other switches/routers in the data center. How EC2 routes the traffic between instances in a VPC and instances in the default EC2 network can reveal its network topology to some extent. Figure 15 shows a sample trace-routing result from an instance in a VPC to an instance in the default EC2 network. We can see that the first two hops of the path are obscured. This prevents us from knowing the switch/router that connects the VPC, thereby hiding the physical location of VPC instances. However, we can still see parts of the path and can infer the end-to-end latency based on the trace-routing result.

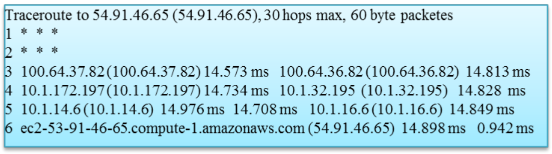
6.2.4 Routing from default EC2 network to VPC

Figure 16 shows a sample trace-routing result from an instance in the default EC2 network to an instance in a VPC. The path is almost symmetric to the path from a VPC to the default EC2 network. Again, the last two hops before reaching the destination are obscured to hide the information of the router/switch.

Overall, EC2 manages a VPC in a transparent fashion, i.e., to a customer it should look like all instances in a VPC are connected by a dedicated switch, just like a real private network. However, instances in the same VPC are not physically located together. These instances are still located in different racks and are connected to different ToR or EoR switches. Thus, the traffic inside a VPC might still traverse multiple switches/routers. Similarly, the traffic between an instance in a VPC and an instance in the default EC2 network can have a similar path to the traffic between two instances in the default EC2 network. However, EC2 hides or obscures certain hops in the path to provide the image of "private network."

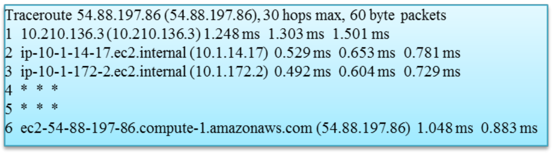
6.3 Co-residence in VPC

The traditional way of achieving co-residence relies on the knowledge of private IP address to seek potential candidates. With VPC, this approach no longer works as VPC hides the private IP address of an instance. An alternative is to infer the physical location of a target based on



```
Traceroute to 54.91.46.65 (54.91.46.65), 30 hops max, 60 byte packetes
1 * * *
2 * * *
3  100.64.37.82 (100.64.37.82) 14.573 ms   100.64.36.82 (100.64.36.82)  14.813 ms
4  10.1.172.197 (10.1.172.197) 14.734 ms   10.1.32.195 (10.1.32.195)  14.828  ms
5  10.1.14.6 (10.1.14.6)  14.976 ms  14.708 ms   10.1.16.6 (10.1.16.6)  14.849 ms
6  ec2-53-91-46-65.compute-1.amazonaws.com (54.91.46.65)  14.898 ms   0.942 ms
```

Figure 15: A sample trace-routing result from an instance in VPC to an instance in EC2.



```
Traceroute 54.88.197.86 (54.88.197.86), 30 hops max, 60 byte  packets
1  10.210.136.3 (10.210.136.3) 1.248 ms  1.303 ms  1.501 ms
2  ip-10-1-14-17.ec2.internal (10.1.14.17)  0.529 ms  0.653 ms  0.781 ms
3  ip-10-1-172-2.ec2.internal (10.1.172.2)  0.492 ms  0.604 ms  0.729 ms
4 * * *
5 * * *
6  ec2-54-88-197-86.compute-1.amazonaws.com (54.88.197.86)  1.048 ms  0.883 ms
```

Figure 16: A sample trace-routing result from an instance in EC2 to an instance in VPC.

the routing paths to the target. Unfortunately, our trace-routing results show that sensitive information of a routing path is obscured by EC2, and therefore it also does not work well.

However, in our trace-routing results we found that the end-to-end latency to and from an instance in a VPC varies with different instance types and the location of the instance. This latency variation can be leveraged to help an attacker speculate the type and location of a target instance. Moreover, while performing trace-routing between an instance in a VPC and an instance in the default EC2 network, the number of hops required is not obscured. Therefore, the number of hops in a path can also be leveraged to derive useful information for achieving co-residence.

Based on our measurement analysis, we propose a new method to achieve co-residence with instances in a VPC. It has two steps: (1) speculate the type and availability zone of a target and (2) launch probing instances with the same type in the same availability zone and perform co-residence verification.

6.3.1 Type and zone speculation

We collected statistical data of the end-to-end latency between a pair of instances with different types and in different zones. Table 2 shows part of the end-to-end latency statistics. Each row represents an instance in a VPC with a certain type and availability zone preference. Each column stands for an instance in the default EC2 network with a certain type and availability zone preference. Each value in the table is calculated as the average of 50 samples. Each sample is obtained with a distinct instance pair and is averaged over five rounds of latency measurement. With this latency table, we are able to construct a latency vector for each target instance in a VPC and use the latency vectors to speculate the type and availability zone of a target.

There are three availability zones and each zone has six types: t1.micro, m1.small, m1.medium, m1.large, m3.medium, and m3.large. Thus, the complete version of Table 2 has 18 rows and 18 columns, which can be found in our technical report [22]. Note that each row in the table can represent a latency vector, and such a latency vector derived from our controlled sampling is called a *baseline vector*.

In each different availability zone, we randomly select an instance for each different type, resulting in 18 (3×6) sample instances in total for testing type and zone speculation. For each target in a VPC, we perform trace-routing from each of our sample instances to the target for 5 times and record the average end-to-end latency of each pair. Such measurement can provide us 18 end-to-end latency values, which constitute an input vector of length 18. We then calculate the cosine similarity between the input vector and these 18 baseline vectors. The baseline latency vector that has the highest similarity with the target input vector is selected, and we can speculate that the target instance has the same {instance type, availability zone} as the instance in the selected baseline vector.

6.3.2 Verifying co-residence

To achieve co-residence with an instance in a VPC, our probing instances are also launched in a VPC. There are two reasons that we do not use the instances in the default EC2 network as probing instances. First, it is possible that EC2 uses a separate algorithm to place instances in a VPC. In other words, compared to an instance in the default EC2 network, an instance in a VPC may have a better chance to achieve co-residence with an instance in another VPC. Second, as we have observed, the end-to-end latency between two instances in two different VPCs is more stable than the latency between an instance in the default EC2 network and an instance in a VPC, which allows us to leverage latency for pre-filtering.

Similar to verifying co-residence in the default EC2 network, verifying co-residence in a VPC also includes two steps: pre-filtering and covert channel construction. While the way of using covert channel construction to confirm co-residence remains the same, the pre-filtering process in a VPC is different.

To verify whether an attack instance is co-resident with a target, we rely on two rounds of pre-filtering to screen out irrelevant candidates. First, we perform trace-routing from our 18 sample instances to our attack instance and the target instance. If any path from the sample instance to the attack instance is not equivalent to the corresponding path from the sample instance to the target in terms of number of hops, this attack instance is abandoned.

Second, if all the paths match in the number of hops, we measure end-to-end latency between our attack instance and the target instance. Figure 17 shows a sample latency distribution between an instance in a VPC

Table 2: End to end latency between different instances.

	1a-t1.micro	1a-m1.small	1a-m1.medium	1b-t1.micro	1b-m1.small	1b-m1.medium
1a-t1.micro	1.224ms	1.123ms	1.025ms	2.237ms	2.221ms	2.304ms
1a-m1.small	1.361ms	1.059ms	1.100ms	2.208ms	2.055ms	2.198ms
1a-m1.medium	1.165ms	1.102ms	0.986ms	2.211ms	2.060ms	1.988ms
1b-t1.micro	2.101ms	2.235ms	2.188ms	1.108ms	1.243ms	1.202ms
1b-m1.small	2.202ms	2.003ms	2.190ms	1.131ms	0.968ms	1.048ms
1b-m1.medium	2.087ms	2.113ms	1.965ms	1.088ms	1.023ms	0.855ms

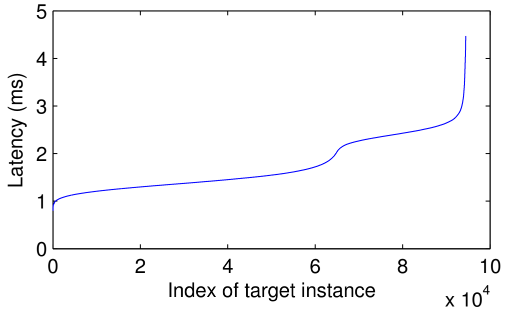


Figure 17: End-to-end latency between an instance in VPC and all other instances in other VPCs in EC2.

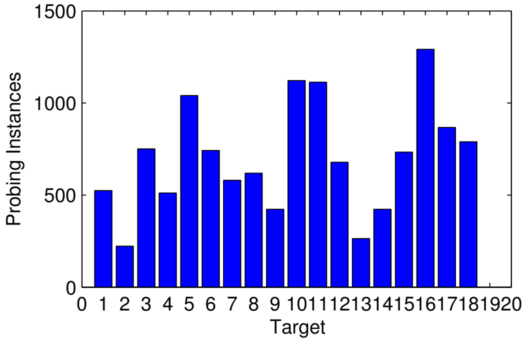


Figure 18: The effort for co-residence with instances in VPC.

with the micro type in availability zone 1a to all live VPC instances in EC2. As we can see, most end-to-end latency values (over 99%) are above 1ms, and in very rare cases (below 0.1%) the latency is below 0.850ms. We perform such latency measurement from 18 sample VPC instances with different types in different availability zones, and similar distribution is repeatedly observed. Based on such observations and the heuristics that instances located on the same physical machine should have lower latency than instances located in a different physical location, we set a latency threshold for each type of instance in each availability zone. The threshold is selected so that for an instance in a VPC with certain type and availability zone, the end-to-end latency between the instance and 99.9% of all other VPC instances should be above the threshold. For example, based on our measurement introduced above, if we speculate that the target VPC instance is located in availability zone 1a with micro type, the latency threshold is set to 0.850ms. Only if the end-to-end latency between a probing instance and a target instance is below the threshold, will the probing instance be considered as a co-residence candidate.

If the probing instance passes the two rounds of filtering, we will perform covert-channel construction to confirm co-residence.

6.4 VPC co-residence evaluation

To verify the feasibility of our VPC co-residence approach, we conducted a series of experiments in EC2. We first tested whether our approach can speculate the type and availability zone of a target instance correctly. We launched VPC instances in three availability zones with six different types. For each combination, 20 instances were launched. We applied our approach to speculate the type and availability zone of the target. If both the type and availability zone are correctly inferred, we consider that the target instance is correctly identified. Table 3 lists our evaluation results. Each number in the table indicates the number of the successfully identified instances among the 20 launched instances for a zone-type combination (e.g., 1a-t1.micro means t1.micro instances launched in the us-east-1a zone). The results show that our type/zone speculation can achieve an accuracy of 77.8%.

We then evaluated the overall effectiveness of our approach for achieving co-residence. We launched 40 instances in one VPC, with different types and availability zones. We performed the full process of achieving co-residence with VPC instances.

First, we measured the effectiveness of our two-stage filtering technique. Among all the probing instances we launched, 63.2% of them did not pass the first step filtering. For the second stage, our technique filtered out 97.9% of the instances that passed the first stage filtering. For all the instances passed the two-stages filtering, 17.6% of them passed the covert-channel verification, which are the instances actually co-resident with the target.

Eventually, among 40 instances, we successfully achieved co-residence with 18 of them. Figure 18 illustrates the effort we paid to achieve co-residence, showing that to achieve co-residence in VPC is not an easy task. An attacker may need to launch more than 1,000 probing instances and such a process can take many hours.

Overall, we are the first to demonstrate that an attacker can achieve co-resident with a target inside a VPC with high cost, and hence VPC only mitigates co-residence threat rather than eliminating the threat all together.

Table 3: The number of successfully identified targets.

	1a-t1.micro	1a-m1.small	1a-m1.medium	1a-m1.large	1a-m3.medium	1a-m3.large
Success	16	13	18	14	16	17
	1b-t1.micro	1b-m1.small	1b-m1.medium	1b-m1.large	1b-m3.medium	1b-m3.large
Success	13	13	19	16	20	17
	1d-t1.micro	1d-m1.small	1d-m1.medium	1d-m1.large	1d-m3.medium	1d-m3.large
Success	12	18	15	13	14	18

7 A More Secure Cloud

Based on our measurement analysis, we have proposed some guidelines towards more secure IaaS cloud management.

First, the cloud should manage the naming system properly. In general, a domain name is not sensitive information. However, EC2's automatic naming system reveals its internal space. In contrast, Azure and Rackspace employ flexible naming systems that can prevent automatic location probing. However, automatic domain name generation is more user-friendly since it allows a user to launch instances in batch, while a customer can only launch instances one by one in Azure and Rackspace. Moreover, automatic domain name generation can help an IaaS vendor manage the cloud more efficiently. To balance management efficiency and security, we suggest that IaaS clouds integrate automatic domain name generation with a certain randomness. For example, a random number that is derived from the customer's account information can be embedded into the EC2 default domain name. This improved naming approach can prevent location probing while not degrading management efficiency.

Second, it is controversial to publish all IP ranges of a cloud. With the introduction of ZMap [10], it is not difficult to scan all public IPs in the cloud. We have demonstrated that such scanning can cause serious security concerns.

Third, the routing information should be well-protected. While trace-routing is a tool for a customer to diagnose a networking anomaly, it can also be exploited by an attacker to infer the internal networking information of the cloud. However, the approach taken by Azure and Rackspace is too strict. The prohibition of networking probing deprives a customer from self-diagnosis and self-management. A good trade-off is to show only part of the paths, but always obscure the first hop (ToR) and the last second hop.

Fourth, VM placement should be more dynamic and have more constraints. Locality reduction will make it more difficult for an attacker to locate a target. IaaS vendors can also leverage some historical information of a user's account to prevent the abuse of launching instances. While EC2 has significantly increased the difficulty of achieving machine-level co-residence, it is also necessary to suppress rack-level co-residence in the future.

8 Conclusion

We have presented a systematic measurement study on the co-residence threat in Amazon EC2, from the perspectives of VM placement, network management, and VPC. In terms of VM placement, we have demonstrated that time locality in VM placement is significantly reduced and VM placement in EC2 becomes more dynamic, indicating that EC2 has adjusted its VM placement policy to mitigate co-residence. Regarding network management, by conducting a large-scale trace-routing measurement, we have shown that EC2 has refined networking configurations and introduced VPC to reduce the threat of co-residence. We have also proposed a novel method to identify a ToR-connected or non-ToR-connected topology, which can help an attacker to achieve rack-level co-residence. As the first to investigate the co-residence threat in VPC, on one hand, we have confirmed the effectiveness of VPC in mitigating the co-residence threat. On the other hand, we have shown that an attacker can still achieve co-residence by exploiting a latency-based probing method, indicating that VPC only mitigates co-residence threat rather than eliminating the threat.

9 Acknowledgement

We would like to thank our shepherd Chris Grier and the anonymous reviewers for their insightful and detailed comments. This work was partially supported by ONR grant N00014-13-1-0088.

References

[1] Amazon elastic compute cloud (ec2). `http://aws.amazon.com/ec2/`.

[2] Google cloud platform. `https://cloud.google.com/compute/`.

[3] Instance types in ec2. `http://aws.amazon.com/ec2/instance-types/`.

[4] Microsoft azure services platfor. `http://www.microsoft.com/azure/default.mspx`.

[5] Aviram, A., Hu, S., Ford, B., and Gummadi, R. Determinating timing channels in compute clouds. In *Proceedings of ACM CCSW'10*, pp. 103–108.

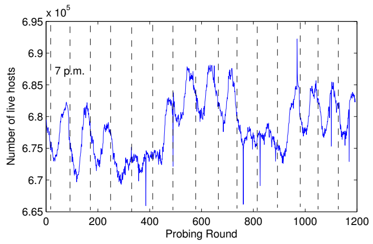


Figure 19: Number of live instances in EC2 US east region.

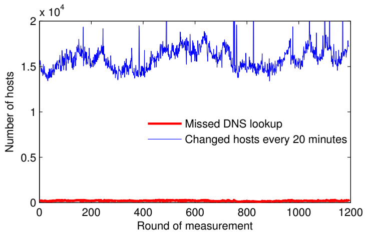


Figure 20: Number of changed instances between each round of measurement and number of missed DNS lookups in each round.

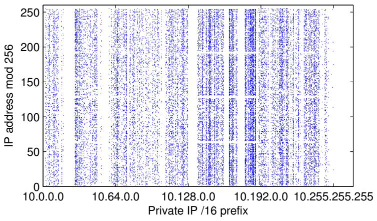


Figure 21: The private IP addresses whose mappings to public IP have changed.

[6] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *Proceedings of ACM SOSP'03*.

[7] Bates, A., Mood, B., Pletcher, J., Pruse, H., Valafar, M., and Butler, K. Detecting co-residency with active traffic analysis techniques. In *Proceedings of ACM CCSW'12*.

[8] Bermudez, I., Traverso, S., Mellia, M., and Munafo, M. Exploring the cloud from passive measurements: the amazon aws case. In *Proceedings of IEEE INFOCOM'13*, pp. 230–234.

[9] Birke, R., Podzimek, A., Chen, L. Y., and Smirni, E. State-of-the-practice in data center virtualization: Toward a better understanding of vm usage. In *Proceedings of IEEE/IFIP DSN'13*.

[10] Durumeric, Z., Wustrow, E., and Halderman, J. A. Zmap: Fast internet-wide scanning and its security applications. In *Proceedings of USENIX Security'13*, pp. 605–620.

[11] He, K., Fisher, A., Wang, L., Gember, A., Akella, A., and Ristenpart, T. Next stop, the cloud: understanding modern web service deployment in ec2 and azure. In *Proceedings of ACM IMC'13*, pp. 177–190.

[12] Kim, T., Peinado, M., and Mainar-Ruiz, G. System-level protection against cache-based side channel attacks in the cloud. In *Proceedings of USENIX Security'12*.

[13] LaCurts, K., Deng, S., Goyal, A., and Balakrishnan, H. Choreo: network-aware task placement for cloud applications. In *Proceedings of ACM IMC'13*.

[14] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of ACM CCS'09*, pp. 199–212.

[15] Varadarajan, V., Ristenpart, T., and Swift, M. Scheduler-based defenses against cross-vm side-channels. In *Proceedings of USENIX Security'14*.

[16] Varadarajan, V., Zhang, Y., Ristenpart, T., and Swift, M. A placement vulnerability study in multi-tenant public clouds. In *Proceedings of USENIX Security'15*.

[17] Wang, G., and Ng, T. E. The impact of virtualization on network performance of amazon ec2 data center. In *Proceedings of IEEE INFOCOM'10*.

[18] Wang, L., Nappa, A., Caballero, J., Ristenpart, T., and Akella, A. Whowas: A platform for measuring web deployments on iaas clouds. In *Proceedings of ACM IMC'14*.

[19] Wu, Z., Xu, Z., and Wang, H. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *Proceedings of USENIX Security'12*, pp. 159–173.

[20] Xu, Y., Bailey, M., Jahanian, F., Joshi, K., Hiltunen, M., and Schlichting, R. An exploration of L2 cache covert channels in virtualized environments. In *Proceedings of ACM CCSW'11*, pp. 29–40.

[21] Xu, Y., Musgrave, Z., Noble, B., and Bailey, M. Bobtail: avoiding long tails in the cloud. In *Proceedings of USENIX NSDI'13*, pp. 329–342.

[22] Xu, Z., Wang, H., and Wu, Z. Technical Report: WM-CS-2015-03. `http://www.wm.edu/as/computerscience/documents/cstechreports/WM-CS-2015-03.pdf`.

[23] Xu, Z., Wang, H., Xu, Z., and Wang, X. Power attack: An increasing threat to data centers. In *Proceedings of NDSS'14*.

[24] Zhang, Y., Juels, A., Oprea, A., and Reiter, M. K. Homealone: Co-residency detection in the cloud via side-channel analysis. In *Proceedings of IEEE S&P'11*, pp. 313–328.

[25] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of ACM CCS'14*, pp. 990–1003.

[26] Zhang, Y., and Reiter, M. K. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of ACM CCS'13*, pp. 827–838.

A Business scale of EC2

Figure 19 illustrates the number of all the detected live instances in EC2 US east region during the measurement period. We can see that the business scale in EC2 US east region is very impressive. Our scanning can always detect more than 650,000 live instances in the cloud. During the peak time, we can detect almost 700,000 live instances. It is noteworthy that our system only scans some common ports. Besides the instances we detected, there are some instances with no common ports opened or within the VPC that do not have public IP addresses. Thus, the real number of live instances in the cloud could be even larger.

Table 4 lists the break-down statistics, showing the number of instances hosting a certain service on average. It is obvious that web service still dominates the usage in IaaS. Most customers rent the instances to host their web services. Among these web services (i.e., HTTP), more than half of them deploy HTTPS at the same time. Since the default way of accessing an instance in EC2 is through SSH, the number of instances listening on port 22 is the second largest. There are also considerable instances hosting FTP service, DNS service, and database service (MYSQL+SQL). For the rest of services, the number of instances hosting them are less significant.

Table 4: Number of instances hosting a certain service

	FTP	SSH	Telnet	SMTP	WHOIS	DNS	DHCP	Finger	HTTP	SQL	HTTPS	MYSQL
Live instances	24,962	327,294	350	18,376	305	3,392	15	68	441,499	48	261,446	25,872

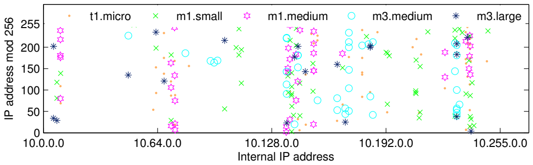


Figure 22: The distribution of internal IP addresses of instances with different types in availability zone us-east-1a.

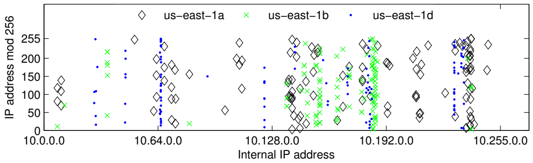


Figure 23: The distribution of internal IP addresses of instances in different availability zones.

B Dynamic environment of EC2

Our measurement can also reflect the dynamic environment of EC2 to some extent. First, as shown in Figure 19, the number of live instances varies over time within a day. We observed a similar pattern each day: the peak time is around 5 p.m. (EST) while the service reaches a valley around 4 a.m. (EST). Despite this diurnal pattern, the difference in the number of live instances between peak and valley is not as significant as we expected. There are only 1,000 more live instances at peak than valley, which is relatively small considering the overall 650,000 live instances. The diurnal pattern is reasonable, as 4 a.m. EST is very early morning for the US east coast and it is also midnight for the US west coast. It is intuitive that at this time period fewer users are using EC2. The small difference between peak and valley can be explained from two aspects. First, most instances run stable services such as web and database services. These instances remain active all the time. Second, although the data center is located in the US, the customers are distributed all around the world. For instance, Bermudez et al. [8] demonstrated that the Virginia data center is responsible for more than 85% of EC2 traffic in Italy. The time of 4 a.m. on the US east coast is 10 a.m. in Italy when customers are very active there.

We are also interested in how dynamic the cloud environment is. Figure 20 illustrates how many instances are shutdown, newly booted, or re-located between each round of measurement. We can see there are more than 15,000 hosts that are changed every 20 minutes, indicating that EC2 is a very dynamic environment with tens of VMs booted and shut down every second.

Besides the dynamics of live instances, we are also interested in the networking dynamics. During our measurement, we observed overall 975,032 distinct private IP addresses and 1,024,589 distinct public IP addresses. We recorded all the mappings from public IP to private IP and the mappings from private IP to public IP during our measurement. We also recorded the mappings that are changed during the measurement period. Over the course of our 15-day measurement, 103,242 mappings changed. This implies that EC2 has likely recruited dynamic NAT for address translation.

Figure 21 shows the private IP addresses that are included in the changed mappings. It is clear that the IP address pool in the cloud is dynamic as well. The density of the IPs in a certain range is significantly higher than other areas. This range of private IPs are mostly assigned to micro and small instances. Since micro and small instances are usually used for temporary purposes, ON/OFF operations on them are more frequent, leading to more frequent changes in private-public IP mappings.

C VM placement locality in EC2

To investigate the VM placement locality in EC2, we launched numerous instances with different types and in different availability zones to study whether the type or zone will impact the physical location of an instance.

Figure 22 illustrates the private IP distribution of some sample instances with different types in zone us-east-1a. The IP distribution exhibits a certain type locality. We can see from the figure that the instances of the same type tend to have closer internal IPs, i.e., they are more likely to be placed physically close to one another. However, compared with corresponding results in 2008 [14], we can see that such type locality has been significantly weakened.

We also study how availability zone could affect VM placement. Figure 23 illustrates the internal IP distribution of instances in different availability zones. As we can see, VM placement still has availability zone locality, i.e., instances in the same zone are more likely to have their internal IP addresses located within a certain range. However, such locality is also much weaker than in 2008 [14].

Towards Discovering and Understanding Task Hijacking in Android

Chuangang Ren[1], Yulong Zhang[2], Hui Xue[2], Tao Wei[2] and Peng Liu[1]

[1]Pennsylvania State University, State College
[2]Fireeye, Inc.

Abstract

Android multitasking provides rich features to enhance user experience and offers great flexibility for app developers to promote app personalization. However, the security implication of Android multitasking remains under-investigated. With a systematic study of the complex tasks dynamics, we find design flaws of Android multitasking which make all recent versions of Android vulnerable to *task hijacking* attacks. We demonstrate proof-of-concept examples utilizing the task hijacking attack surface to implement UI spoofing, denial-of-service and user monitoring attacks. Attackers may steal login credentials, implement ransomware and spy on user's activities. We have collected and analyzed over 6.8 million apps from various Android markets. Our analysis shows that the task hijacking risk is prevalent. Since many apps depend on the current multitasking design, defeating task hijacking is not easy. We have notified the Android team about these issues and we discuss possible mitigation techniques in this paper.

1 Introduction

In the PC world, computer multitasking means multiple processes are running at the same period of time. In Android systems, however, multitasking is a unique and very different concept, as defined in Android documentation: "A *task* is a collection of activities that users interact with when performing a certain job" [1]. In other words, a task contains activities [4] (UI components) that may belong to multiple apps, and each app can run in one or multiple processes. The unique design of Android multitasking helps users to organize the user sessions through tasks and provides rich features such as the handy application switching, background app state maintenance, smooth task history navigation using the "back" button, etc. By further exposing task control to app developers, Android tasks have substantially enhanced user experience of the system and promoted personalized features for app design.

Despite the merits, we find that the Android task management mechanism is plagued by severe security risks. When abused, these convenient multitasking features can backfire and trigger a wide spectrum of *task hijacking attacks*. For instance, whenever the user launches an app, the attacker can condition the system to display to the user a spoofed UI under attacker's control instead of the real UI from the original app, without user's awareness. All apps on the user's device are vulnerable, including the privileged system apps. In another attack, the malware can be crafted as one type of ransomware, which can effectively "lock" the tasks that any apps belong to on the device (including system apps or packages like "Settings" or "Package Installer"), i.e. restricting user access to the app UIs and thus disabling the functionality of the target apps; and there is no easy way for a normal user to remove the ransomware from the system. Moreover, Android multitasking features can also be abused to create a number of other attacks, such as phishing and spyware. These attacks can lead to real harms, such as sensitive information stolen, denial-of-service of the device, and user privacy infringement, etc.

The Android multitasking mechanism and the underlying feature provider, the Activity Manager Service (AMS), haven't been thoroughly studied before. In this paper, we take the first step to systematically investigate the security implications behind Android multitasking design and the AMS. At the heart of the problem, although the Android security model renders different apps sandboxed and isolated from one another, Android allows the UI components (i.e., activities) from different apps to co-reside in the same task. Given the complexity of task dynamics, as well as the vagaries of additional task controls available to developers, the attacker can play tricky maneuvers to let malware reside side by side with the victim apps in the same task and hijack the user sessions of the victim apps. We call this *task hijack-*

Attacks Types	Consequences	Vulnerable system & apps
Spoofing	Sensitive info stolen	all; all
Denial-of-service	Restriction of use access to apps on device	all; all
Monitoring	User privacy infringement	Android 5.0.x; all

Table 1: Types of task hijacking attacks presented in this paper (system versions considered - Android 3.x, 4.x, 5.0.x).

ing.

Given the security threats, it becomes important to fully study Android multitasking behaviors in a systematic way. We approach this topic by projecting the task behaviors into a state transition model and systematically study the security hazards originated from the discrepancies between the design assumptions and implementations of Android tasks. We find that there is a plethora of opportunities of task hijacking exploitable to create a wide spectrum of attacks. To showcase a subset of the attack scenarios and their consequences, we implement and present a set of proof-of-concept attacks as shown in Table 1.

We do vulnerability assessment to the task hijacking threats and discover that all recent Android versions, including Android 5, can be affected by these threats, and all apps (including all privileged system apps) are vulnerable to most of our proof-of-concept attacks on a vulnerable system. By investigating the employment of task control features by app developers based on 6.8 million apps in various Android markets, we find that despite the serious security risks, the “security-sensitive” task control features are popular with developers and users. We have reported our findings to the Android security team, who responded to take a serious look into the issue. We summarize our contributions below:

- To the best of our knowledge, we are the first to systematically study the security implications of Android multitasking and the Activity Manager Service design in depth.
- We discover a wide open attack surface in Android multitasking design that poses severe threats to the security of Android system and applications.
- Base on our vulnerability analysis over 6.8 million apps, we find that this problem is prevalent and can lead to a variety of serious security consequences.
- We provide mitigation suggestions towards a more secure Android multitasking sub-system.

2 Background

Android Application Sandbox: The Android security model treats third-party apps as untrusted and isolates them from one another. The underlying Linux kernel enforces the Linux-user based protection and process isolation, building a sandbox for each app. By default, the components of one app run in the same Linux process with an unique UID. Components from different apps run in separate processes. One exception is that different apps can run in one process only if they are from the same developer (same public key certificate), and the developer explicitly specifies the same process in the manifest file. The Linux sandbox provides the foundation for app security in Android. In addition, Android provides a permission model [12, 19] to extend app privileges based on user agreement, and offers an inter-component communication scheme guarded by permissions for inter-app communication.

Activity: *Activity* is a type of app component. An activity instance provides a graphic UI on screen. An app typically has more than one activities for different user interactions such as dialing phone numbers and reading a contact list. All activities must be defined in an app’s *manifest file*.

Intent: To cross the process boundaries and enable communication between app components, Android provides an inter-component communication (ICC) scheme supported by an efficient underlying IPC mechanism called *binder*. To perform ICC with other components, an component use *intent*, an abstract description of the operations to be performed. An intent object is the message carrier object used to request an action from another component, e.g., starting an activity instance by calling `startActivity()` function. Intent comes in two flavors. *Explicit intent* specifies the component to start explicitly by name. *Implicit intent* instead encapsulates a general type of action, category or data for a component to take. The system will launch a component “capable” of handling this intent. If more than one target activities exist in the system, the user is prompted to choose a preferred one.

Activity Manager Service (AMS): AMS is an Android system service that supervises all the activity instances running in the system and controls their life cycles (creation, pause, resume, and destroy). The interaction and communication protocols between activities and the AMS are implemented by the Android framework code, which is transparent to app developers, leaving developers focusing on the app functionality. While Window Manager Service (WMS) manages all windows in the system and dispatches user inputs from the windows, AMS organizes all the activities in the system into tasks, and is responsible for managing the tasks and supporting the multitasking features as will be described in Section 3.

In addition, AMS is in charge of supervising service components, intent routing, broadcasting, content providers accesses, app process management, etc., making itself one of the most critical system services in the Android system.

3 Android Tasks State Transition Model

3.1 Task and Back Stack

In Android, a *task* [1] is a collection of activities that users have visited in a particular job. The activities in a task are kept in a stack, namely *back stack*, ordered by the time the activities are visited, such that clicking the "back" button would navigate the user back to the most recent activity in the current task. The activities in the back stack may be from the same or different apps.

The activity displayed on the screen is a *foreground activity* (on the top of the back stack) and the task associated with it is a *foreground task*. Therefore, there is only one foreground task at a time and all other tasks are *background tasks*. When switched to the background, all activities in a task stop, and remain intact in the back stack of the task, such that when the users return they can pick up from where they left off. This is the fundamental feature that Android multitasking offers to users.

3.2 A Tasks State Transition Model

The status of tasks in a system keeps changing as a result of user interaction or app program behaviors. To understand the complex task dynamics and its behind security implications, we view the task transitions through time as a state transition model. The model is described by $(S, E, \Lambda, \rightarrow)$, where S denotes a set of task states; E and Λ are sets of events and conditions respectively; and $\rightarrow$ indicates a set of feasible transactions allowed by the system under proper events and conditions.

1. **Task state** (**$s \in S$**): represents the state of all tasks (specifically, the back stacks) in the system and their foreground/background statuses. In other words, the tasks in the system remain in one state iff the activity entries and their orders in the back stacks stay the same, and the foreground task remains to be the same task.
2. **Event** (**$e \in E$**): denotes the event(s) it takes to trigger the state transition, for example, pressing the "back" button or calling `startActivity()` function.
3. **Condition** (**$\lambda \in \Lambda$**): the prerequisites or configurations (usually default) that enable a state transition under certain events. We denote $\lambda^{default}$ as the system default conditions in this paper.

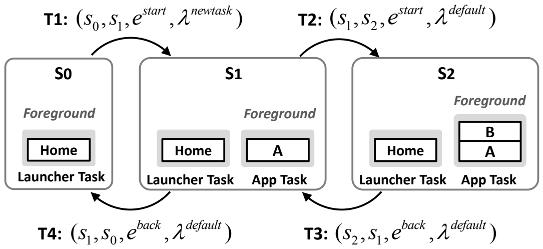

T1: $(s_0, s_1, e^{start}, \lambda^{newtask})$
T2: $(s_1, s_2, e^{start}, \lambda^{default})$
S0
Foreground
Home
Launcher Task
S1
Foreground
Home
A
Launcher Task
App Task
S2
Foreground
B
A
Home
Launcher Task
App Task
T4: $(s_1, s_0, e^{back}, \lambda^{default})$
T3: $(s_2, s_1, e^{back}, \lambda^{default})$

Figure 1: A simple task state transition example.

4. **Transition** (**$\rightarrow$**): stands for a feasible state transition. Not all task transitions are feasible, e.g., the order of activities in back stack cannot be changed arbitrarily (only push and pop are viable operations over the stack). A viable transaction is also represented as $s_1 \rightarrow s_2$, or (s_1, s_2, e, λ), where $s_1, s_2 \in S$.

3.3 A Task State Transition Example

Given the state transition model, we depict a simple task state transition example in Figure 1. The figure shows three task states, and the state transitions reflect the process in which the user first launches an app from the home screen ($s_0 \rightarrow s_1$), visits an additional activity UI in the app ($s_1 \rightarrow s_2$) and returns to the home screen by pressing the "back" button twice ($s_2 \rightarrow s_1 \rightarrow s_0$).

In each task state, we show all existing tasks and their back stacks. For example, s_0 is a task state in which no task, except the launcher task, is running in the system. The launcher task has only one activity in its back stack - the home screen from which users can launch other apps.

In $(s_0, s_1, e^{start}, \lambda^{newtask})$, a new app task is created and brought to the foreground in the resulting state s_1. e^{start} represents the event that `startActivity()` is called by the home activity in the launcher task. This event could happen when the user clicks the app's icon on the home screen. $\lambda^{newtask}$ specifies a special condition, i.e., the FLAG_ACTIVITY_NEW_TASK flag is set to the input intent object to `startActivity()` function. This flag notifies the AMS the intention of creating a new task to host the new activity. Note that in this example most state transitions are under default conditions, indicated by $\lambda^{default}$, while here $s_0 \rightarrow s_1$ is an exception because the launcher app customizes the condition ($\lambda^{newtask}$) for a valid design purpose: start the app in a brand new task when the user launches an new app. This is an example where app developers can customize certain configurable conditions to implement helpful app features. However, condition like $\lambda^{newtask}$ can be abused in a task hijacking attack, as discussed in Section 4.

Next, $(s_1, s_2, e^{start}, \lambda^{default})$ is triggered by event e^{start} again (this time called by activity A instead), yet under the default condition. By default, AMS pushes the new activity instance B on top of the current back stack

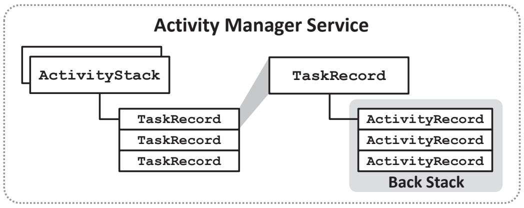


Figure 2: Data structures of tasks, activities and back stacks in the Activity Manager Service.

as shown in s_2. The previous activity A is stopped and its state is retained. In $(s_2, s_1, e^{back}, \lambda^{default})$, e^{back} represents the event of user pressing the "back" button. As expected by the user, the next activity A on stack is brought back to the screen, and its original state is resumed. Activity B is popped from the back stack and destroyed by the system. The initial state s_0 is finally restored through $(s_1, s_0, e^{back}, \lambda^{default})$ when the user presses "back" button again. The app's task is destroyed because when the popped activity is the last activity in the back stack, the activity is destroyed together with the "empty" task.

Note that activities from different apps can co-reside in the same task (e.g. activity A and B in this example). In other words, although activities from different apps are isolated and protected within their own process sandboxes, Android allows different apps to co-exist in a common task. This creates opportunities for malicious activities to interfere with other activities once they are placed in the same task, and the system passes the program control to the malicious activities.

In reality, the amount of possible task states in a system is big, and the state transitions can be complex, e.g., each state may again have numerous incoming and outgoing transitions connecting with other states. In Section 4, we discuss what may go wrong during the complex task state transitions.

3.4 Android Implementation

AMS maintains Android tasks and activities in a hierarchy shown in Figure 2. AMS uses `TaskRecord` and `ActivityRecord` objects to represent tasks and activities in the system respectively. A `TaskRecord` maintains a stack of `ActivityRecord` instances, which is the back stack of that task. Similar to the activities in a back stack, tasks are organized in a stack as well, maintained by a `ActivityStack` object, such that when a task is destroyed, the next task on stack is resumed and brought to the foreground. There are usually two `ActivityStack` containers in the system - one containing only the launcher's tasks and the other holding all remaining app tasks.

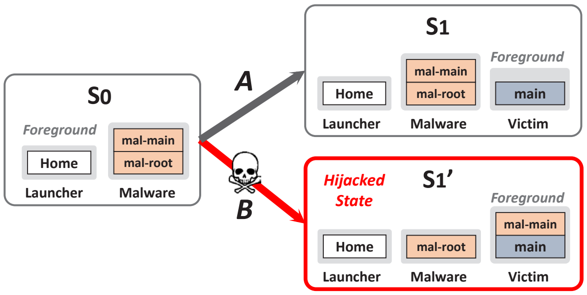


Figure 3: Task state transition of spoofing attack (A: task state transition by system-default. B: Hijacking state transition).

4 Task Hijacking in Android

In this section, we first discuss an example showing how an attacker could manipulate the task state transitions to his advantage, causing task hijacking attacks. We then explore the extent of different task hijacking methods and how they can be used for other various attack goals.

4.1 Motivating Example

Suppose attacker's goal is to launch an UI spoofing attack. Specifically, when the user launches a victim app from the home screen, a spoofing activity with an UI masquerading the victim app's main activity (e.g. the login screen of a bank app) shows up instead of the original activity.

Figure 3 shows the task state transitions of the UI spoofing attack. Initially in s_0, the home screen is displayed to the user while a malware task waits in the background. Like the task state transition example just shown in Section 3.3, when the user launches the victim app from the launcher, state transition A is supposed to occur by default, i.e. a new task is created and the app's main activity is displayed on screen. However, as shown in state transition B, the malware can manipulate the task state transition conditions such that the system instead displays the spoofing UI of activity "mal-main" by relocating "mal-main" from the background task to the top of victim app's back stack. The user has no way to detect the spoofing UI since the original activity UI is not shown on screen at all, and the "mal-main" activity appears to be part of the victim app's task (perceivable in recent task list). By this means, the victim task is smoothly hijacked by the malware activity from launch time, and all user behaviors within this task are now under malware's control.

In this example, the attacker successfully misleads the system and launches the spoofing UI by abusing some task state transition conditions, i.e. `taskAffinity` and `allowTaskReparenting`. We will introduce them together with other exploitable conditions/events in

Conditions	
Intent Flags (FLAG_ACTIVITY_*)	*Activity Attribute*
NEW_TASK SINGLE_TOP CLEAR_TOP REORDER_TO_FRONT NO_HISTORY CLEAR_TASK NEW_DOCUMENT (API 21) MULTIPLE_TASK	launchMode allowTaskReparenting taskAffinity allowTaskReparenting documentLaunchMode (API 21) finishOnTaskLaunch

Events	
Callback Function	*Framework APIs*

onBackPressed()	startActivity()
	startActivities()
	TaskStackBuilder class

Table 2: Task control knobs - configurable task state transition conditions and events provided by Android.

Section 4.5 and 4.6.

4.2 Adversary Model

We assume the user's Android device already has a malware installed (similar assumptions are made in [8, 25, 34, 38]). The malware pretends to seem harmless, requiring only a minimum set of widely-requested permissions such as INTERNET permission. The attacker's goal is clear: blend the malicious activities with the target app's activities in one task, and intercept the normal user operations to achieve malicious purposes.

4.3 Hijacking State Transition

A *hijacked task state* is a desirable state to attackers, in which at least one task in the system contains both malicious activities (from malware) and benign activities (from the victim app). The task state $s_1^{'}$ in the spoofing attack is an example of hijacked task state. A *hijacking state transition* (HST) is a state transition which turns the tasks in the system to a dangerous hijacked task state, e.g., the task state transition B in the previous example. Conceptually, there are two types of HSTs:

1. The malicious activity gets pushed onto the victim task's back stack (malware⇒victim);
2. The victim app activity is "tricked" by malware and pushed on the malware's back stack (victim⇒malware).

4.4 The Causes of HSTs

Android provides a rich set of task control features, i.e., task state transition conditions and events. We call these features as *task control knobs*. The task control knobs provide app developers with broad flexibility in controlling the launch of new activities, the relocation of existing activity to another task, "back" button behaviors, even the visibility of a task in the recent task list (a.k.a overview screen), etc. Table 2 lists such conditions and events in four categories: activity attribute, intent flags, call-back functions, and framework APIs. All these control flexibility further complicates task state transitions.

Due to HST's potential threats to app and system security, understanding the extent of HSTs in the complex task state transitions becomes important. To achieve this, we simulate the task state transitions in a Android system and try to capture all possible HSTs and hijacked task states that occur during the state transitions.

In theory, there are a huge number of possible task states (each app may have a number of activities, and an activity can be instantiated for multiple times). We confine the number of task states to more interesting cases by adding two constraints: (1) each app only has two activities - the main activity and another public exported activity (can be invoked by other apps), and (2) each activity can only be instantiated once. In the simulation, we specify three apps in the system - namely, Alice, Bob and Mallory (the malware) - as it covers most HST cases.

Given the task states, we generate the task state transition graph by connecting pairs of states with directed edges. For instance, state s_1 and s_2 are connected only if $\exists e \in E, \lambda \in \Lambda$, such that (s_1, s_2, e, λ) or (s_2, s_1, e, λ) are valid transitions, where E denotes all feasible events and Λ represents all possible conditions in Table 2. After constructing the task state transition graph, all hijacked states and HSTs are highlighted. We show a sub-graph of the resulting task state transition graph in Figure 4(a) and visualize the task states in Figure 4(b). For clarity of the presentation, we only show the interesting branches of the over-sized graph and have skipped many duplicated HST cases. Moreover, we zoom in each of the HSTs and show their detailed information in Table 3, including the conditions and events that trigger the HSTs. We manually verify all presented HSTs on real systems and these HSTs are proven to be exploitable to launch real attacks (indicated in the last column in Table 3).

We make two important observations from our result. First, once exploited, the hijacked states shown in Figure 4(a) could result in serious security hazards. For example, HST#3 is the task state transition of our example attack discussed earlier. As a result of this HST, the screen is under attacker's control in state s_{14}. As another example, in HST#2, the benign activity B2 is tricked to be placed in Mallory's task instead of Alice's task during start-up. This can also lead to spoofing attack or GUI confidentiality breaches.

Second, compared with the HST triggered by the system-default conditions and events (e.g., HST#1), more HST scenarios are produced under the configurable conditions and events (HST#2-6) . It means that, by abusing the flexible task control "knobs" readily offered

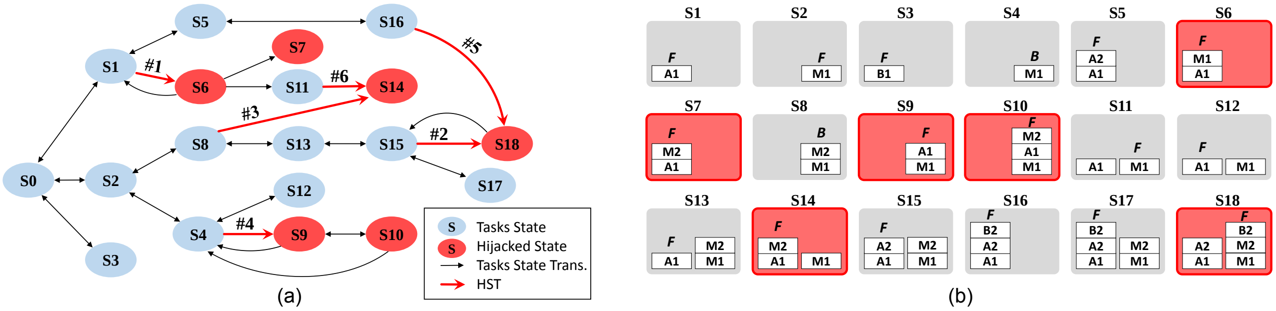


Figure 4: (a) A sub-graph of the over-sized task state transition graph for a simulated system with three apps. The sub-graph shows the typical cases of HSTs (red edges with HST indexes) and the resulting hijacked task states (red nodes). s_0 represents the initial state, i.e., no tasks except the launcher task exists in the system. (b) Visualization of task states of all nodes in figure (a). A, B and M represent the activities from Alice, Bob and Mallory (the malware) respectively. We skip showing the launcher task in the task states. Hijacked states are highlighted as red boxes. F and B denote foreground and background tasks respectively.

HST #	HST Type	Conditions	Events	Attacks in Section 5
1	malware⇒victim	Default	A1: startActivity(M1)	phishing I
2	victim⇒malware	M1:taskAffinity=B2 NEW_TASK intent flag set or B2:launchMode=”singleTask”	A2: startActivity(B2)	phishing II
3	malware⇒victim	M2:taskAffinity=A1; M2:allowTaskReparenting=”true” NEW TASK intent flag set	launcher: startActivity(A1)	spoofing

4	victim⇒malware	M1:taskAffinity=A1; NEW_TASK intent flag set	launcher: startActivity(A1)	denial-of-use;
				ransomware;
				spyware

5	victim⇒malware	M1:taskAffinity=B2; B2:allowTaskReparenting=”true”	startActivities([M1, M2]) or use TaskStackBuilder	phishing III
6	malware⇒victim	M2:taskAffinity=A1 NEW_TASK intent flag set or M2:launchMode=”singleTask”	M1: startActivity(M2)	-

Table 3: Detailed information of the HSTs (red edges with HST indexes in Figure 4). E.g., condition “M1:taskAffinity=B2” indicates that the taskAffinity attribute of activity M1 is set to that of B2; Event “launcher:startActivity(A1)” means that activity A1 is started by the launcher.

by the Android system, the attacker can actively create a plethora of HSTs that harm other apps. In Figure 4(a), we only show several typical HST cases, yet there are much more HST instances of these types in the complete state transition graph.

The HST cases and their conditions/events summarized in Table 3 may now look mysterious. We will demystify these conditions and events in the rest of this section.

4.5 Exploiting Conditions

In Table 3, HSTs #2, #4, #6 are similar with respect to their state transition conditions, i.e. all three HSTs occur by virtue of customized activity launch mode (by setting `launchMode` attribute or NEW_TASK intent flag). HSTs #3, #5 are similar as they both use `allowTaskReparenting` attribute to enable activity re-parenting.

4.5.1 Activity Attributes

One can define the attributes [2] of an activity in the `<activity>` element in manifest file. The attributes not explicitly defined are set to default values.

Task Affinity: Task affinity declares what task an activity prefers to join. It is a hard-coded string defined as `<android:taskAffinity="affinity">`, where `affinity` is the task affinity string that can be defined arbitrarily. By explicitly declaring a task affinity, an activity is able to actively “choose” a preferable task to join within its life cycle. If not explicitely specified in the manifest, the task affinity of an activity is the app package name, such that all activities in an app prefer to reside in the same task by default. The affinity of a task is determined by the task affinity of the task’s *root activity* (the activity on the bottom of back stack).

Task affinity is a crucial condition used in most of the HSTs in Table 3. There are two occasions in which an activity can “choose” its preferred host task: (1) when an activity attempts to be started as a new task (i.e., “`singleTask`” launch mode or NEW_TASK intent flag as in HST#2, #4, #6), and (2) if the `allowTaskReparenting` activity attribute is set to true, and another task with the same task affinity is brought to the foreground (as in HST#3, #5). We explain the above two cases in detail in the following paragraphs.

Launch Mode: Activity launch mode defines how an activity should be started by the system. Based on the launch mode, the system determines: (1) if a new activity instance needs to be created, and (2) if yes, what task should the new instance be associated with. The launch mode can be either statically declared by specifying `<android:launchMode="value">` in the manifest file or dynamically defined using intent flags discussed in Section 4.5.2.

By default, `launchMode="standard"`. In this mode, the AMS would create a new activity instance and put it on top of the back stack on which it is started. It's possible to create multiple instances of the same activity and those instances may or may not belong to the same task. With `launchMode="singleTask"`, the decision-making of activity start-up is more complex. An investigation into Android source code reveals three major steps the AMS takes towards starting an activity. First, if the activity instance already exists, Android resumes the existing instance instead of creating a new one. It means that there is at most one activity instance in the system under this mode. Second, if creating a new activity instance is necessary, the AMS selects a task to host the newly created instance by finding a "matching" one in all existing tasks. An activity "matches" a task if they have the same *task affinity*. After finding such a "matching" task, the AMS puts the new instance into the "matching" task. This explains why in HST #2 and #6, the newly-started and foreground activities (B2 and M2) are put on other "matching" tasks (with the same task affinity) instead of the tasks who start them. Third, without finding a "matching" task, the AMS creates a new task and makes the new activity instance the root activity of the newly created task.

Task Re-parenting: By default, once an activity starts and gets associated with a task, such association persists for the activity's entire life cycle. However, setting `allowTaskReparenting` to true breaks this restriction, allowing an existing activity (residing on an "alien" task) to be re-parented to a newly created "native" task, i.e., a task having the same task affinity as the activity.

For example, in HST#3 resembles the spoofing attack example discussed in Section 4.1. M2 is supposed to stay on Mallory's task at all time. However, M2 has its `allowTaskReparenting` set to true, and `taskAffinity` set to Alice's package name, such that when Alice's task is started (A1 as the root activity) by the launcher, M2 is re-parented to Alice's new task and the user sees M2 on screen instead of A1. In this process, A1 is never brought to the screen at all. Likewise, HST #5 occurs due to similar reason, except that this time the benign activity B2 (with its `allowTaskReparenting` set to true) is re-parented to the malware task.

The above activity attributes offer attackers with great flexibility. The attackers can put their malicious activities to a preferred hosting tasks under certain events, e.g., `singleTask` launch mode during an activity start-up and `allowTaskReparenting` during a new task creation. Furthermore, an activity is free to choose any app as their preferred task owner (including the privileged system apps) by specifying the target app's package name as their task affinity. These conditions lead to a bulk of HSTs in the simulation, and these HSTs can be employed to launch powerful task hijacking attacks as we will see in Section 5.

4.5.2 Intent Flags

Before sending an intent to start an activity, one could set intent flags to control how the activity should be started and maintained in the system by calling `intent.setFlags(flags)`. `intent` is the intent object to be sent, and `flags` is an `int` value (each bit indicates a configuration flag to the AMS).

Noticeably, the `FLAG_ACTIVITY_NEW_TASK` intent flag, if set, lets an activity be started as if its `launchMode="singleTask"`, i.e. the system goes through the same procedures as explained in launch mode to find a "matching" task or create a new task for the new activity instance. This is the dynamic way of setting activity's launch mode. Launcher app always uses this flag to start an app in a new task as in HST#4.

4.6 Exploiting Events

4.6.1 Callback Function

Android framework provides a variety of callback functions for activities to customize their behaviors under particular events, e.g., activity life cycle events (start, pause, resume or stop), key pressing events, system events, etc.

`onBackPressed()` is a callback function defined in `Activity` class, and is invoked upon user pressing the "back" button. The default implementation in framework code simply stops and destroys the current activity, and it then resumes the next activity on top of the current back stack, as we have seen in Section 3.3. However, an attacker can override this callback function for its malicious activity and arbitrarily define a new behavior upon "back" button pressing, or simply disable the "back" button by providing an empty function. As a result, once the malicious activity is brought to the foreground, pressing the "back" button triggers the code of attacker's control.

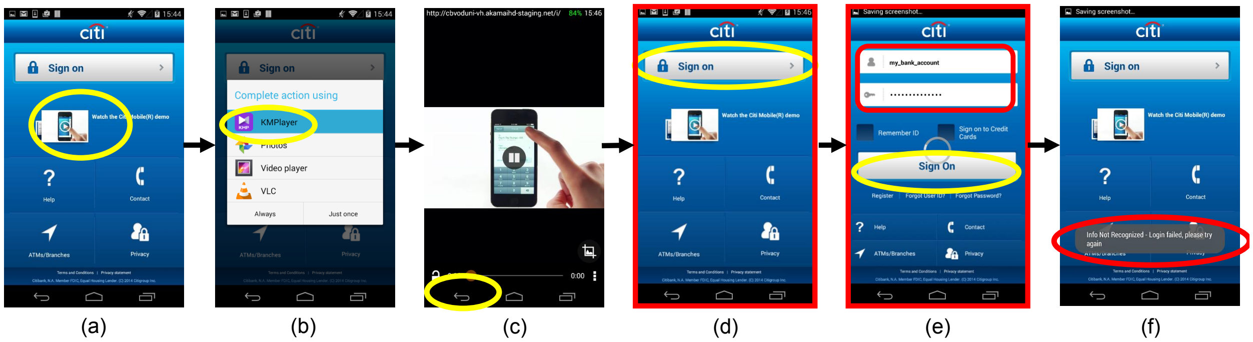


Figure 5: The process of "back hijacking" phishing attack to a well-known bank app. (a) shows the main activity of the bank app. A new user taps on the tutorial video link in the bank app; In (b), a system dialog prompts the user to choose a video player available in the system; In (c), the video player activity is started, and the user later clicks "back" button, intending to "goes back" to the original main activity; In (d) and (e), the back button directs the user to the phishing UIs, which spoof the user and steal bank account credentials. The phishing activity then quits after user clicks "Sign On"; In (f), the original main activity is resumed, with a log-in failure toast message displayed by the quitting malware.

4.6.2 Framework API

Android framework provides APIs to create new tasks with established back stacks. For example, `TaskStackBuilder` is a utility class that allows an app developer to construct a back stack with specified activities, and to start the back stack as a brand new task in the system at a later time (e.g. using a PendingIntent). Similarly, `startActivities()` in `Activity` class achieves the same thing except that it builds and starts the tasks in one API function call. These framework APIs are helpful for attackers to build and launch new tasks containing designated back stacks without explicitly displaying all activities in the back stacks on screen.

5 Task Hijacking Attack Examples

In this section, we demonstrate more attack examples utilizing exploitable HSTs in Table 3. These attacks can breach the integrity, availability and confidentiality of victim apps' UIs respectively. We have tested these attacks on Android 3.x, 4.x and 5.0.x.

5.1 Breaching UI Integrity

The UI integrity here means the "origin/source integrity" of the victim app's activities, instead of the "data integrity". That is, instead of modifying the original activities of the victim app, attackers deceive the user by spoofing UIs, which can prevent the original UIs from being displayed on screen.

5.1.1 Spoofing Attack

As we have already seen in Section 4.1 and 4.5, by manipulating `allowTaskReparenting` and `taskAffinity`, an attacker can successfully hijack a new task with a spoofing activity. This attack affects all apps on device including the most privileged system apps (e.g., Settings). The attacker can even target multiple apps on user device at the same time, as long as the background malware tasks (targeting different task affinity) are started in advance.

Stealthiness: In order to make the spoofing attack more stealthy, the attacker could take advantage of other task transition conditions and events to achieve this. For example, the attacker can make its background malware tasks absent from the recent task list by setting the activity attribute `excludeFromRecents` to true. As another example, the user may accidentally resume the app's original activity (the root activity of victim app's task) by clicking the "back" button from the on-screen spoofing activity. To prevent users from observing this abnormal app behavior, the attacker can override `onBackPressed()` of the spoofing activity, bringing the home screen back to the foreground, such that it gives the user an illusion that it is in coherence with the system's default "back" behavior.

5.1.2 Phishing Attack - "Back Hijacking"

The back button is popular with users because it allows users to navigate back through the history of activities. However, attackers may abuse the back button to mislead the user into a phishing activity.

We devise three phishing attack methods that target the same banking app, and demonstrate two of them in this paper. Figure 5 shows the screen shots of the phishing attack process. The phishing UIs show up when the user returns from a third-party app activity, and the user unwittingly believes that he/she has returned to the original bank activity.

Figure 6 shows the state transition diagrams of two attack methods. The two attack methods differ in that,

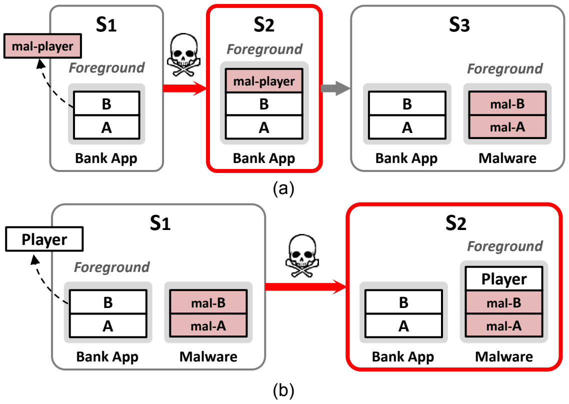


Figure 6: Tasks state transition diagrams of "back hijacking" attacks. Figure (a) and (b) shows method I and II respectively.

user chooses a malicious video player in the first attack, while in the second attack, even though the user chooses a benign player, the bank task can still be hijacked when the user launches the video player.

Method I: Figure 6(a) shows the state transition diagram of the first attack method. We skip the unrelated task(s) (e.g. launcher) in the system and only show tasks of interest. In s_1, the bank app task contains activities A and B, in which B is the login activity. The HST occurs in $s_1 \rightarrow s_2$, triggered by the event that the user clicks the tutorial video from the login UI, sending out a implicit intent to look for an exported activity in the system capable of playing the tutorial video. Unfortunately, the user selects the malicious video player activity "mal-player" from the system pop-up and this results in the hijacked state s_2. After user finishes watching the video, $s_2 \rightarrow s_3$ is triggered by user pressing the "back" button. However, the "back"-pressing event is modified by overriding `onBackPressed()` in the "mal-player" activity. As a result, instead of resuming activity B, a new malicious task is created (by using `TaskStackBuilder`) and brought to the front. As can be seen, the HST takes place under default conditions as in HST#1 (in Table 3).

The user session is hence hijacked to the malware task, which contains "mal-A" and the foreground "mal-B" phishing activities. Note that in this attack, the malware need to camouflage as a useful app (e.g. a video player in this case) that users are likely to use.

Method II: As shown in Figure 6(b), the same phishing attack can succeed even when the user selects a benign video player. In s_1, a malware task with two phishing activities lurks in the background. Similarly, HST occurs in $s_1 \rightarrow s_2$, when the user launches a benign video player. However, as shown in the resulting state s_2, instead of joining the banking task, the new video player activity is pushed in the malware task's back stack, such that pressing the "back" button after the video play resumes the phishing activity "mal-B".

This HST is similar to HST#2 (in Table 3) in that the benign video player attempts to be started as a new task, either because of the `NEW_TASK` flag set in the intent by the bank activity, or the "singleTask" launch mode set by the video player. Furthermore, the existing malware task has its `taskAffinity` maliciously set to the benign video player.

Stealthiness: We employ similar methods in the previous spoofing attack to ensure the stealthiness of the background malware tasks in both phishing attack methods. Moreover, we disable the animation of task switching, producing an illusion to the user that the screen transition is within the same task/app.

5.2 Breaching UI Availability

Task hijacking can also be leveraged to restrict the availability of an app's UI components, or in other words, to prohibit user access to part or all functionality of an victim app.

5.2.1 Preventing Apps from Being Uninstalled

In this example, the attacker is able to completely prevent apps from being uninstalled.

Ways to Uninstall An App: There are generally three ways for a user to uninstall an app from the device: (1) uninstall from the system Settings app; (2) dragging the app icon to the "trash bin" on home screen; or (3) uninstall with the help of a third-party app, e.g. an anti-virus app. In these scenarios, the Settings, Launcher, and the third-party apps will respectively generate an request to uninstall the app. Such a request eventually reaches the system package installer, which has the exclusive privilege to install/uninstall apps. Upon receiving the request, package installer pops up a dialog for the user to confirm. The dialog itself is an activity (namely uninstaller activity) from the system package installer and is pushed in the back stack of whoever is making the request (e.g. s_4 in Figure 7). No app can be uninstalled without user confirmation on the uninstaller activity.

Attack Method: The attacker can prevent app uninstallation by restricting user access to the uninstaller activity when it shows up on screen. In this attack, once the uninstaller is found to be in the foreground, a malicious activity is immediately pushed on top of the uninstaller activity in the same back stack, such that the uninstaller is "blocked" and becomes inaccessible to the user.

Figure 7 shows the state transition diagram of this attack targeting Settings app. Similar methods can be easily adopted to block app un-installation from the launcher or the anti-virus apps (e.g. when malware is detected).

In s_1, a task with only one root activity ("mal-root") from the malware is waiting in the background,

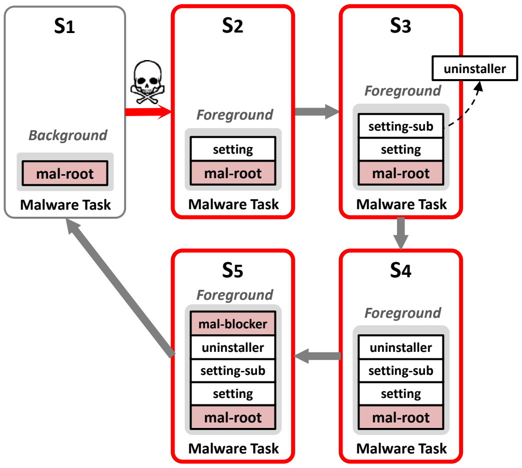


Figure 7: Tasks state transition diagram of application uninstall prevention attack.

with its `taskAffinity` set to the Settings app (com.android.settings). The HST occurs in $s_1 \to s_2$, triggered when the user opens up the Settings from the home screen (we skip Launcher task in the figure). In s_2, instead of hosting the newly-created "setting" activity in a new task, "setting" activity is pushed on top of the malware's back stack because the it is started by the launcher with a `NEW_TASK` flag. As a result, upon start-up, the privileged Settings app is unwittingly "sitting" on a task owned by the malware. This is similar to HST#4 in Table 3.

The user then goes through a few more sub-setting menus to find the app (as shown in s_3) and clicks the uninstall button, after which the uninstaller activity shows up for user confirmation (as shown in s_4). Once this happens, a malicious activity namely "mal-blocker" is immediately (even without user awareness of the uninstaller dialog) launched by a malicious background service, which keeps monitoring the foreground activity. The "mal-blocker" activity, started by a `NEW_TASK` flagged intent and with the same task affinity as the Settings app, is thus pushed in the same task, and effectively blocks the uninstaller as shown in s_5. The "mal-blocker" activity has its "back" button disabled, such that the user has no way to access the uninstaller activity right below it in the back stack whatsoever, and thus cannot confirm the app uninstalling operation.

In fact, the "back" button of "mal-blocker" is not only disabled, but is also augmented with a new event that triggers $s_5 \to s_6$: invoking (call `startActivity()`) the "mal-root" activity with an intent having `CLEAR_TOP` flag set, which results in the killing of the uninstaller and Settings activities in the task.

Preventing Un-installation from adb: An advanced user may resort to *Android Debug Bridge* (adb), a client-server program used to connect Android devices from a computer, and uninstall the malware from adb. However, in order to use adb, the user needs to first enable USB debugging in the Settings. The malware can block it in the Settings using similar technique and prevent the use of adb, as long as the USB debugging is not enabled before the attack (which is the case for most normal users).

5.2.2 Ransomware

Ransomware blackmails people for money in exchange of their data, and it has recently hit Android in a large scale [5]. The attackers may use UI hijacking to implement ransomware.

The malicious background service mentioned above takes the following two responsibilities and is difficult to be completely stopped. (1) Assure the malicious root activity ("mal-root") is alive: it re-creates a new root activity once the activity is found to be destroyed; and (2) monitor the foreground activity: if the target activity shows up, it immediately starts "mal-blocker" to block user access to the target activity, as we have seen in $s_4 \to s_5$. To prevent itself from being killed, the service registers itself in the system alarm service, who fires a pending intent in every given fixed time interval, relaunching the service if it is found to be killed.

By this mean, the ransomware is able to restrict user access to any target apps of attacker's choice, and can potentially render the Android device completely useless.

5.3 Breaching UI Confidentiality

The attack method in Section 5.2 can also be deployed to devise a new spyware, namely "TaskSpy" capable of monitoring the activities within any tasks in the newest Android 5.0.x systems (API 21), without requiring any permissions.

In Android, the system regards the owner of the root activity in a back stack to be the owner of the corresponding task. Android 5.0 allows an app to get the information of the caller app's own tasks (including the activities in the tasks) without requiring any permission. It means that, if a spyware can "own" the tasks of all the apps it intends to spy on, it is able to get the information of these tasks that in fact contain the victim apps' activities. Task hijacking is especially useful to "TaskSpy" in this case. In other words, "TaskSpy" can use the HST presented in Section 5.2 to "own" the tasks of any victim apps and thus stealthily spy on their activities without using any permission. Chen et. al. have achieved the same goal in their work [8] by monitoring and interpreting the shared VM information via public side channels. Compared with their attack, task hijacking can do this in a more direct and reliable way on Android 5.0.x.

Vul. app	Atk #	Vul. conditions	% of vul.	Tol. % of vul.
V	I	Send implicit intent for exported activities	93.9	93.9
	II	Send implicit intent for exported activities and use intent flag NEW_TASK	65.5	
S	II	Contains public exported activity and lauchMode="singleTask"	14.2	14.4
	III	Contains public exported activity and allowTaskReparenting="true"	1.4	

Table 4: Percentage of vulnerable victim apps (V) and "service" apps (S) to the "back hijacking" phishing attacks respectively, among 10,985 most popular Google Play apps.

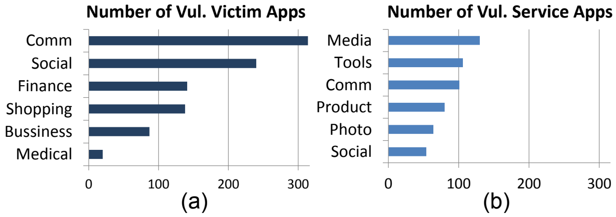


Figure 8: (a) Breakdown of vulnerable victim apps in security-sensitive app categories. (b) Breakdown of vulnerable "service" apps in the most widely useful app categories.

6 Evaluation

We first seek to understand the extent of vulnerable systems and apps to the attacks we have presented in Section 5. By doing large-scale app analysis across various markets, we then provide the current use status of the task control knobs in real implementations. Base on our insights from the result, we provide mitigation suggestions to defend against task hijacking threats in Section 7.

6.1 Vulnerability Analysis

Vulnerable Android Versions: We say an Android version is vulnerable to a particular attack if a malware can successfully launch the attack to a victim app on the system. Since the unique multitasking is part of Android design and most features have been introduced early in Android's evolution, we find that recent Android versions, including 3.x, 4.x and 5.0.x, are vulnerable to all our presented attacks, except the "TaskSpy" attack. As discussed in Section 5.3, "TaskSpy" relies on specific APIs introduced from API 21, and therefore, only affects the newest Android 5.0.x systems.

Apps Vulnerable to Task Hijacking Attacks: As summerized in Table 1, all the apps installed on a vulnerable Android system (including the privileged system apps) are vulnerable to all the attacks presented in this paper, except the "Back Hijacking" phishing attacks, which require certain prerequisites for an app to be vulnerable. Despite the prerequisites, the "Back Hijacking" phishing attacks are extremely stealthy, can be easily crafted and can cause serious consequnces. We try to further understand the scale of apps vulnerable to the "Back Hijacking" phishing attack by analyzing the most popular apps in Google Play.

Apps Vulnerable to "Back Hijacking": In a phishing attack, the attacker would be likely to target the most popular and valuable apps. Therefore, we focus our vulnerability analysis on the most popular 10,985 apps from Google Play, i.e., apps with over 1 million installs.

We indicate a vulnerable app in the phishing attacks to be of either one or both of the following two types: (1) victim app - the target victim app of the phishing attack (e.g. the bank app); and (2) "service" app - the benign app that provides publicly exported activities and is exploitable by the attacker to conduct user phishing on the victim apps (e.g. the benign video player). We do static analysis on the apps. Specifically, we perform inter-procedural analysis to identify all implicit intents (without permissions guarded) and the associated flags, and conduct manifest scan to find all activity attributes and public exported activities (excluding the main activities which are always exported). Table 4 lists the vulnerability conditions, and shows the percentages of both vulnerable victim apps and "service"apps to each and all the attack methods respectively.

As can be seen, 93.9% of the most popular apps in Google Play are vulnerable. This is partially because most apps would send out implicit intents (without permissions guided), which could potentially invoke a malware activity as in attack I. By taking a closer look at the results, among these apps, a majority (65% of apps) are vulnerable to attack II, i.e., they are vulnerable to phishing attack even if users launch trusted benign "service" apps from these apps. Moreover, 14.36% "service" apps can be exploited to "help" attack the apps who invoke these "service" apps, even if the apps being attacked may not be vulnerable by themselves.

The consequence and severity of a phishing attack depend on the content and sensitivity of the stolen information. To have a rough idea of the potential consequences caused by the "Back Hijacking" phishing attacks, we selectively show in Figure 8(a) the population of vulnerable victim apps in a few security-sensitive app categories. Noticeably, We observe that a significant number of security-sensitive apps are vulnerable, including the financial apps like banking and credit card payment (e.g., Citibank, Chase, Google Wallet), the most popular communication and social media apps (e.g. Google Hangouts, facebook), and shopping apps from the ma-

Activity Attribute	% of Apps	Intent Flag	% of Apps
allowTaskReparenting="true"	0.80	NEW_TASK	79.42
launchMode="singleTask"	24.63	CLEAR_TOP	37.59
launchMode= other non-default modes	24.75	EXCLUDE_FROM_RECENTS	10.08
taskAffinity= own pck. name	2.36		
taskAffinity= other	1.60	**Events**	
excludeFromRecents="true"	12.45	onBackPressed()	62.00
alwaysRetainTaskState="true"	2.03	TaskStackBuilder	7.27
		startActivities()	5.47

Table 5: Percentage of 6.8 million market apps that use each of the "security-sensitive " task control knobs.

Task Affinity	# of Apps
com.android.settings	492
com.android.camera	325
com.android.update	279
com.tencent.mm	273
com.gau.go.launcherex	237
com.fractalist	194
com.android.activity	158
com.xiaomi.payment	147

Table 6: Top package names specified as the task affinity by other apps

jor electronic commerce companies (e.g. Ebay, Amazon Shopping), etc. Similarly, in Figure 8(b), we show the statistics of a few app categories in which the vulnerable "service" apps and their functionality are most widely used, including the most famous photo editing tools, document editors, and file sharing services, etc.

6.2 Market-scale Study on the Use of Task Control Knobs

Due to the task hijacking threats, we have a pressing need for a defense strategy that can mitigate these threats while minimizing the side effects on Android multitasking features. To this end, it is important to first understand the current status about the use of Android multitasking features in real implementation, especially the use of "security-sensitive" task control knobs.

We analyzed 6.8 million Android apps from a variety of markets including Google Play and other 12 popular third-party app markets worldwide (e.g., from China). The analysis does not include duplicated apps (apps with same package name, public key certificate and app version number) distributed across multiple markets.

Table 5 shows the percentage of apps that use each of the task control knobs respectively. As shown in the table, a majority of the task control features are popular with app developers and users. For example, "singleTask" launch mode and NEW_TASK intent flag are used in a significant portion of apps to control the association of new activities with tasks. The flexibility of "back" button customization is widely adopted (as high as 62% apps). One reason is that the `onBackPressed()` callback function is heavily used by ad libs (which embed ads in app activities) for data clean-up before the activities are destroyed. In addition, a significant portion of activities can hide their associated tasks from the overview screen (by defining "excludeFromRecents" attribute or setting EXCLUDE_FROM_RECENTS intent flag).

Case Study - Task Affinity: Since task affinity can be abused in the most dreadful attacks, we are particularly interested in its use. 3.96% apps we studied explicitly declare task affinity. A considerable portion (1.6% of all apps) set their activities' taskAffinity string without containing their own package names. It means that, if there are task affinity conflicts, these 1.6% apps (totally 109 thousand apps) may interfere with the multitasking behaviors of one another. They may even affect other apps if the task affinity attributes are intentionally set to the package name of other apps (recall that the taskAffinity string can be set arbitrarily). We are especially interested in the latter case, and in our analysis, we find a total of 3293 apps of this kind. Table 6 lists the top package names designated as task affinity by these apps.

By reverse engineering a number of these apps, we find that intentionally setting the task affinity as another app is particularly useful in a class of "plug-in" apps, i.e. apps that provide complementary features to existing (and usually popular) apps just like a web browser's plug-ins (except that here the "plug-in" itself is implemented in a separate app). By being in the same task with the popular app, the "plug-in" app can change normal user experience and fulfill its feature functionality in the context of the app it serves. For example, an phone call recorder app namely FonTel can display an array of buttons on screen whenever there is a phone call, letting users to control phone call recording. The control buttons are contained in an mostly transparent activity. By setting the task affinity of the activity to `com.android.phone`, it can be pushed on top of the Android telephony task when a phone call occurs, such that users can access both the recording control buttons and telephony activity at the same time.

In summary, despite the security risks, Android multitasking features are popular with developers and even become indispensable to the normal functions of a significant number of apps that provide favorable features.

7 Defense Discussion

Given the pervasive use of the "security-sensative" task control features, simply disabling these features would greatly hurt app functions and user experience. Mitigating the task hijacking threats become a trade-off between app security and multitasking features.

7.1 Detection in Application Review

Existing app vetting processes such as Bouncer [31] may conduct a inspection over the "sensitive" task control knobs, a light-weight defense strategy without significantly affecting existing multitasking features.

However, specifying a guideline balancing the security/feature trade-off is non-trivial. For example, a tentative guideline could be: `taskAffinity` attribute should be specified in a strict format, e.g., with app package name followed by developer-defined affinity name (now task affinity can be any string); and the task affinity should not contain any other app's package name, except that the two apps are from the same developer. This effectively eliminates a big portion of hijacking state transitions where a malicious activity specifies the victim app as its preferred affinity. However, this rule also restricts useful features and contradicts with an important principle of Android multitasking design - give an activity the freedom to live in its preferred task even though they are from different apps. This contradiction cannot be solved by app review alone in this case. We need system support together with app review to achieve a good balance of security/feature trade-off.

Moreover, detecting problematic events can be sometimes difficult for the app review. For instance, one could confine the behaviors in `onBackPressed()`, preventing it from generating potential hijacking transition event. However, discovering all possible program behaviors using static analysis is an undecidable problem. A skillful attacker can replace class methods (`onBackPressed()` method in `Activity` class) with another method by changing Dalvik internals using native code during runtime, and static analysis does not know this by simply looking at the original `onBackPressed()` method. Dynamic analysis is of little help as well since this behavior can be triggered only after passing the app review.

As a result, completely mitigating task hijacking risks and without affecting existing features in app review remains challenging.

7.2 Secure Task Management

An alternative approach involves security enhancement to the task management mechanism of Android system.

A more secure task management could introduce additional security guides or logic, which draws developers' awareness of the security risk and limits the attacker surface. Take the above task affinity for example, an additional boolean attribute can be introduced for each app to decide if it allows the activities from other apps to have the same affinity as the app. If the boolean is "false" (also by default), the system would not unconditionally relocate the "alien" activities to the app's task or vice versa, even though the "alien" activities declare to have the same task affinity as the app. Likewise, a finer-grained boolean attribute can be further employed for `allowParentReparenting` attribute - determining if to allow "alien" activities to be re-parented to the app's task (even though defining the same task affinity is permitted). For other "security-sensative" features, we suggest first consider the same approach. Considering the serious security hazards that can be prevented, it is well worth of making such changes. At the very least, enhanced security scheme like this has to be applied to assure the security of the most privileged system apps.

Completely defeating task hijacking is not easy. As we have discussed in the last section, it is difficult to identify the exact behavior of pressing "back" in an activity during app review phase. For these popular and security-sensitive features, more powerful runtime monitoring mechanism is required to fully mitigate task hijacking threats.

In summary, we advocate future support for security guidance and/or mechanism, which can protect Android apps from task hijacking threats and bring along a both secure and feature-rich multitasking environment for Android users and developers.

8 Related Work

GUI security : GUI security has been extensively studied in traditional desktop and browser environments [14, 29], e,g., UI spoofing [9], clickjacking [3, 17], etc. Android, on the other hand, is unique in the design of its GUI sub-systems. It has been shown that the GUI confidentiality in Android can be breached by stealthily taking screen shots due to adb flaws [22], via embedded malicious UIs [28, 24], or through side channels, e.g. shared-memory side channel [8] or reading device sensors information [25, 34]. In contrast to existing work, this paper focuses on the fundamental design flaws of the task management mechanism (supported by the AMS), the control center that organizes and manages all existing UI components in the Android system.

Android Vulnerability: The security threats in the inter-component communication (ICC) has been widely studied [13, 23, 10, 20, 32]. Moreover, there has been considerable prior work on emerging Android vulnerabilities and their mitigation measures in many aspects [38, 40, 18, 33, 27, 7, 30, 15, 21]. However, the critical Android multitasking mechanism and the feature provider, the AMS, have not been deeply studied before. This paper fills in this gap by systematically studying the Android multitasking and the security implications of this design.

Android Malware: Many prior efforts focus on large-scale detection of malicious or high-risk Android apps [39], e.g., fingerprinting or heuristic-based methods [26, 41, 16], malware classification based on machine learning techniques [37, 6], and in-depth data flow analysis for app behaviors [11, 35, 36, 6]. The attack surface discovered in this paper can be easily employed by attackers to create a wide spectrum of new malwares, as discussed in Section 5. We report our threat assessment based on over 6 million market apps and provide defense suggestions in order to prevent the outburst of task hijacking threats in advance.

9 Conclusion

This paper systematically investigated the security implications of Android task design and task management mechanism. We discover a plethora of task hijacking opportunities for attackers to launch different attacks that may cause serious security consequences. We find that these security hazards can affect all recent versions of Android. Most of our proof-of-concept attacks are able to attack all installed apps including the most privileged system apps. We analyzed over 6.8 million apps and found task hijacking risk prevalent. We notified the Android team about these issues and we discussed possible mitigation techniques.

10 Acknowledgment

We would like to thank anonymous reviewers whose comments help us improve the quality of this paper. We thank Dr. Sencun Zhu and Dr. Dinghao Wu from Pennsylvania State University for providing valuable feedback. Chuangang Ren was supported in part by ARO W911NF-09-1-0525 (MURI). Peng Liu was supported by ARO W911NF-09-1-0525 (MURI) and ARO W911NF-13-1-0421 (MURI).

References

[1] Task and Back Stack. http://developer.android.com/guide/components/tasks-and-back-stack.html.

[2] App Manifest. http://developer.android.com/guide/topics/manifest/activity-element.html.

[3] AKHAWE, D., HE, W., LI, Z., MOAZZEZI, R., AND SONG, D. Clickjacking Revisited: A Perceptual View of UI Security. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)* (2014).

[4] Android Activity. http://developer.android.com/reference/android/app/Activity.html.

[5] Simplocker: First Confirmed File-Encrypting Ransomware for Android, 2014. http://www.symantec.com/connect/blogs/simplocker-first-confirmed.

[6] ARP, D., SPREITZENBARTH, M., HUBNER, M., GASCON, H., AND RIECK, K. Drebin: Effective and Explainable Detection of Android Malware in Your Pocket. In *Proceedings of Network and Distributed System Security Symposium (NDSS)* (2014).

[7] CHEN, E., PEI, Y., CHEN, S., TIAN, Y., KOTCHER, R., AND TAGUE, P. OAuth Demystified for Mobile Application Developers. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)* (2014).

[8] CHEN, Q. A., QIAN, Z., AND MAO, Z. M. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *Proceedings of the USENIX Security Symposium* (2014).

[9] CHEN, S., MESEGUER, J., SASSE, R., WANG, H., AND WANG, Y. A Systematic Approach to Uncover Security FLaws in GUI Logic. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)* (2007).

[10] CHIN, E., FELT, A. P., GREENWOOD, K., AND WAGNER, D. Analyzing Inter-Application Communication in Android. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)* (2011).

[11] ENCK, W., GILBERT, P., CHUN, B., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2010).

[12] FELT, A. P., CHIN, E., HANNA, S., SONG, D., AND WAGNER, D. Android Permissions Demystified. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)* (2011).

[13] FELT, A. P., WANG, H. J., MOSHCHUK, A., HANNA, S., AND CHIN, E. Permission Re-delegation: Attacks and Defenses. In *Proceedings of the USENIX Security Symposium* (2011).

[14] FESKE, N., AND HELMUTH, C. A Nitpickers Guide to a Minimal-complexity Secure GUI. In *Proceedings of Annual Computer Security Applications Conference (ACSAC)* (2005).

[15] GRACE, M., ZHOU, Y., WANG, Z., AND JIANG, X. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of Network and Distributed System Security Symposium (NDSS)* (2012).

[16] GRACE, M., ZHOU, Y., ZHANG, Q., ZOU, S., AND JIANG, X. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)* (2012).

[17] HUANG, L., MOSHCHUK, A., WANG, H. J., SCHECHTER, S., AND JACKSON, C. Clickjacking: Attacks and Defenses. In *Proceedings of the USENIX Security Symposium* (2012).

[18] JIN, X., HU, X., YING, K., DU, W., AND YIN, H. Code Injection Attacks on HTML5-based Mobile Apps: Characterization, Detection and Mitigation. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)* (2014).

[19] K. W. Y. AU AND Y. ZHOU AND Z. HUANG AND D. LIE. PScout: Analyzing the Android Permission Specification. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)* (2012).

[20] KANTOLA, D., CHIN, E., HE, W., AND WAGNER, D. Reducing attack surfaces for intra-application communication in android. In *Proceedings of the ACM workshop on Security and privacy in smartphones and mobile devices (SPSM)* (2012).

[21] Li, T., Zhou, X., Xing, L., Lee, Y., Naveed, M., Wang, X., and Han, X. Mayhem in the Push Clouds: Understanding and Mitigating Security Hazards in Mobile Push-Messaging Services. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)* (2014).

[22] Lin, C., Li, H., Zhou, X., and Wang, X. Screenmilker: How to Milk Your Android Screen for Secrets. In *Proceedings of Network and Distributed System Security Symposium (NDSS)* (2014).

[23] Lu, L., Li, Z., Wu, Z., Lee, W., and Jiang, G. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)* (2012).

[24] Luo, T., Hao, H., Du, W., Wang, Y., and Yin, H. Attacks on WebView in the Android System. In *Proceedings of Annual Computer Security Applications Conference* (2011).

[25] Miluzzo, E., Varshavsky, A., and Balakrishnan, S. TapPrints: Your Finger Taps Have Fingerprints. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)* (2012).

[26] Peng, H., Gates, C., Sarma, B., Li, N., Qi, Y., Potharaju, R., Nita-Rotaru, C., and Milloy, I. Using Probabilistic Generative Models for Ranking Risks of Android Apps. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)* (2012).

[27] Poeplau, S., Fratantonio, Y., Bianchi, A., Kruegel, C., and Vigna, G. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Proceedings of Network and Distributed System Security Symposium (NDSS)* (2014).

[28] Roesner, F., and Kohno, T. Securing Embedded User Interfaces: Android and Beyond. In *Proceedings of the USENIX Security Symposium* (2013).

[29] Shapiro, J., Vanderburgh, J., Northup, E., and Chizmadia, D. Design of the EROS Trusted Window System. In *Proceedings of the USENIX Security Symposium* (2004).

[30] Sounthiraraj, D., Sahs, J., Greenwood, G., Lin, Z., and Khan, L. SMV-HUNTER: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps. In *Proceedings of Network and Distributed System Security Symposium (NDSS)* (2014).

[31] Android and Security, 2012. `http://googlemobile.blogspot.com/2012/02/android-and-security.html`.

[32] Wei, F., Roy, S., Ou, X., and Robby. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)* (2014).

[33] Wu, L., Grace, M., Zhou, Y., Wu, C., and Jiang, X. The Impact of Vendor Customizations on Android Security. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)* (2013).

[34] Xu, Z., Bai, K., and Zhu, S. TapLogger: Inferring User Inputs on Smartphone Touchscreens Using On-board Motion Sensors. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Netowrks (WiSec)* (2012).

[35] Yang, W., Xiao, X., Andow, B., Li, S., Xie, T., and Enck, W. AppContext: Differentiating Malicious and Benign Mobile App Behavior under Contexts. In *Proceedings of International Conference on Software Engineering (ICSE)* (2015).

[36] Yang, Z., Yang, M., Zhang, Y., Gu, G., Ning, P., and Wang, X. S. AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection. In *Proc. CCS13*.

[37] Zhang, M., Duan, Y., Yin, H., and Zhao, Z. Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. In *Proc. CCS14*.

[38] Zhou, X., Demetriou, S., He, D., Naveed, M., Pan, X., Wang, X., Gunter, C., and Nahrstedt, K. Identity, Location, Disease and More: Inferring Your Secrets from Android Public Resources. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)* (2013).

[39] Zhou, Y., and Jiang, X. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)* (2012).

[40] Zhou, Y., and Jiang, X. Detecting Passive Content Leaks and Pollution in Android Applications. In *Proceedings of Network and Distributed System Security Symposium (NDSS)* (2013).

[41] Zhou, Y., Wang, Z., Zhou, W., and Jiang, X. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of Network and Distributed System Security Symposium (NDSS)* (2012).

Cashtags: Protecting the Input and Display of Sensitive Data

Michael Mitchell, An-I Andy Wang
Florida State University

Peter Reiher
University of California, Los Angeles

Abstract

Mobile computing is the new norm. As people feel increasingly comfortable computing in public places such as coffee shops and transportation hubs, the threat of exposing sensitive information increases. While solutions exist to guard the communication channels used by mobile devices, the visual channel remains largely open. Shoulder surfing is becoming a viable threat in a world where users are often surrounded by high-power cameras, and sensitive information can be extracted from images using only modest computing power.

In response, we present Cashtags: a system to defend against attacks on mobile devices based on visual observations. The system allows users to safely access pieces of sensitive information in public by intercepting and replacing sensitive data elements with non-sensitive data elements before they are displayed on the screen. In addition, the system provides a means of computing with sensitive data in a non-observable way, while maintaining full functionality and legacy compatibility across applications.

1. Introduction

Shoulder surfing has become a concern in the context of mobile computing. As mobile devices become increasingly capable, people are able to access a much richer set of applications in public places such as coffee shops and public transportation hubs. Inadvertently, users risk exposing sensitive information to bystanders through the screen display. Such information exposure can increase the risk of personal, fiscal, and criminal identity theft. Exposing trade or governmental secrets can lead to business losses, government espionage, and other forms of cyber terrorism [12, 13, 14].

This problem is exacerbated by the ubiquity of surveillance and high-power cameras on mobile devices such as smartphones and on emerging wearable computing devices such as Google Glass [57]. Additionally, the trend toward multicore machines, GPUs, and cloud computing makes computing cycles more accessible and affordable for criminals or even seasoned hobbyists to extract sensitive information via off-the-shelf visual analysis tools [58].

This paper presents the design, implementation, and evaluation of Cashtags, a system that defends against shoulder surfing threats. With Cashtags, sensitive information will be masked with user-defined aliases, and a user can use these aliases to compute in public. Our system is compatible with legacy features such as auto correct, and our deployment model requires no changes to applications and the underlying firmware, with a performance overhead of less than 3%.

1.1 The shoulder-surfing threat

The threat of exposing sensitive information on screen to bystanders is real. In a recent study of IT professionals, 85% of those surveyed admitted seeing unauthorized sensitive on-screen data, and 82% admitted that their own sensitive on-screen data could be viewed by unauthorized personnel at times [1]. These results are consistent with other surveys indicating that 76% of the respondents were concerned about people observing their screens [2], while 80% admitted that they have attempted to shoulder surf the screen of a stranger [3].

The shoulder-surfing threat is worsening, as mobile devices are replacing desktop computers. More devices are mobile (over 73% of annual technical device purchases [4]) and the world's mobile worker population will reach 1.3 billion by 2015 [5]. More than 80% of U.S. employees continues working after leaving the office [6], and 67% regularly access sensitive data at unsafe locations [2]. Forty-four percent of organizations do not have any policy addressing these threats [1]. Advances in screen technology further increase the risk of exposure, with many new tablets claiming near 180-degree screen viewing angles [8].

1.2 The dangers are everywhere

Observation-based attacks to obtain displayed sensitive information can come in many forms. There are more than 3 billion camera-enabled phones in circulation [4]. Some of these devices can capture images at 40 megapixels of resolution and over 10 times optical zoom [7]. High-resolution and often insecure "security" cameras are abundant in major metropolitan areas. For example, the average resident of London is captured on CCTV over 300 times per day [9]. Finally, sensitive data can be captured by simple human sight.

Observation-based attacks can be complex. Partial images can be merged, sharpened, and reconstructed, even from reflections. Offline and cloud-based optical character recognition (OCR) solutions have only a small percentage of recognition errors, even on inexpensive low-end devices [10].

Personal information exposure can also enable other attacks, such as social engineering, phishing, and other personal identity theft threats.

1.3 The consequences can be severe

Observation-based leaks of sensitive information have led to significant personal and business losses. Recently, an S&P 500 company's profit forecasts were leaked as a result of visual data exposure [4]. In a different case, government documents were leaked when a train passenger photographed sensitive data from a senior officer's computer screen [11]. Security cameras captured the private details of Bank of America clients through windows [12]. Sensitive information relating to Prince William was captured and published because of a screen exposure to a bystander [13].

The risk of loss from shoulder surfing is also hurting business productivity. Figures show that 57% of people have stopped working in a public place due to privacy concerns and 70% believe their productivity would increase if others could not see their screen [2].

1.4 Current solutions

Techniques are available to limit the visual exposure of sensitive information. However, the focus of these systems has been limited to password entries [22, 23, 24, 25, 33, 34, 35]. Once the user has been successfully authenticated, all of the accessed sensitive information is displayed in full view. Clearly, such measures are insufficient for general computing in public when the need to access sensitive information arises. Unfortunately, many techniques used to prevent visual password leaks cannot be readily generalized beyond password protection, which motivates our work.

2. Cashtags

We present Cashtags, a system that defends against observation-based attacks. The system allows a user to access sensitive information in public without the fear of leaking sensitive information through the screen.

2.1 Threat model

We define the threat model as passive, observation-based attacks (e.g., captured video or physical observation by a human). We assume the attacker can observe both the screen of the user as well as any touch sequences the user may make on the screen, physical buttons, or keyboards. We also assume the absence of an active attack; the observer cannot directly influence the user in any way.

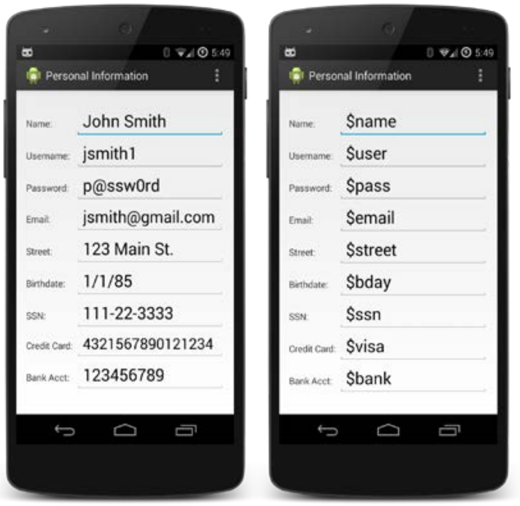
Although sensitive information can be presented in many forms, we focus on textual information to demonstrate the feasibility of our framework. Protecting sensitive information in other forms (e.g., images and bitmaps) will be the subject of future work.

2.2 User model

Conceptually, Cashtags is configured with a user-defined list of sensitive data items (Table 2), each with a respective Cashtags alias or a cashtag (e.g., `$visa` to represent a 16-digit credit-card number). Whenever the sensitive term would be displayed on the screen, the system displays the predefined alias instead (Fig 2.1).

Type	Actual	Alias
Name	John Smith	`$name`
Email	jsmith@gmail.com	`$email`
Username	Jsmith1	`$user`
Password	p@ssw0rd	`$pass`
Street Address	123 Main St.	`$addr`
Phone number	555-111-2222	`$phone`
Birthday	1/1/85	`$bday`
SSN	111-22-3333	`$ssn`
Credit Card	4321 5678 9012 1234	`$visa`
Account number	123456789	`$acct`

Table 2: Sample mapping of sensitive data to cashtag aliases.



Name: John Smith
Username: jsmith1
Password: p@ssw0rd
Email: jsmith@gmail.com
Street: 123 Main St.
Birthdate: 1/1/85
SSN: 111-22-3333
Credit Card: 4321567890121234
Bank Acct: 123456789

Name: $name
Username: $user
Password: $pass
Email: $email
Street: $street
Birthdate: $bday
SSN: $ssn
Credit Card: $visa
Bank Acct: $bank

Fig. 2.1: On-screen sensitive data (left) and data protected by masking with cashtag aliases (right).

At the point at which the sensitive data would be used internally by the device or app, cashtags will be replaced by their represented sensitive data items, allowing whatever login, communication, computation, transmission, or upload to proceed normally.

Cashtags also provides secure data input. A user can type in a cashtag in place of the sensitive term, permitting complex data-sensitive tasks, such as filling out a reward-card application without risk of observation from a bystander. In addition, cashtags are easier to remember than the actual sensitive data term. For example, `$visa` can be used as a shortcut for entering a 16-digit credit card number.

Users can interact with Cashtags by entering data in either alias or actual form. If the user enters the actual term, it will be converted into its corresponding alias once the full term is entered. This has the potential to inadvertently expose partial private data, an attacker could potentially see all but the last character input. In practice, auto completion is likely to expand the sensitive information within the first few characters and display it in the alias form. Entering data into the system in alias form ensures that no such partial information exposure can occur during input and is the best option to maximize protection.

2.3 Compared to password managers

The user model of Cashtags is similar to that of a password manager. To add an entry to a password manager, a user is required to key in the username and password pair. Typically, subsequent usage of the stored password involves only selecting the respective account pre-populated with the stored password. Therefore, an observer cannot see the keyed-in sequence for passwords. Similarly, Cashtags requires the user to pre-configure the system by first entering the sensitive term to be protected and the corresponding alias to represent the term. When a sensitive term is displayed, our system replaces the sensitive term with its alias without user intervention. To enter a sensitive term, the user can enter the alias, and our system will translate it into the sensitive term prior to being processed by the underlying apps.

While a password-manager-like user model provides a familiar interface, it also shares a similar threat vector of a centralized target and weakened protection in the case of physical theft. However, overcoming the shortcomings of password managers is orthogonal to the focus of this research and threat model. As research in bettering password managers advances, we can apply those techniques to enhance our system.

2.4 Design overview

Although conceptually simple, Cashtags addresses a number of major design points.

Intercepting sensitive data: Cashtags intercepts sensitive data items as they are sent to the display. For apps, this is at their common textual rendering library routines, and for users, this is at the routines that handle software keyboards and physical devices (e.g., USB and wireless input devices).

User interface: Users can type in cashtags instead of sensitive data items to compute in public. This interface allows cashtags to be compatible with existing tools such as auto completion, spellcheckers, and cut and paste. Thus, users can enter the first few characters and auto-complete the full cashtag.

Accessing sensitive data: User-entered cashtags are converted internally to the sensitive data items before the apps access the data. This way, Cashtags will not break applications due to unanticipated input formats.

Variants of data formats: Cashtags can leverage existing libraries to match sensitive data items represented in different formats (e.g., John Smith vs. John Q. Smith).

Development and deployment models: Cashtags uses a code-injection framework. This approach avoids modifying individual apps and the firmware, while altering the behavior of the overall system to incorporate Cashtags at runtime.

Cashtag repository: The mapping of cashtags to sensitive data items is stored in a password-protected repository.

3. Cashtags Design

This section will detail each Cashtags design point.

3.1 Observation-resistant approaches

We considered screen-level-masking and data-entry-tagging system design spaces prior to using the current keyword-oriented design. While all of these approaches can prevent sensitive information from being displayed, the main differences are the interception granularity and the portability of the underlying mechanisms.

Screen-level masking: One coarse-grained approach is to mask the full application window or screen regions when encountering sensitive information [61, 62]. While this approach prevents information leakage, it also prevents the user from computing using the sensitive information. For example, when encountering

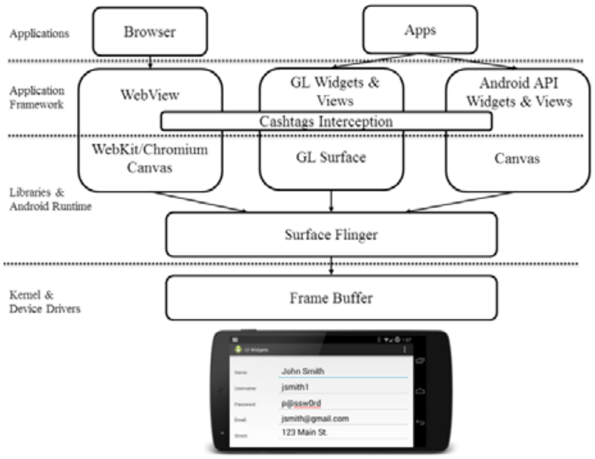


Fig. 3.1. Display data paths for the Android platform.

an online purchase screen, the entire screen could be blurred due to sensitive information, making it unusable.

Tag-based approach: A tag-based approach requires the user to predefine the data elements as sensitive. These data elements are then tracked as they propagate through the system [16]. If a tracked data element is to be displayed on the screen, the rendering is intercepted, and the tracked data element is replaced with its corresponding alias.

This approach requires a significant system modification to support this granularity of data tracking, making it less deployable. In addition, the system resources and required accounting to track the data result in significant processing overhead incurred by the system.

Keyword-based approach: Another approach is to utilize keywords and perform pattern matching on on-screen text elements. Like the tag-based approach, this option works at the individual data element or word granularity. It also has the requirement that the sensitive data element be specified to the system prior to the point of screen rendering and subsequent visual data exposure.

The primary difference, however, is the method in which the sensitive data is identified. Rather than tracking sensitive data as it propagates through the system, this method parses data fields prior to the screen display. If a predefined sensitive element is matched, it is replaced with its alias before being rendered to the screen. We chose this approach because it achieves word granularity protection without the tag-based overhead and deployment issues.

3.2 Where to intercept sensitive data

To decide where to intercept sensitive data, we first need to understand how sensitive data traverses from apps to the screen through various display data paths. Fig. 3.1 shows the display data paths under the Android application development platform. Although different, the display data paths of iOS and Windows generally have one-to-one mappings of the components.

Window manager: A typical app displays information by invoking some user-level display or graphics library routines. Various routines eventually invoke routines in the underlying window management system (e.g., Surface Flinger for Android) before information is processed by the OS and displayed on the screen.

Arguably, the window management system might seem to be a single point at which all sensitive data can be captured. Unfortunately, by the time sensitive information arrives there, some sensitive information may have been translated into bitmaps. While OCR techniques can be used to extract sensitive text, they are still too heavyweight to be used in the display data path, which is critical for user interactions. Replacing sensitive bitmaps with non-sensitive ones would pose other obstacles we would like to avoid.

Applications: Another extreme is to intercept it at the app level, where the sensitive information is introduced. Potentially, we can modify a few popular, general-purpose apps (e.g., browsers) and capture most of the sensitive information. However, such solutions may tie users to specific tools. In addition, statistics show that specific app usage accounts for 86% of user time, trending away from general-purpose browsers [56]. Thus, we would need to initially modify more apps and track their updates to achieve a good coverage.

Library routines: Thus, an intermediary ground is to intercept sensitive data within a few key display and graphics library routines.

3.3 User interface

Early design: In our early user-interface design, users-defined English-like aliases in a repository to indicate sensitive data items that they wish not to be shown (e.g., use John to represent Joe). To discern these aliases when processing, we used an alternative input channel to mark them. This initial design proved problematic.

Our initial prototype was a software keyboard app with elevated privilege to offer input across applications. This implementation would be easier to port across platforms, deploy, install, and update. However, changing keyboards in the midst of a stream of input is cumbersome in practice. This method also interacted poorly with legacy swipe-based inputs, emoticon support, auto correction, and custom dictionaries.

Further, we would need to replace the default keyboard with ours, and provide ways to switch modes between normal and sensitive entries (e.g., a screen tapping sequence). By doing so, we could retain legacy functionalities such as auto correction. On the other hand, the development effort of this approach would have been much higher, and novice users might have trouble replacing the default keyboard.

Direct input of cashtags: While there are other input interface options, the need to switch input modes to allow aliases to appear as normal text seemed superfluous (e.g., using "visa" to represent the 16-digit credit card number).

Thus, we explored the use of cashtags, where aliases are prepended with a `$` sign, to represent sensitive information. By doing so, a user can directly enter cashtags, and the mode change is encoded in the cashtag alias (e.g., use `$fname` to represent John and `$gmail` to represent jsmith@gmail.com). This method can leverage the existing custom dictionary for auto completion, which makes it easier for the user to remember and input the cashtags. This method can also utilize standard application-level development techniques, opening up the range of supported device platforms and decreasing development and installation efforts.

Direct input of sensitive information: Another supported alternative input mechanism (with some information leak) is for a user to enter the initial characters of a sensitive data item. As soon as the auto completion detects that, Jo is likely to mean Joe, for example, it will be automatically masked with `$john`. The user then can choose `$john` and proceed.

Additional Cashtags semantics: Recursion is supported, so we can use `$signature` to represent `$fname $lname $gmail`, which in turn maps to John Smith, jsmith@gmail.com. We disallow circular cashtags mappings (e.g., use `$john` to represent `$joe`, and `$joe` to represent `$john`).

3.4 Accessing sensitive information

One design issue addresses converting cashtags back to the sensitive data for access by apps. Normally, when an app wants to access the sensitive information and to send it back to the hosting server, we must make sure that the conversion is performed prior to access, so that the app would never cache, store, or transmit the cashtags. The concern is that cashtags may break an app due to the violation of the type or formatting constraints.

We also must make sure that the cashtags are actually entered by the user, not just pre-populated by the app. Otherwise, a malicious app can extract sensitive information just by displaying cashtags.

There are certain exceptions where it is desirable to operate directly on cashtags instead of the sensitive information. For example, the auto-completion task will auto complete cashtags (`$fn` to `$fname`), not the sensitive information it represents. By doing so, the handling of text span issues is simplified because cashtags usually differ in text lengths when compared to the sensitive information they represent.

3.5 Variants of data formats

Sensitive data may be represented in multiple formats. For example, names can be represented as combinations of first, last, and middle initials (e.g., John Smith; John Q. Smith). Accounts and social security numbers can be represented using different spacing and/or hyphenation schemes (e.g., 123456789; 123-45-6789). Fortunately, we can leverage existing regular expression libraries (`java.util.regex.*`) to perform such matching.

Another issue involves the type restriction of the input field. For example, a number field (e.g., SSN) may prevent the use of cashtags (`$ssn`). To circumvent these restrictions, we allow users to define special aliases (e.g., 000-00-0000) to represent certain types of sensitive information (e.g., social security numbers).

3.6 Deployment and development models

To avoid modifying individual applications, we considered two options to provide system-level changes: (1) custom system firmware images (ROMs) and (2) code-injection frameworks (e.g., Android Xposed)

By utilizing a custom system firmware image, complete control of the operating system is provided. (This approach assumes that the full source is available.) In addition, ROM-based solutions can offer a more unified testing environment. However, the changes would be restricted to device-specific builds. Only hardware for which the source is built would have access to the modified system. This also limits user preference by restricting use only for a specific system image. It would additionally require regular maintenance, and it would break vendor over-the-air update functionality.

Instead, we used a code-injection framework, which overrides library routines and incorporates our framework into execution prior to the starting of apps.

Code injection offers streamlined development, as standard application development tools can be used. In addition, these modules can be distributed and deployed as applications. Because code injection only relies on the underlying system using the same set of libraries, the resulting system is more portable and less coupled to versions and configurations of system firmware.

The installation and use of the code-injection framework requires root access to the device. This is, however, not a firm requirement and exists only for this prototype; Vendors and OEMs can incorporate Cashtags into system firmware providing the same functionality without exposing root. This deployment model is advisable to further enhance security for a production system.

3.7 Cashtags app and repository

Cashtags aliases and sensitive data items are maintained in a repository. The Cashtags app coordinates the interactions between various apps and the repository. The app also provides password-protected access to add, edit, remove, import, and export sensitive terms and corresponding cashtags.

Cashtags provides per-application blacklisting, excluding specific applications from being code-injected (or activated) with cashtag-replacement code. For example, the cashtag repository itself must be excluded due to circular dependencies. To illustrate, suppose a cashtag entry maps `$fname` to Joe. If Cashtags is enabled, the screen will show that `$fname` is mapped to `$fname`; when saved, Joe will be mapped to Joe. Apps with a low risk to reveal sensitive information can be excluded for performance issues (e.g., games, application launchers, home screens).

4. Implementation

We prototyped Cashtags on the open-source Android platform. Our code-injection framework allows Cashtags to operate on any Android device with the same display and graphics libraries and root access. This section will first detail the Android display data paths, explain the code-injection framework, and discuss how various display data paths are intercepted and how cashtags are stored.

4.1 Android display elements

Fig 3.1 has shown a top-level view of the various ways Android apps and browsers display information on the screen. This section provides further background on Android terminologies. Corresponding terminologies for text widgets on Apple and Windows devices are listed in Table 4.

	Android	Apple	Windows
Text Labels	TextView	UITextView	TextBlock
OpenGL Text	GLES20 Canvas	GLKView	Direct3D
Editable Text	TextView	UITextView	TextBlock
Webapp Text	WebView	UIWebView	WebView
Browser/ WebView	WebView	UIWebView	WebView

Table 4: Widget terminologies on Android, Apple, and Windows platforms.

The Android display is composed of views, layouts, and widgets. `View` is the base class for all on-screen user interface components.

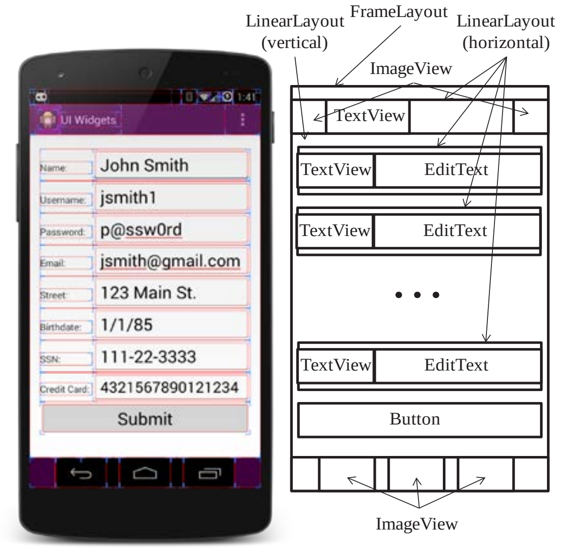
Widgets: The term widget is used to describe any graphic on-screen element. Different widgets can be used to display static text labels (e.g., `TextView`), user input boxes (e.g., `EditText`), controls (e.g., `Buttons`), and other media (e.g., `ImageView`).

`Views` are organized into `ViewGroups`, the base class for all screen layouts. `Layouts` are arrangements of views within vertical or horizontal aligned containers (e.g., `LinearLayout`), or arranged relative to other views. Nesting of `ViewGroups` and `Layouts` allows complex custom composites to be defined.

Collectively, this tree of layouts and widgets forms a view hierarchy. When the screen is drawn, the view hierarchy is converted from logical interface components into a screen bitmap. Fig. 4.1 shows a simple user input form and its composition of various widgets and layouts.

Text rendering: Text can be rendered on the screen through several mechanisms (Fig 3.1), most frequently the `TextView` widget. An `EditText` is an extension of the `TextView` that provides an interface for text input from the user via the on-screen software keyboard, hardware keypads, voice input, and gestures. These widgets can be pre-filled with text by the app internally or through suggestion or auto-correction interfaces.

Text can also be rendered on screen via `OpenGL Canvas`, other graphic rendering libraries, browser rendering engines, and other methods, many of which do not inherit from the base `TextView` widget. This plethora of methods complicates practical capture of all possible ways text could get onto the screen.



Name: John Smith
Username: jsmith1
Password: p@ssw0rd
Email: jsmith@gmail.com
Street: 123 Main St.
Birthdate: 1/1/85
SSN: 111-22-3333
Credit Card: 4321567890121234

Fig. 4.1. Decomposition of on-screen views, layouts, and widgets of a simple app input form.

4.2 Android code-injection framework

Cashtags uses the Android Xposed code-injection framework. The development cycle is accelerated by short-circuiting the need to perform complete device firmware rebuilds from scratch.

Underneath the hood, whenever an app is started, Android forks off a new virtual machine. The Android Xposed framework allows overriding library routines to be inserted into the Java `classpath`, prior to the execution of the new virtual machines. Thus, the overall system behavior is altered without modifying either the apps or the underlying firmware.

Individual class methods can be *hooked*, allowing injected code to be executed prior to, following the completion of, or in place of the base-method calls. Private or protected member fields and functions can also be accessed and modified, and additional fields or functions can be added to the base class or object granularity.

4.3 Sensitive data intercepting points

With the background of the Android display data paths (Fig. 3.1) and the code-injection framework, we can determine where and how to intercept sensitive information. Since all text-displaying screen widgets are descendants of the `TextView` class (Fig. 4.2), we only needed to hook `TextView` (`android.widget.TextView`) to intercept all widgets containing static sensitive text. For user input, we hooked `EditText` (`android.widget.EditText`) to capture sensitive data or cashtags entered via on-screen software keyboards, (integrated, plugged, or wirelessly connected) hardware keypads, voice input, and gestures. For display through the OpenGL libraries, we intercepted GLText (`android.view.GLES20Canvas`). For browsers, we intercepted `WebView` (`android.WebKit/WebView`).

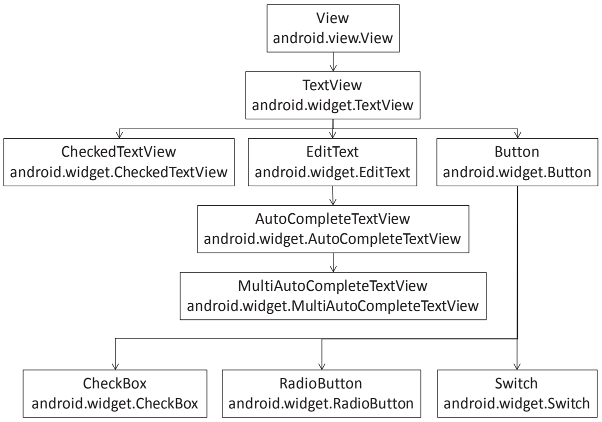


Fig. 4.2. Simplified inheritance hierarchy of Android on-screen views and widgets.

4.4 `TextView`

Fig. 4.3 shows a simplified version of the implementation of the `TextView` widget in the Android API, present since version 1 of the Android SDK. The `getText`() and `setText`() methods of the `TextViews` are hooked and modified (the `setText`() method in `TextView` is inherited by `EditText`, to be detailed later). We also added `mAlias` to map the sensitive text to the corresponding cashtag.

4.5 `EditText`

In addition to the functionality inherited from `TextView`, `EditText` must also handle additional actions that can be performed by the user, app, or system to modify on-screen text.

```
public class TextView extends View
  implements ViewTreeObserver.
  OnPreDrawListener {
    ...
    private CharSequence mText;
    private CharSequence mAlias:
    ...
    public CharSequence getText() {
      return mText;
    }
    ...
    private void setText(CharSequence
     text, BufferType type, boolean
     notifyBefore, int oldlen) {
      ...
      mBufferType = type;
      mText = text;
    }
    ...
  }
```

Fig. 4.3. Simplified TextView implementation. Bolded functions getText() and setText() are hooked and modified. An additional private field mAlias is added for mapping to a displayed cashtag, if applicable.

For cases where the system or app has pre-populated a text box with input, the TextView injection handles the cashtag replacement. Since the EditText class extends from the TextView base class, this functionality is provided through inheritance. This is also the case for nearly every other on-screen text widget, which also descend from the base TextView.

Fig. 4.4 and Fig. 4.5 show how Cashtags interacts with TextView and EditText objects. When these getText() and setText() methods are called by the app or through system processes, such as auto correct or to be rendered on screen, Cashtags will determine whether to return the alias or the sensitive data, depending on the caller.

User input can be entered through software keyboards or through physical devices. In both cases, Cashtags operates similar to, and through the same interface, as the auto-correct service. This TextWatcher (android.text.TextWatcher) interface handles events when on-screen text has been modified. EditTexts internally maintain an array of these TextWatcher event handlers. Cashtags, as one of these handlers, is activated after any character is modified within the text field.

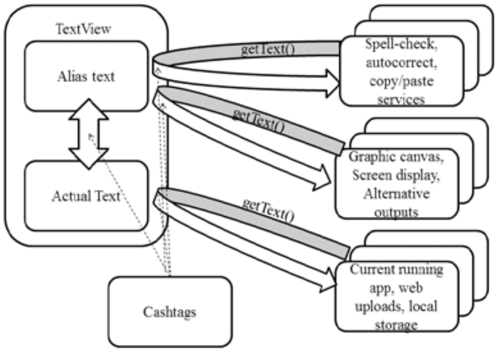

TextView
Alias text
Actual Text
getText()
getText()
getText()
Spell-check, autocorrect, copy/paste services
Graphic canvas, Screen display, Alternative outputs
Current running app, web uploads, local storage
Cashtags

Fig. 4.4. Interactions among Cashtags, TextView, and other software components. getText() returns either the cashtag or actual text depending upon the service making the request.

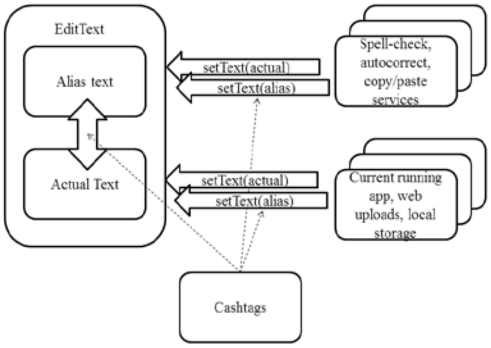


Fig. 4.5. Interactions among Cashtags, EditText, and other software components. setText() returns either the cashtag or actual text depending upon the service making the request.

This functionality is also achieved through the view OnFocusChangeListener (android.view.OnFocusChangeListener). This event handler works at the granularity of the full text field rather than the individual characters of the TextWatcher. This is more efficient, since the text replacement only occurs once per text field. It does, however, risk additional on-screen exposure of sensitive information, since direct input of actual sensitive terms would remain on-screen as long as the cursor remains in that text field. The input of the cashtag alias does not have this risk and further reduces any partial exposure during term input.

In both cases, the constructor of the EditText class is hooked and the corresponding OnFocusChangeListener or TextWatcher object is attached at runtime. User settings allow activation of either or both interception methods.

4.6 `OpenGLES Canvas`

The implementation for `OpenGLES Canvas` is similar to the base `TextView` only with different parameter types. The only distinction is that no accompanying `getText`() equivalent is present in this object, so no manipulation is necessary beyond `drawText`().

4.7 `WebView`

Unlike the previous screen widgets, the relevant interception points for screen rendering for WebKit or Chromium browser engines are all below the accessible Android/Java layer. Thus, they cannot be code-injected through the same mechanisms used for previous screen widget cases. We attempted custom compilations of the browser engines with similar widget display interception, but abandoned the effort for portability concerns.

Instead, `WebView` interception is handled similarly to a web browser plug-in. This decision maintains the portability goal of the system design.

Cashtags intercepts web rendering immediately before it is first displayed on-screen. The HTML is pre-processed with JavaScript to extract the DOM. Cashtags iterates over the text nodes and makes the appropriate text replacements of sensitive data to corresponding cashtags.

Other options were explored using specific browser and proxy requests through the web server. However, all apps that use cross-platform frameworks (PhoneGap, Apache Cordova, JQuery Mobile, etc.) run locally and could not easily be piped though this service. For this reason, we favored the plug-in approach over other alternatives.

4.8 Cashtags repository

Sensitive terms are stored as encrypted `SharedPreference` key/value pairs, which uses the AES encryption from the Java Cryptography Architecture (`javax.crypto.*`). This structure is accessed by enabled apps through the `XposedSharedPreference` interface.

5. Evaluation Methods

Cashtags was evaluated for how well the intercepted points prevent specified information from being displayed on the screen, verified by screen captures and OCR processing. This coverage was measured by enumerating common paths sensitive text can traverse to the screen. We also evaluated user input through popular apps, making sure that the cashtags reverted to the sensitive data items when accessed by the apps. Finally, we evaluated the performance overhead of Cashtags.

5.1 API coverage evaluation

For Android API coverage, we focus on the `TextView` and `EditText` display data paths, which account for 86% of usage hours for mobile devices [56], though Cashtags can also handle the remaining 14% of browser-based access. The selected sensitive information (Table 2) is based on the Personally Identifiable Information (PII) specified by the U.S. government and NIST standards [59]. We enumerate all combinations of the input phrase type (e.g., numbers, strings), case sensitivity, common widget, layout, theme, and lifecycle for these data paths. Each combination is used to demonstrate that the PII terms are not displayed on the screen from the app, as user input of the sensitive data, or as user input of the cashtag alias. In all three cases, we also demonstrate that the PII term is correctly returned from Cashtags when used by the app.

This totals 1,728 tests for the static text widgets and inputs, with 526 additional cases for widgets that permit user input via software keyboards as well as physical devices (on-board hardware, USB, or wireless input devices). Table 5.1 shows the full list of configurations.

For each combination, the Android Debug Bridge [60] and UIautomator tool [36] are used to capture the device layout view hierarchies and screenshots. The contents of the actual and cashtag fields are compared for conversion correctness. The device screenshot is processed using Tessseract OCR [21] to confirm if the actual PII term has been properly masked on the screen.

For each combination, we also demonstrate that both text input as a sensitive term and cashtag are correctly converted to the sensitive term when accessed by the app. Since the access of sensitive data within the app normally involves remote actions, we also emulated this scenario and performed remote verification. Once screen processing is completed, the app accesses the text fields and uploads them to Google Sheets/Form. The uploaded sensitive items and cashtag submissions are compared for accuracy based on expected values.

Our results show that Cashtags behaves correctly for all test cases. For each test case, Cashtags identified input containing sensitive data in both the actual and cashtag form, prevented the display on the screen of the sensitive term, and determined correctly when to convert back to the sensitive data.

Input phrase type (4):	Alphabetic phrase, numeric phrase, alphanumeric phrase, Alphanumeric with symbols.
Phrase case (2):	Case Sensitive Text, Case In-sensitive Text
Widget type (9):	`TextView` (`android.widget.TextView`), `CheckedTextView` (`android.widget.CheckedTextView`), `Button` (`android.widget.Button`), `CheckBox` (`android.widget.CheckBox`), `RadioButton` (`android.widget.RadioButton`), `Switch` (`android.widget.Switch`), `EditText` (`android.widget.EditText`), `AutoCompleteTextView` (`android.widget.AutoCompleteTextView`), `MultiAutoCompleteTextView` (`android.widget.MultiAutoCompleteTextView`)
Layout type (2):	`LinearLayout` (`android.widget.LinearLayout`), `RelativeLayout` (`android.widget. RelativeLayout`)
Theme type (3):	Default theme, System theme, User-defined theme.
Generation method (2):	Static XML, Dynamic Java
Lifecycle type (2):	Activity-based lifecycle, Fragment-based lifecycle

Table 5.1: Android API test combinations.

5.2 App coverage evaluation

The Google Play market has more than one million of published applications accessible by thousands of different hardware devices [70], making the enumeration of all possible users, devices, and application scenarios infeasible. Thus, we chose a representative subset of popular apps to demonstrate coverage of Cashtags. Categorically, these application types are email, messaging, social media, cloud and local storage, office, and finance. Table 5.2 shows the selected apps, arranged according to these categories. These apps were selected using download metrics from the Google Play marketplace, excluding games and utility apps for lack of relevance in terms of displaying sensitive data on screen. The presence of a form of external verification was also used in the application selection. Apps typically bundled with mobile devices were also tested for correct operation.

Email: AOSP Email, Gmail, K9 Mail:	User reads an email containing a sensitive term in its cashtag form. A Cashtags-enabled system should display the email with two instances of the cashtag. User composes an email with a sensitive term and its cashtag. Remote system not running Cashtags should display email with two sensitive term instances.
Messaging: Messaging, Hangouts, Snapchat:	User reads a message containing a sensitive term and cashtag. A Cashtags-enabled system should display the message containing two instances of the cashtag. User composes a message with a sensitive term and its cashtag. A remote system not running Cashtags should receive the message with two sensitive term instances.
Social: Facebook, Twitter, Google+:	User reads text with a sensitive term and its cashtag from a tweet/post/update. A Cashtags-enabled system should display the tweet/post/update with two instances of the cashtag. User composes a new tweet/post/update with a sensitive term and its cashtag. A remote system not running Cashtags should receive the tweet/post/update with two sensitive term instances.
Storage: Dropbox, MS OneDrive, File Manager:	User opens an existing file containing a sensitive term and its cashtag. A Cashtags-enabled system should display the file containing two instances of the cashtag. User creates a file with a sensitive term and its cashtag. A remote system not running Cashtags should display file with two sensitive term instances.
Office: GoogleDocs, MS Office, QuickOffice:	User reads a document containing a sensitive term and its cashtag. A Cashtags-enabled system should display the file with two instances of the cashtag. User creates a document containing a sensitive term and its cashtag. Remote system not running Cashtags should see two sensitive term instances.
Finance: Google Wallet, Paypal, Square:	User reads a document containing a sensitive term and its cashtag. Cashtag-enabled system should display the document with two instances of cashtag. User creates a document containing a sensitive term and its cashtag. A remote system not running Cashtag should see two sensitive term instances.

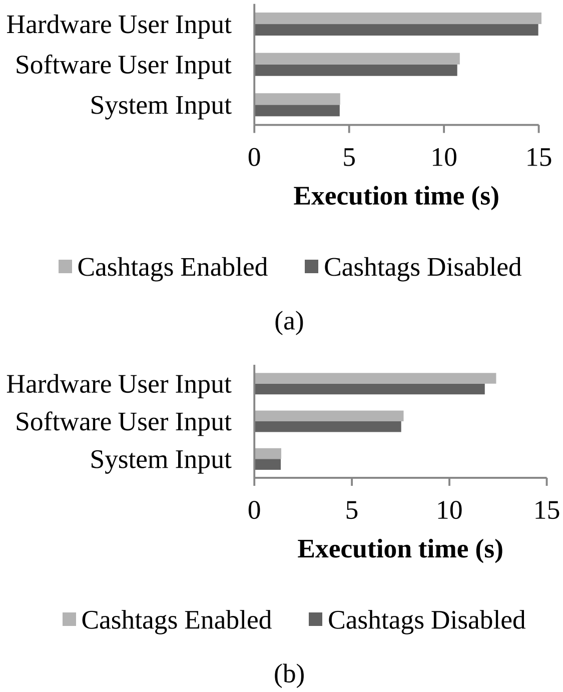
Table 5.2: Per-category app test tasks.

The operation performed on each is based on a commonly performed use case or task for each category. Table 5.2 shows the operation performed for each category and respective app.

Our results show that Cashtags behaves correctly for 97% of task and app combinations, except the MS Office Mobile tests. The reason is because of the custom View (`docRECanvasHost`) used for the primary user interaction, which is not a descendant of an `EditText`. Thus, our system does not intercept it. All of the other apps tested have user input through an `EditText`, or a custom class inheriting from an `EditText`. This exception, as well as other custom views could be made to work with Cashtags using case-specific handling for internal functions and parameters that map to the equivalent `EditText` functions.

5.3 Overhead

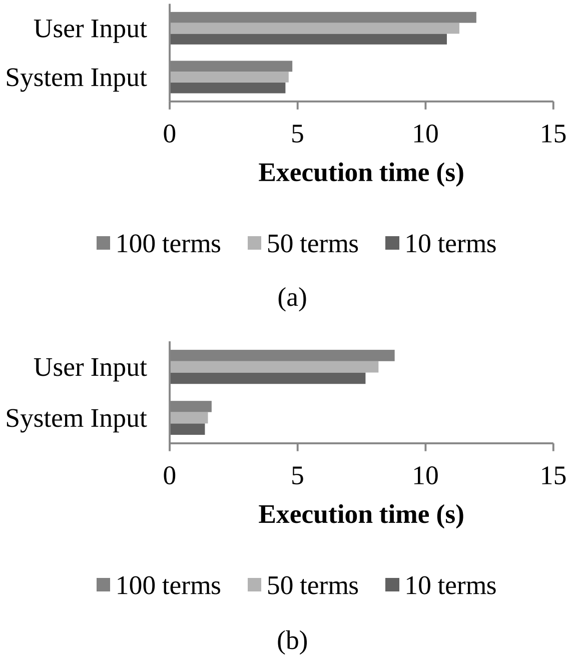
Regarding overhead, we measured the incremental lag Cashtags added to the system. We ran a modified version of the Android API coverage test (Section 5.1) with and without Cashtags enabled. The screenshots, layout hierarchy dumping, and all other non-essential automation elements were removed prior to the test execution. The test execution durations are compared, and the additional lag introduced by the system was calculated. This test was run with and without the remote data verification to determine the effects of network lags on our system overhead.



(a)

(b)

Fig. 5.1. Comparison of mean app task execution time with and without Cashtags enabled, using system, software and hardware text input (a) with and (b) without web request for tests. Hardware refers to input from physically or wirelessly connected hardware keyboard and software refers to input via on-screen software keyboard.



(a)

(b)

Fig. 5.2. Comparison of mean app task execution time with an increasing number of cashtag entries, using system and user inputs (a) with and (b) without web request for tests.

Fig. 5.1(a) shows the Cashtags system incurs an average 1.9% increase in application runtime. For tests including remote verification, Cashtags incurred an average of 1.1% increase over the baseline tests. For tests excluding the time consuming remote verification, Fig. 5.1(b) shows that Cashtags incurred an average of 2.6% over baseline. Therefore, the additional overhead of Cashtags would not be perceivable to the user.

Testing was repeated using 50 and 100 items, which is significantly more than the list of terms specified by PII. Fig. 5.2 show that the performance degrades linearly as the number of cashtags entries increases. However, we can easily replace the data structure to make the increase sublinear.

Cashtags is additionally evaluated for boot-time overhead. Changes to the Cashtags repository currently require a reboot to take full effect. While this operation is not in the critical path, the overhead is relevant. The results of the boot lag are shown in Fig. 5.3.

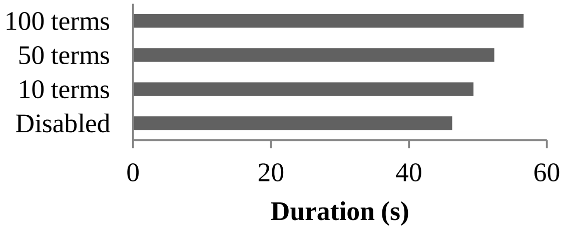


Fig. 5.3. Comparison of device startup times with varying number of cashtag entries and with system disabled.

5.4 Usability overhead

To demonstrate the usability of Cashtags, we calculated the configuration and usage overhead seen by a typical user working on typical data. We used common sensitive data elements with known character lengths (such as credit card numbers) as well as those for which published data on average term length was available (such as common names) [63, 64].

For other elements without known average length availability, we substituted a typical minimum accepted value. Table 5.3 shows the comparison of these fields against the suggested cashtag alias length.

In nearly every case, the cashtag alias is significantly shorter than the average length of the sensitive data element. On a keystroke count basis, the amortized effort of the initial configuration can be overcome with only two or three uses of the Cashtag alias. Longer names and emails require additional keystrokes for initial configuration but yield greater keystroke savings for each time the data element is entered. In addition, the aliases in the table are based on the suggested terms for ease of user recall; even shorter terms could be substituted to reduce additional data entry overhead.

5.5 Quantification of the time savings for an end user

The real efficiency savings for Cashtags is the ability to represent hard-to-recall terms (e.g., account numbers) with easy-to-recall aliases. The user simply assigns the sensitive data a memorable alias and can refer to them.

This also adds convenience when sensitive data changes. For example, consider the case of a stolen credit card. Once the replacement is issued, the user need only to update the underlying data element, continuing to use the alias without the need to memorize a new number. In some cases of personal information change, the defunct data could still be considered sensitive and be prevented from being displayed on the screen. In such cases, the defunct data element could be assigned to a new alias. For example, consider a street address change: the alias `$street` is assigned to the new data, and the past street address can be assigned to `$street_old`.

Type	Actual	Alias	Alias	Diff
First Name	6	`$fname`	6	0
Last Name	6	`$lname`	6	0
Email	20	`$email`	6	14
Username	9	`$user`	5	4
Password	9	`$pass`	5	4
Phone number	10	`$cell`	5	5
Birthday	10	`$bday`	5	5
SSN	9	`$ssn`	4	5
Credit Card	16	`$visa`	5	11
Acct. number	12	`$acct`	5	7

Table 5.3: Typical keystroke counts for common sensitive private data terms [63, 64] and corresponding suggested Cashtag alias.

6. Related Work

Previous works include both systems that secure against observation-based attacks and those that provide similar privacy protection over network channels.

6.1 Visual authentication protection

Prior work on protection against visual exposure is focused on securing the act of authentication. By far the earliest is the technique of Xing out or not printing entered passwords on login screens [15]. Most others can be generalized as an augmentation or replacement of password entry mechanisms.

Password managers: Password managers allow the user to select a predefined username and password pair from a list for entry into the login fields [14]. This also allows a user to use different passwords for different apps without the need to remember each individually.

Hardware-based authentication: Hardware-based authentication techniques utilize specialized USB dongles [17], audio jacks [18], short-range wireless communication [19], or Bluetooth connections [20] to connect to the authenticating machine. Therefore, a bystander cannot obtain the authentication information by observation.

Graphical passwords: Graphical passwords or Graphical User Authentication (GUA) [22] replace the alpha-numeric password with a series of images, shapes, and colors. The user needs to click a sequence of human faces [23], object sequences as part of a story [24], or specific regions within a given image [25].

Biometrics: Biometric authentication mechanisms augment password entry (something one knows) with a feature unique to one's personal biology (something one is). The most commonly used of these biometric identifiers includes contours of the fingerprints [26], iris and retinal configuration of the eye [27], and geometries of the face [28] and hand [29]. Behavioral characteristics such as keystroke latency [30], gait [31], and voice [32] can also be used.

Gesture-based authentication: Gesture-based authentication techniques allow the user to perform specific tap [33], multi-finger presses [34], or swipe sequences on-screen [35] to represent a password.

Cognitive challenges and obfuscation: Other techniques have attempted to make games of the authentication procedure [37]. These techniques utilize challenge-response questions and cognitive tasks to increase the difficulty of the login session [38]. Other techniques use obfuscation (e.g., the hiding of cursors [39], confusion matrices [40], and recognition [41]) rather than recall-based methods, to confuse onlookers.

Alternative sensory inputs: Some systems utilize audio direction [42] or tactile and haptic feedback from the vibration motors on devices [43] to provide the user with the appropriate cue for the necessary response. The user then responds with the passphrase corresponding to the cue via traditional input methods.

There are also systems that extend GUAs by requiring sequential graphical inputs and use mechanics, such as eye tracking, blinking and gaze-based interaction for the user to input the graphical sequence [44]. Systems have even demonstrated the capability of using brain waves for this task; a user may only need to think a specific thought to authenticate with a system [45]. These methods are also useful alternatives for authentication of people with visual or audio sensory disabilities [46].

6.2 Physical barriers and screen filters

Physical barriers can serve as a means of limiting the amount of screen exposure to bystanders. This can be as simple as office cubicles. However, they are clearly not suitable for the increasingly mobile modern workforce. Other solutions, such as the 3M Privacy Filter [67] approach the problem by limiting the field of view of the screen. This may serve to reduce exposure, but larger screens are still visible for a larger area and can be seen by unauthorized viewers directly behind the device.

The Lenovo Sun Visor [68] and Compubody Sock [69] reduce the screen visibility further by completely blocking out all non-direct visual exposure. However, this also blocks the user's own field of view, leaving them susceptible to external threats such as pick pocketing.

6.3 Wearable devices

Wearable headsets such as Google Glass [57] prevent screen exposure by moving the screen directly in front of the user's eyes. However, the current generation devices have limited application support. In addition, much of the user input is performed by audio cues, which translates the visual sensitive data leaks to audio ones.

Oculus Rift [65] and Samsung Galaxy Wearable [66] permit similar private screen viewing. However, they currently do not permit general-purpose computing. Additionally, like physical barriers, these devices block the user's field of view, increasing the vulnerability to external physical threats.

6.4 Digital communication channel protection

Many protocols and systems have been developed to handle other aspects of privacy-oriented attacks through the encryption of the digital communication channel. Transport Layer Security and Secure Sockets Layer can enhance security by providing session-based encryption [47]. Virtual Private Networks can be used to enhance security by offering point-to-point encryption to provide secure resources access across insecure network topologies [48]. Proxy servers [49] and onion routing protocols such as Tor [50], can add extra privacy by providing obfuscation of the location and anonymization of IP addresses.

Other solutions have been developed to enhance privacy of browsers. Do-not-track requests can be included in the HTTP headers to request that the web server or application disable its user and cross-site tracking mechanisms [51]. Many browser extensions and plug-ins exist to block ads [52], analytics, beacons, and other tracking mechanisms [53]. Other systems alert the user when specific sensitive elements are leaked [54]. They prevent the transmission of sensitive data without explicit user permission [55], and the cryptography secures access to sensitive data outside of trusted situations [16].

6.5 Compared to Cashtags

Despite the various mechanisms mentioned, the visual channel remains largely open. A limited number of tools are available to obfuscate sensitive data other than during the act of authentication. Other tools developed for data encryption are not originally designed for such purposes.

Password-based solutions and biometrics are effective in handling visual leaks during the act of authentication, but they cannot be generalized to handle other cases. No existing mechanism is in place to allow arbitrary data to be marked as sensitive. To our best knowledge, Cashtags is the only system that can protect general data from shoulder surfing.

7. Discussion and Limitations

Increased coverage: Cashtags widget-level text manipulation works for apps that use standard text-rendering methods. Should developers deviate from such standards, Cashtags would not capture such cases. Still, the additions required to incorporate these custom methods to work within Cashtags would be minimal if knowledge of the custom text display functions and parameters were provided.

Cashtags currently is optimized for coverage rather than performance. Thus, one future direction is to explore better text-processing methods and data structures.

Common names: Commonly occurring names can be problematic. Consider a user John Smith, with Cashtag aliases of his name: John→`$fname`, and Smith→`$lname`. Therefore, all on-screen instances of John are masked as `$fname`. If John opens his browser and Googles the name John Travolta, all returned search results would be displayed with on-screen representations as `$fname` Travolta. If an on-looker was able to observe the above search queries, and was aware of the operation of Cashtags, he or she might be able to derive the sensitive data from the context. This limitation is isolated to common phrases; numerical phrases would be less subject to this issue.

Data formatting: For data formatting and types, many cases are handled through transformations of text fields, including the removal of spaces and symbols, and capitalization mismatches. However, data that expands across `TextViews` is not recognized (e.g., input fields for a credit card split into parts rather than combined into a single field). Cashtags could handle this if each part of the credit-card number were added to the repository.

Handling business use cases: This paper presents Cashtags in light of protecting personal sensitive information. However, with more advanced cashtag mapping rules internally implemented via regular expression templates, we can extend our framework to handle business use cases. For example, for banking account apps, we can mask all of the dollar amounts to `$#`, with a fixed number of digits. A user can specify common preset amounts using cashtags (e.g., pay `$rent` to `$apt_acct`). For database apps, we can mask fields with specific template formats (e.g., phone numbers, identification numbers with certain prefixes). While such extension will require ways to specify app-specific rules, our core framework remains the same.

Generalization of approach: The Cashtags system was prototyped on Android. However, the general approach of screen rendering and user input interception can easily be generalized.

Human subject study: One aspect that is important to system usability is the frequency that sensitive data is entered or displayed. The actual utility of Cashtags is directly related to how frequently personally identifiable information is accessed and input by the user. Unfortunately, to our best knowledge statistics on the frequency of such accesses are not available. A future human subjects study of Cashtags can help determine this frequency of sensitive data access, as well as further evaluate system effectiveness and usability.

8. Conclusion

Cashtags is a first step toward protection against visual leaks of on-screen data. The system demonstrates that it is possible to perform most mobile computing tasks in public locations without exposing sensitive information. The evaluation of the system shows that this can be accomplished efficiently, with minimal perceived overhead. The app coverage test confirms that the system handles general-purpose tasks and maintains full functionality with nearly all tested common use cases. These results suggest that Cashtags will likely work on most other mobile apps, providing unified, device-wide protection against shoulder surfing.

9. Acknowledgements

We would like to thank our shepherd Tadayoshi Kohno, and anonymous reviews for their invaluable feedback. This work is sponsored by NSF CNS-1065127. Opinions, findings, and conclusions or recommendations expressed in this document do not necessarily reflect the views of the NSF, FSU, UCLA, or the U.S. government.

References

[1] Honan, Brian. “Visual Data Security White Paper”, July 2012. BH Consulting & European Association for Visual Data Security. http://www.visualdatasecurity.eu/wp-

content/uploads/2012/07/Visual-Data-Security-White-Paper.pdf. Retrieved 4/2014

[2] Thomson, Herbert H, PhD. "Visual Data Breach Risk Assessment Study." 2010. People Security Consulting Services, Commissioned by 3M. http://solutions.3m.com/3MContentRetrievalAPI/BlobServlet?assetId=1273672752407&assetType=MMM_Image&blobAttribute=ImageFile. Retrieved 4/2014

[3] Vikuiti Privacy Filters. "Shoulder Surfing Survey". 2007. Commissioned by 3M UK PLC. http://multimedia.3m.com/mws/mediawebserver?6666660Zjcf6lVs6EVs66SlZpCOrrrrQ-. Retrieved 4/2014

[4] European Association for Visual Data Security. "Visual Data Security", March 2013. http://www.visualdatasecurity.eu/wp-content/uploads/2013/03/Secure-Briefing-2013-UK.pdf. Retrieved 4/2014

[5] International Data Corporation. "Worldwide Mobile Worker Population 2011-2015 Forecast." http://cdn.idc.asia/files/5a8911ab-4c6d-47b3-8a04-01147c3ce06d.pdf. Retrieved 4/2014

[6] Good Technology. "Americans are Working More, but on their Own Schedule", July 2012. http://www1.good.com/about/press-releases/161009045.html. Retrieved 4/2014

[7] Nokia, USA. "Nokia Lumia 1020", http://www.nokia.com/us-en/phones/phone/lumia1020/. Retrieved 4/2014

[8] NPD DisplaySearch. "Wide Viewing Angle LCD Technologies Gain Share Due to Tablet PC Demand". January 2012. http://www.displaysearch.com/cps/rde/xchg/displaysearch/hs.xsl/120119_wide_viewing_angle_lcd_technologies_gain_share_due_to_tablet_pc_demand.asp. Retrieved 4/2014

[9] Pillai, Geetha. "Caught on Camera: You are Filmed on CCTV 300 Times a Day in London", International Business Times, March 2012. http://www.ibtimes.co.uk/britain-cctv-camera-surveillance-watch-london-big-312382. Retrieved 4/2014

[10] Loh Zhi Chang and Steven Zhou ZhiYing. "Robust pre-processing techniques for OCR applications on mobile devices", In Proceedings of the 6th International Conference on Mobile Technology, Application & Systems (Mobility '09). ACM, New York, NY, USA, Article 60 , 4 pages. DOI=10.1145/1710035.1710095 http://doi.acm.org/10.1145/1710035.1710095

[11] Owen, Glen. "The zzzzivil servant who fell asleep on the train with laptop secrets in full view", November 2008. http://www.dailymail.co.uk/news/article-1082375/The-zzzzivil-servant-fell-asleep-train-laptop-secrets-view.html. Retrieved 4/2014

[12] Penn, Ivan. "Simple fix to bank security breach: Close the blinds", Tampa Bay Times. December 2010. http://www.tampabay.com/features/consumer/simple-fix-to-bank-security-breach-close-the-blinds/1139356. Retrieved 4/2014

[13] Davies, Caroline. "Prince William photos slip-up forces MoD to change passwords", The Guardian, November 2102. http://www.theguardian.com/uk/2012/nov/20/prince-william-photos-mod-passwords. Retrieved 4/2014

[14] J. Alex Halderman, Brent Waters, and Edward W. Felten. "A convenient method for securely managing passwords", In Proceedings of the 14th international conference on World Wide Web (WWW '05). ACM, New York, NY, USA, 471-479. DOI=10.1145/1060745.1060815 http://doi.acm.org/10.1145/1060745.1060815

[15] CTSS Programmers Guide, 2nd Ed., MIT Press, 1965

[16] C. Fleming, P. Peterson, E. Kline and P. Reiher, "Data Tethers: Preventing Information Leakage by Enforcing Environmental Data Access Policies," in International Conference on Communications (ICC), 2012.

[17] Yubico, Inc. "About YubiKey", 2014. http://www.yubico.com/about. Retrieved 4/2014

[18] Square, Inc. "About Square", 2014. https://squareup.com/news. Retrieved 4/2014

[19] Google, Inc. "Google NFC YubiKey Neo", September 2013. http://online.wsj.com/news/articles/SB10001424127887323585604579008620509295960

[20] Wayne Jansen and Vlad Korolev. "A Location-Based Mechanism for Mobile Device Security", In Proceedings of the 2009 WRI World Congress on Computer Science and Information Engineering (CSIE '09), Vol. 1. IEEE Computer Society, Washington, DC, USA, 99-104. DOI=10.1109/CSIE.2009.719 http://dx.doi.org/10.1109/CSIE.2009.719

[21] Google, Inc. Tesseract-OCR. https://code.google.com/p/tesseract-ocr/

[22] Blonder, Greg E. "Graphical Passwords". United States patent 5559961, Lucent Technologies, Inc. 1996.

[23] Passfaces Corporation. "The Science Behind Passfaces", June 2004. http://www.realuser.com/published/ScienceBehindPassfaces.pdf

[24] Darren Davis, Fabian Monrose, and Michael K. Reiter. "On user choice in graphical password schemes", In Proceedings of the 13th conference on USENIX Security Symposium - Volume 13 (SSYM'04), Vol. 13. USENIX Association, Berkeley, CA, USA, 11-11.

[25] Susan Wiedenbeck, Jim Waters, Jean-Camille Birget, Alex Brodskiy, and Nasir Memon. "PassPoints: design and longitudinal evaluation of a graphical password system" International Journal of Human-Computer Studies. 63, 1-2 (July 2005), 102-127. DOI=10.1016/j.ijhcs.2005.04.010 http://dx.doi.org/10.1016/j.ijhcs.2005.04.010

[26] Jain, A.K.; Hong, L.; Pankanti, S.; Bolle, R., "An identity-authentication system using fingerprints," Proceedings of the IEEE, vol.85, no.9, pp.1365, 1388, Sep 1997. doi: 10.1109/5.628674

[27] J. Daugman. "How iris recognition works", IEEE Transactions on Circuits and Systems for Video Technology. 14, 1 (January 2004), 21-30. DOI=10.1109/TCSVT.2003.818350 http://dx.doi.org/10.1109/TCSVT.2003.818350

[28] Anil K. Jain, Arun Ross, Sharath Pankanti. "A Prototype Hand Geometry-based Verifcation System", In Proceedings of 2nd International Conference on Audio- and Video-based Biometric Person Authentication (AVBPA), Washington D.C., pp.166-171, March 22-24, 1999.

[29] W. Zhao, R. Chellappa, P. J. Phillips, and A. Rosenfeld. "Face recognition: A literature survey". ACM Computing Surveys. 35, 4 (December 2003), 399-458. DOI=10.1145/954339.954342 http://doi.acm.org/10.1145/954339.954342

[30] Rick Joyce and Gopal Gupta. "Identity authentication based on keystroke latencies", Communications of the ACM ,33, 2 (February 1990), 168-176. DOI=10.1145/75577.75582 http://doi.acm.org/10.1145/75577.75582

[31] Davrondzhon Gafurov, Kirsi Helkala, Torkjel Søndrol. "Biometric Gait Authentication Using Accelerometer Sensor", Journal of Computers, Vol. 1, No. 7, October 2006.

[32] Roberto Brunelli and Daniele Falavigna. "Person Identification Using Multiple Cues", IEEE Transactions on Pattern Analysis and Machine Intelligence. 17, 10 (October 1995), 955-966. DOI=10.1109/34.464560 http://dx.doi.org/10.1109/34.464560

[33] Alexander De Luca, Alina Hang, Frederik Brudy, Christian Lindner, and Heinrich Hussmann. "Touch me once and I know it's you!: implicit authentication based on touch screen patterns", In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '12). ACM, New York, NY, USA, 987-996. DOI=10.1145/2207676.2208544 http://doi.acm.org/10.1145/2207676.2208544

[34] Ioannis Leftheriotis. "User authentication in a multi-touch surface: a chord password system" In CHI '13 Extended Abstracts on Human Factors in Computing Systems (CHI EA '13). ACM, New York, NY, USA, 1725-1730. DOI=10.1145/2468356.2468665 http://doi.acm.org/10.1145/2468356.2468665

[35] Ming Ki Chong, Gary Marsden, and Hans Gellersen. "GesturePIN: using discrete gestures for associating mobile devices", In Proceedings of the 12th international conference on Human computer interaction with mobile devices and services (MobileHCI '10). ACM, New York, NY, USA, 261-264. DOI=10.1145/1851600.1851644 http://doi.acm.org/10.1145/1851600.1851644

[36] Android Developers, Uiautomator. https://developer.android.com/tools/help/uiautomator/index.html

[37] Volker Roth, Kai Richter, and Rene Freidinger. "A PIN-entry method resilient against shoulder surfing", In Proceedings of the 11th ACM conference on Computer and communications security (CCS '04). ACM, New York, NY, USA, 236-245. DOI=10.1145/1030083.1030116 http://doi.acm.org/10.1145/1030083.1030116

[38] T. Perkovic, M. Cagalj, and N. Rakic. "SSSL: shoulder surfing safe login", In Proceedings of the 17th international conference on Software, Telecommunications and Computer Networks (SoftCOM'09). IEEE Press, Piscataway, NJ, USA, 270-275.

[39] Alice Boit, Thomas Geimer, and Jorn Loviscach. "A random cursor matrix to hide graphical password input", In SIGGRAPH '09: Posters (SIGGRAPH '09). ACM, New York, NY, USA, Article 41, 1 pages. DOI=10.1145/1599301.1599342 http://doi.acm.org/10.1145/1599301.1599342

[40] Rohit Ashok Khot, Ponnurangam Kumaraguru, and Kannan Srinathan. "WYSWYE: shoulder surfing defense for recognition based graphical passwords", In Proceedings of the 24th Australian Computer-Human Interaction Conference (OzCHI '12), ACM, New York, NY, USA, 285-294. DOI=10.1145/2414536.2414584 http://doi.acm.org/10.1145/2414536.2414584

[41] Rachna Dhamija and Adrian Perrig. "Deja; Vu: a user study using images for authentication", In Proceedings of the 9th conference on USENIX Security Symposium - Volume 9 (SSYM'00), Vol. 9. USENIX Association, Berkeley, CA, USA, 4-4.

[42] Mary Brown and Felicia R. Doswell. "Using passtones instead of passwords", In Proceedings of the 48th Annual Southeast Regional Conference (ACM SE '10). ACM, New York, NY, USA, Article 82, 5 pages. DOI=10.1145/1900008.1900119 http://doi.acm.org/10.1145/1900008.1900119

[43] Andrea Bianchi, Ian Oakley, and Dong Soo Kwon. "The secure haptic keypad: a tactile password system", In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10). ACM, New York, NY, USA, 1089-1092. DOI=10.1145/1753326.1753488 http://doi.acm.org/10.1145/1753326.1753488

[44] Alain Forget, Sonia Chiasson, and Robert Biddle. "Shoulder-surfing resistance with eye-gaze entry in cued-recall graphical passwords", In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10). ACM, New York, NY, USA, 1107-1110. DOI=10.1145/1753326.1753491 http://doi.acm.org/10.1145/1753326.1753491

[45] Julie Thorpe, P. C. van Oorschot, and Anil Somayaji. "Pass-thoughts: authenticating with our minds", In Proceedings of the 2005 workshop on New security paradigms (NSPW '05). ACM, New York, NY, USA, 45-56. DOI=10.1145/1146269.1146282 http://doi.acm.org/10.1145/1146269.1146282

[46] Nitesh Saxena and James H. Watt. "Authentication technologies for the blind or visually impaired", In Proceedings of the 4th USENIX conference on Hot topics in security (HotSec'09). USENIX Association, Berkeley, CA, USA, 7-7.

[47] T. Dierks, E. Rescorla. "The Transport Layer Security (TLS) Protocol, Version 1.2", August 2008.

[48] Mason, Andrew G. "Cisco Secure Virtual Private Network". Cisco Press, 2002, p. 7.

[49] Marc Shapiro. "Structure and Encapsulation in Distributed Systems: the Proxy Principle", In Proceedings of the 6th IEEE International Conference on Distributed Computing Systems (ICDCS), Cambridge MA (USA), May 1986.

[50] Roger Dingledine, Nick Mathewson, and Paul Syverson. "Tor: the second-generation onion router", In Proceedings of the 13th conference on USENIX Security Symposium (SSYM'04), Vol. 13. 2004 USENIX Association, Berkeley, CA, USA, 21-21.

[51] Do Not Track. "Do Not Track - Universal Web Tracking Opt Out", http://donottrack.us. Retrieved 4/2014

[52] Adblock Plus. "Adblock Plus : About", https://adblockplus.org/en/about. Retrieved 4/2014

[53] Evidon, Inc. "About Ghostery", https://www.ghostery.com/en/about. Retrieved 4/2014

[54] Braden Kowitz and Lorrie Cranor. "Peripheral privacy notifications for wireless networks", In Proceedings of the 2005 ACM workshop on Privacy in the electronic society (WPES '05). ACM, New York, NY, USA, 90-96. DOI=10.1145/1102199.1102217 http://doi.acm.org/10.1145/1102199.1102217

[55] Sunny Consolvo, Jaeyeon Jung, Ben Greenstein, Pauline Powledge, Gabriel Maganis, and Daniel Avrahami. "The Wi-Fi privacy ticker: improving awareness & control of personal information exposure on Wi-Fi", In Proceedings of the 12th ACM international conference on Ubiquitous computing (Ubicomp '10). ACM, New York, NY, USA, 321-330. DOI=10.1145/1864349.1864398 http://doi.acm.org/10.1145/1864349.1864398.

[56] Simon Khalaf. "Apps Solidify Leadership Six Years into Mobile Revolution," Flurry, http://www.flurry.com/bid/109749/Apps-Solidify-Leadership-Six-Years-into-the-Mobile-Revolution, 2014.

[57] Google, Inc. Google Glass. http://www.google.com/glass/start/

[58] Rahul Raguram, Andrew M. White, Dibyendusekhar Goswami, Fabian Monrose, and Jan-Michael Frahm. 2011. iSpy: automatic reconstruction of typed input from compromising reflections. In Proceedings of the 18th ACM conference on Computer and communications security (CCS '11). ACM, New York, NY, USA, 527-536. DOI=10.1145/2046707.2046769 http://doi.acm.org/10.1145/2046707.2046769

[59] Erika McCallister, Tim Grance, Karen Scarfone. Guide to Protecting the Confidentiality of Personally Identifiable Information (SP 800-122). National Institute of Standards and Technology, http://csrc.nist.gov/publications/nistpubs/800-122/sp800-122.pdf

[60] Android Developers, Android Debug Bridge. https://developer.android.com/tools/help/adb.html

[61] Kino. The Kino Project. http://sourceforge.net/projects/pskino/

[62] Screen Concealer. http://screenconcealer.com/

[63] U.S. Census Bureau, Population Division, Population Analysis & Evaluation Staff. http://factfinder.census.gov/

[64] Jonathan Lampe. "Beyond Password Length and Complexity", The Infosec Institute. January, 2014. http://resources.infosecinstitute.com/beyond-password-length-complexity/

[65] Oculus Rift - Virtual Reality Headset for 3D Gaming. Oculus VR, LLC. https://www.oculus.com/

[66] Samsung Gear VR. Samsung Electronics Co., Ltd. https://www.samsung.com/global/microsite/gearvr/

[67] 3M Privacy and Screen Protectors. The 3M Company. http://solutions.3m.com/wps/portal/3M/en_EU/3MScreens_EU/Home/PrivacyFilters/

[68] Lenovo Sun Visor. Lenovo Group Ltd. http://blog.lenovo.com/en/blog/see-in-the-light

[69] Becky Stern. Compubody Socks or Knitted Body-Technology Interfaces. Sternlab. http://sternlab.org/?p=90

[70] OpenSignal, Inc. "Android Fragmentation Visualized", August, 2014. http://opensignal.com/reports/2014/android-fragmentation/

[71] Sonia Chiasson, P. C. van Oorschot, and Robert Biddle. "A usability study and critique of two password managers," In Proceedings of the 15th conference on USENIX Security Symposium (USENIX-SS'06), Vol. 15, 2006. USENIX Association, Berkeley, CA, USA, p. 1.

SUPOR: Precise and Scalable Sensitive User Input Detection for Android Apps

Jianjun Huang[1], Zhichun Li[2], Xusheng Xiao[2], Zhenyu Wu[2], Kangjie Lu[3], Xiangyu Zhang[1], and Guofei Jiang[2]

[1]Department of Computer Science, Purdue University
[2]NEC Labs America
[3]School of Computer Science, Georgia Institute of Technology
huang427@purdue.edu, {*zhichun, xsxiao, adamwu*}*@nec-labs.com*, *kjlu@gatech.edu*
xyzhang@cs.purdue.edu, *gfj@nec-labs.com*

Abstract

While smartphones and mobile apps have been an essential part of our lives, privacy is a serious concern. Previous mobile privacy related research efforts have largely focused on predefined known sources managed by smartphones. Sensitive user inputs through UI (User Interface), another information source that may contain a lot of sensitive information, have been mostly neglected.

In this paper, we examine the possibility of scalably detecting sensitive user inputs from mobile apps. In particular, we design and implement SUPOR, a novel static analysis tool that automatically examines the UIs to identify sensitive user inputs containing critical user data, such as user credentials, finance, and medical data. SUPOR enables existing privacy analysis approaches to be applied on sensitive user inputs as well. To demonstrate the usefulness of SUPOR, we build a system that detects privacy disclosures of sensitive user inputs by combining SUPOR with off-the-shelf static taint analysis We apply the system to 16,000 popular Android apps, and conduct a measurement study on the privacy disclosures. SUPOR achieves an average precision of 97.3% and an average recall of 97.3% for sensitive user input identification. SUPOR finds 355 apps with privacy disclosures and the false positive rate is 8.7%. We discover interesting cases related to national ID, username/password, credit card and health information.

1 Introduction

Smartphones have become the dominant kind of end-user devices with more units sold than traditional PCs. With the ever-increasing number of apps, smartphones are becoming capable of handling all kinds of needs from users, and gain more and more access to sensitive and private personal data. Despite the capabilities to meet users' needs, data privacy in smartphones becomes a major concern.

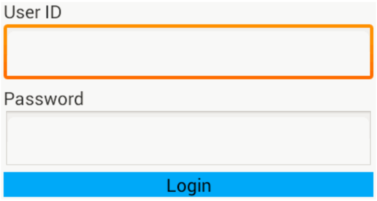


Figure 1: Example sensitive user inputs.

Previous research on smartphone privacy protection primarily focuses on sensitive data managed by the phone OS and framework APIs, such as device identifiers (phone number, IMEI, etc.), location, contact, calendar, browser state, most of which are permission protected. Although these data sources are very important, they do not cover all sensitive data related to users' privacy. A major type of sensitive data that has been largely neglected are the *sensitive user inputs*, which refers to the sensitive information entered by users via the User Interface (UI). Many apps today acquire sensitive credentials, financial, health, and medical information from users through the UI. Therefore, to protect and respect users' privacy, apps must handle sensitive user inputs in a secure manner that matches with users' trust and expectations.

Figure 1 shows an example interface an app uses to acquire users' login credentials via input fields rendered in the UI. When users click the button "Login", the app use the user ID and password to authenticate with a remote service. As the developers may be unaware of the potential risk on the disclosures of such sensitive information, the login credentials are sent in plain text over an insecure channel (HTTP), which inadvertently compromises users' privacy.

In this paper, we propose SUPOR (Sensitive User inPut detectOR), a static mobile app analysis tool for detecting sensitive user inputs and identifying their associated variables in the app code as sensitive information sources. To the best of our knowledge, we are the first to study scalable detection of sensitive user inputs on smartphone platforms.

Previously, there are many existing research efforts [9, 10, 12, 23, 24, 30, 31, 34, 40] on studying the privacy related topics on predefined sensitive data sources on the phone. Our approach enables those existing efforts to be applied to sensitive user inputs as well. For example, with proper static or dynamic taint analysis, one can track the privacy disclosures of sensitive user inputs to different sinks. With static program analysis, one can also identify the vulnerabilities in the apps that may unintentionally disclosure such sensitive user inputs to public or to the attacker controlled output. One could also study how sensitive user inputs propagate to third-party advertisement libraries, etc.

In this paper, to demonstrate the usefulness of our approach, we combine SUPOR with off-the-shelf static taint analysis to detect privacy disclosures of sensitive user inputs.

The major challenges of identifying sensitive user inputs are the following:

(i) How to systematically discover the input fields from an app's UI?

(ii) How to identify which input fields are sensitive?

(iii) How to associate the sensitive input fields to the corresponding variables in the apps that store their values?

In order to detect sensitive user inputs scalably, static UI analysis is much appealing, because it is very difficult to generate test inputs to trigger all the UI screens in an app in a scalable way. For example, an app might require login, which is difficult for tools to generate desirable inputs and existing approaches usually require human intervention [26]. On the other hand, it is also extremely challenging to launch static analysis to answer the aforementioned three questions for general desktop applications.

To this end, we have studied major mobile OSes, such as Android, iOS and Windows Phone systems, and made a few important observations. Then, we implement SUPOR for Android since it is most popular.

First, we find all these mobile OSes provide a standard rapid UI development kit as part of the development framework, and most apps use such a homogeneous UI framework to develop apps. Such UI framework usually leverages a declarative language, such as XML based layout languages, to describe the UI layout, which enables us to statically discover the input fields on the UI.

Second, in order to identify which input fields are sensitive, we have to be able to render the UI, because the rendered UI screens contain important texts as hints that guide users to enter their inputs, which can be used to identify whether the inputs are sensitive. For instance, in Figure 1, the text "User ID" describes the nature of the first input field. Statically rendering UI screens is generally very hard for arbitrary desktop applications. However, with help of WYSIWYG (What You See is What You Get) layout editing feature from the rapid UI development kits of mobile OSes, we are able to statically render the UI for most mobile apps in order to associate the descriptive text labels with the corresponding input fields. Furthermore, due to the relatively small screen size of smartphones, most text labels are concise. As such, current NLP (Natural Language processing) techniques can achieve high accuracy on identifying sensitive terms.

Third, all mobile OSes provide APIs to load the UI layouts made by rapid UI development kits and to bind with the app code. Such a binding mechanism provides us opportunities to infer the relationship between the sensitive input fields from UI layouts to the variables in the app code that store their values.

Our work makes three major contributions:

First, we devise a UI sensitiveness analysis that identifies the input fields that may accept sensitive information by leveraging UI rendering, geometrical layout analysis and NLP techniques. We modify the static rendering engine from the ADT (Android Developer Tools), so that the static rendering can be done with an APK binary instead of source code, and accurately identify the coordinates of text labels and input fields. Then, based on the insight that users typically read the text label physically close to the input field in the screen for understanding the purpose of the input field, we design an algorithm to find the optimal descriptive text label for each input field. We further leverage NLP (nature language processing) techniques [11, 22, 36] to select and map popular keywords extracted from the UIs of a massive number of apps to important sensitive categories, and use these keywords to classify the sensitive text labels and identify sensitive input fields. Our evaluation shows that SUPOR achieves an average precision of 97.3% and an average recall of 97.3% for sensitive user inputs detection.

Second, we design a context-sensitive approach to associate sensitive UI input fields to the corresponding variables in the app code. Instances of sensitive input widgets in the app code can be located using our UI analysis results in a context-insensitive fashion (i.e. based on widget IDs). We further reduce false positives by adding context-sensitivity, i.e. we leverage backward slicing and identify each input widget's residing layout by tracing back to the closest layout loading function. Only if both widget and layout identifiers match with the sensitive input field in the XML layout, we consider the widget instance is associated with the sensitive input field.

Finally, we implement a privacy disclosure detection system based on SUPOR and static taint analysis, and apply the system to 16,000 popular free Android apps collected from the Official Android Market (Google Play). The system can process 11.1 apps per minute on an eight-server cluster. Among all these apps, 355 apps are detected with sensitive user input disclosures. Our manual validation on these suspicious apps shows an overall detection accuracy of 91.3%. In addition, we conduct detailed case studies on the apps we discovered, and show interesting cases of unsafe disclosures of users' national IDs, credentials, credit card and health related information.

2 Background and Motivation Example

In this section, we provide background on sensitive user input identification.

2.1 Necessary Support for Static Sensitive User Input Identification

Modern mobile OSes, such as Android, iOS and Windows Phone system, provide frameworks and tools for rapid UI design. They usually provide a large collection of standard UI widgets, and different layouts to compose the widgets together. They also provide a declarative language, such as XML, to let the developer describe their UI designs, and further provide GUI support for WYSIWYG UI design tools. In order to design a static analysis tool for sensitive user input identification, we need four basic supporting features. The rapid UI development design in modern mobile OSes makes it feasible to achieve such features.

A: statically identify the input fields and text labels;
B: statically identify the attributes of input fields;
C: statically render the UI layout without launching the app;
D: statically map the input fields defined in the UI layouts to the app code.

These four features are necessary to statically identify the sensitive input fields on UIs. In order to infer the semantic meaning of an input field and decide whether it is sensitive, we need (*i*) the attributes of the input field; (*ii*) the surrounding descriptive text labels on the UI. Some attributes of the input fields can help us quickly understand its semantics and sensitiveness. For example, if the input type is password, we know this is a password-like input field. However, in many cases, the attributes alone are not enough to decide the semantics and sensitiveness of the input fields. In those cases, we have to rely on UI analysis. A well-designed app has to allow the user to easily identify the relevant texts for a particular input field and provide appropriate inputs based on his understanding of the meaning of texts. Based on the above observation, we need Feature C to render the UI and obtain the coordinates of input fields and text labels, so that we can associate them and further reason about the sensitiveness of input fields. Once we identify the sensitive input fields, we have to find the variable in the app code used to store the values of the input field for further analysis.

Table 1: UI features in different mobile OSes

	Android	iOS	Windows Phone
Layout format	XML	NIB / XIB / Storyboard	XAML/HTML
Static UI render	ADT	Xcode	Visual Studio
APIs map widgets to code	Yes	Yes	Yes

```
<LinearLayout android:orientation="vertical">
  <TextView android:text="@string/tip_uid" />
  <EditText android:id="@+id/uid" />
  <TextView android:text="@string/tip_pwd" />
  <EditText android:id="@+id/pwd"
       android:inputType="textPassword" />
  <Button android:id="@+id/login"
       android:text="@string/tip_login"/>
</LinearLayout>
```

Figure 2: Simplified layout file *login_activity.xml*.

We have studied Android, iOS and Windows Phone systems. As shown in Table 1, all mobile OSes provide standard formats for storing app UI layouts that we can use to achieve features A and B. All of them have IDEs that can statically render UI layouts for the WYSIWYG UI design. If we reuse this functionality we can achieve static rendering (feature C). Furthermore, all of them provide APIs for developers to map the widgets in layouts to the variables in the app code that hold the user inputs. Combined with static program analysis to understand the mapping, we will be able to achieve feature D.

2.2 Android UI Rendering

For proof of concept, the current SUPOR is designed for the Android platform. An Android app usually consists of multiple activities. Each activity provides a window to draw a UI. A UI is defined by a layout, which specifies the dimension, spacing, and placement of the content within the window. The layout consists of various interactive UI widgets (*e.g.,* input fields and buttons) as well as layout models (*e.g.,* linear or relative layout) that describe how to arrange UI widgets.

At run time, when a layout file is loaded, the Android framework parses the layout file and determines how to render the UI widgets in the window by checking the layout models and the relevant attributes of the UI widgets. At the mean time, all UI widgets in the layout are instantiated and then can be referenced in the code.

An example layout in XML is presented in Figure 2 and the code snippet of the corresponding activity is

```
1  public class LoginActivity extends Activity
       implements View.OnClickListener {
2   private EditText txtUid, txtPwd;
3   private Button btnReset;
4   protected void onCreate(Bundle bundle) {
5    super.onCreate(bundle);
6    setContentView(R.layout.login_activity);
7    txtUid = (EditText) findViewById(R.id.uid);
8    txtPwd = (EditText) findViewById(R.id.pwd);
9    btnLogin = (Button) findViewById(R.id.login);
10   btnLogin.setOnClickListener(this);
11  }
12  public void onClick(View view) {
13   String uid = txtUid.getText().toString();
14   String pwd = txtPwd.getText().toString();
15   String url = "http://www.plxx.com/Users/" +
         "login?uid=" + uid + "&pwd=" + pwd;
16   HttpClient c = new DefaultHttpClient();
17   HttpGet g = new HttpGet(url);
18   Object o = c.execute(g, new
         BasicResponseHandler());
19   // following operations are omitted
20  }
21 }
```

Figure 3: Simplified Activity example.

shown in Figure 3. This layout includes five UI widgets: two text labels (`TextView`), two input fields (`EditText`) and a button. They are aligned vertically based on the `LinearLayout` at Line 1. The first text label shows “User ID” based on the attribute `android:text="@string/tip_uid"`, which indicates a string stored as a resource with the ID `tip_uid`. The *type* attribute of the second input field is `android:inputType="textPassword"`, indicating that it is designed for accepting a password, which conceals the input after the users enter it. Instead of explicitly placing text labels as in Figure 2, some developers decorate an input field with a *hint* attribute, which specifies a message that will be displayed when the input is empty. For instance, developers may choose to display “User ID” and “Password” inside the corresponding input fields using the *hint* attribute.

Figure 1 shows the rendered UI for the layout in Figure 2. The layout including all the inner widgets is loaded into the screen by calling `setContentView()` at Line 6 in Figure 3. The argument of `setContentView()` specifies the reference ID of the layout resource. Similarly, a runtime instance of a widget can also be located through a `findViewById()` call with the appropriate reference ID. For example, the reference ID `R.id.uid` is used to obtain a runtime instance of the input field at Line 7 in Figure 3.

2.3 UI Sensitiveness Analysis

Existing techniques usually consider permission protected framework APIs as the predefined sensitive data sources. However, generic framework APIs, such as `getText()`, can also obtain sensitive data from the user inputs. To precisely detect these sensitive sources, we need to determine which GUI input widgets are sensitive.

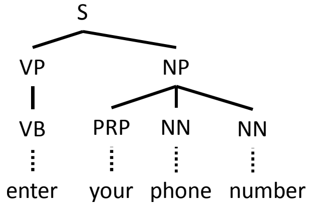


Figure 4: Parse tree of an example sentence.

Two kinds of information are useful for this purpose. First, certain attributes of the widgets can be a good indicator about whether the input is sensitive. Using the `inputType` attribute with a value “textPassword”, we can directly identify password fields. However, not all sensitive input fields use this attribute value. The hint attributes also may contain useful descriptive texts that may indicate the sensitiveness of the input fields.

Besides attributes of UI widgets, we observe that nearby text labels rendered in the UI also provide indication about the sensitiveness of the widgets. For example, a user can easily understand he is typing a user ID and a password when he sees the UI in Figure 1 because the text labels state what the input fields accept. In other words, these text labels explain the purposes of the UI widgets, and guide users to provide their inputs. Based on these observations, we propose to leverage the outcome of UI rendering to build a precise model of the UI and analyze the text labels and hints associated with the widgets to determine their sensitiveness.

The major task of analyzing text labels is to analyze the text labels’ texts, which are written in natural language. As smartphones have relatively small screens, the texts shown in the UI are usually very concise and straightforward to understand. For example, these texts typically are just noun/verb phrases or short sentences (such as the ones shown in Figure 1), and tend to directly state the purposes for the corresponding GUI widgets. Since there is no need to analyze paragraphs or even long sentences, we propose a light-weight keyword-based algorithm that checks whether text labels contain any sensitive keyword to determine the sensitiveness of the corresponding GUI widgets.

2.4 Natural Language Processing

With recent research advances in the area of natural language processing (NLP), NLP techniques have been shown to be fairly accurate in highlighting grammatical structure of a natural language sentence. Recent work has also shown promising results in using NLP techniques for analyzing Android descriptions [13, 30]. In our work, we adapt NLP techniques to extract nouns and noun phrases from the texts collected from popular apps, and identify keywords from the extracted nouns and noun phrases. We next briefly introduce the key NLP techniques used in this work.

Our approach uses **Parts Of Speech (POS) Tagging** [22,36] to identify interesting words, such as nouns, and filter unrelated words, such as conjunctives like "and/or". The technique tags a word in a sentence as corresponding to a particular part of speech (such as identifying nouns, verbs, and adjectives), based on both its definition and its relationship with adjacent and related words in a phrase, sentence, or paragraph. The state-of-the-art approaches can achieve around 97% [36] accuracy in assigning POS tags for words in well-written news articles.

Our approach uses **Phrase and Clause Parsing** to identify phrases for further inspection. Phrase and clause parsing divides a sentence into a constituent set of words (*i.e.,* phrases and clauses). These phrases and clauses logically belong together, *e.g.,* Noun Phrases and Verb Phrases. The state-of-the-art approaches can achieve around 90% [36] accuracy in identifying phrases and clauses over well-written news articles.

Our approach uses **Syntactic parsing** [21], combined with the above two techniques, to generate a parse-tree structure for a sentence, and traverse the parse tree to identify interesting phrases such as noun phrases. The parse tree of a sentence shows the hierarchical view of the syntax structure for the sentence. Figure 4 shows the parse tree for an example sentence "enter your phone number". The root node of the tree is the sentence node with the label *S*. The interior nodes of the parse tree are labeled by non-terminal categories of the grammar (*e.g.,* verb phrases *VP* and noun phrases *NP*), while the leaf nodes are labeled by terminal categories (*e.g.,* pronouns *PRP*, nouns *NN* and verbs *VB*). The tree structure provides a basis for other tasks within NLP such as question and answer, information extraction, and translation. The state of the art parsers have an F1 score of 90.4% [37].

3 Design of SUPOR

In this section, we first present our threat model, followed by an overview of SUPOR. Then, we describe each component of SUPOR in details.

3.1 Threat Model

We position SUPOR as a static UI analysis tool for detecting sensitive user inputs. Instead of focusing on malicious apps that deliberately evade detection, SUPOR is designed for efficient and scalable screening of a large number of apps. Most of the apps in the app markets are legitimate, whose developers try to monetize by gaining user popularity, even though some of them might be a little bit aggressive on exploiting user privacy for revenue. Malware can be detected by existing works [5, 15, 39], which is out of scope of this paper.

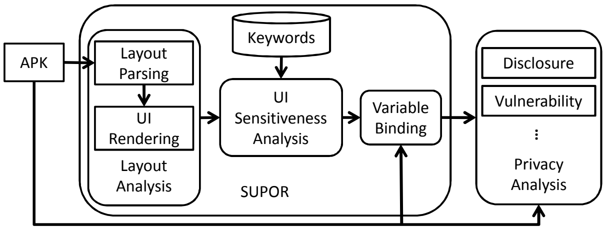


Figure 5: Overview of SUPOR.

Though the developers sometimes dynamically generate UI elements in the code other than defining the UI elements via layout files, we focus on identifying sensitive user inputs statically defined in layout files in this work.

3.2 Overview

Figure 5 shows the workflow of SUPOR. SUPOR consists of three major components: *Layout Analysis*, *UI Sensitiveness Analysis*, and *Variable Binding*. The layout analysis component accepts an APK file of an app, parses the layout files inside the APK file, and renders the layout files containing input fields. Based on the outcome of UI rendering, the UI sensitiveness analysis component associates text labels to the input fields, and determines the sensitiveness of the input fields by checking the texts in the text labels against a predefined sensitive keyword dataset (Section 3.6). The variable binding component then searches the code to identify the variables that store the values of the sensitive input fields. With variable binding, existing research efforts in studying the privacy related topics on predefined well-known sensitive data sources can be applied to sensitive user inputs. For example, one can use taint analysis to detect disclosures of sensitive user inputs or other privacy analysis to analyze vulnerabilities of sensitive user inputs in the apps. Next we describe each component in detail.

3.3 Layout Analysis

The goal of the layout analysis component is to render the UIs of an Android app, and extract the information of input fields: types, hints, and absolute coordinates, which are later used for the UI sensitiveness analysis.

As we discussed in Section 2.3, if we cannot determine the sensitiveness of an input field based on its type and hint, we need to find a text label that describes the purpose of the input field. From the *user's perspective*, the text label that describes the purpose of an input field must be *physically close to the input field* in the screen; otherwise the user may correlate the text label with other input fields and provide inappropriate inputs. Based on this insight, the layout analysis component renders the UIs as if the UIs are rendered in production runs, mimicking how users look at the UIs. Based on the rendered

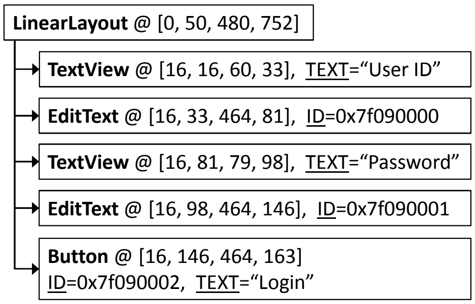

LinearLayout @ [0, 50, 480, 752]
TextView @ [16, 16, 60, 33], TEXT="User ID"
EditText @ [16, 33, 464, 81], ID=0x7f090000
TextView @ [16, 81, 79, 98], TEXT="Password"
EditText @ [16, 98, 464, 146], ID=0x7f090001
Button @ [16, 146, 464, 163]
ID=0x7f090002, TEXT="Login"

Figure 6: UI model for Figure 1 on 480x800 screen. Only the ID, relative coordinates and text of the widgets are presented here.

UIs, the distances between text labels and input fields are computed, and these distances are used later to find the best descriptive text labels for each input field. We next describe the two major steps of the layout analysis component.

The first step is to identify which layout files contain input fields by parsing the layout files in the APK of an Android app. In this work, we focus on input fields of the type `EditText` and all possible sub-types, including custom widgets in the apps. Each input field represents a potential sensitive source. However, according to our previous discussion, the sensitiveness cannot be easily determined by analyzing only the layout files. Thus, all the files containing input fields are used in the second step for UI rendering.

The second step is to obtain the coordinate information of the input fields by rendering the layout files. Using the rapid UI development kit provided by Android, the layout analysis component can effectively render standard UI widgets. For custom widgets that require more complex rendering, the layout analysis component renders them by providing the closest library superclass to obtain the best result. After rendering a layout file, the layout analysis component obtains a UI model, which is a tree-structure model where the nodes are UI widgets and the edges describe the parent-child relationship between UI widgets. Figure 6 shows the UI model obtained by rendering the layout file in Figure 2. For each rendered UI widget, the coordinates are relative to its parent container widget. Such relative coordinates cannot be directly used for measuring the distances between two UI widgets, and thus SUPOR converts the relative coordinates to absolute coordinates with regards to the screen size.

Coordinate Conversion. SUPOR computes the absolute coordinates of each UI widget level by level, starting with the root container widget. For example, in Figure 6, the root container widget is a `LinearLayout`, and its coordinates are (0, 50, 480, 752), representing the left, top, right, and bottom corners. There is no need to convert the coordinates of the root UI widget, since its coordinates are relative to the top left corner of the screen, and thus are already absolute coordinates. For other UI widgets, SUPOR computes their absolute coordinates based on their relative coordinates and their parent container's absolute coordinates. For example, the relative coordinates of the second UI widget, `TextView`, are (16, 16, 60, 33). Since it is a child widget of the root UI widget, its absolute coordinates is computed as (16, 66, 60, 83). This process is repeated until the coordinates of every UI widget are converted.

Algorithm 1 UI Widget Sensitiveness Analysis

```
Require: $I$ as an input field, $S$ as a set of text labels, $KW$
    as a pre-defined sensitive keyword dataset
Ensure: $R$ as whether $I$ is sensitive
Divide the UI plane into *nine* partitions based on $I$'s boundary
for all $L \in S$ do
    $score = 0$
    for all $(x,y) \in L$ do
        $score \mathrel{+}= distance(I,x,y) * posWeight(I,x,y)$
    end for
    $L.score = score \ / \ L.numOfPixels$
end for
$T = min(S)$
$R = T.text$ matches $KW$
```

In addition to coordinate conversion, SUPOR collects other information of the UI widgets, such as the texts in the text labels and the attributes for input fields (e.g., `ID` and `inputType`).

3.4 UI Sensitiveness Analysis

Based on the information collected from the layout analysis, the UI sensitiveness analysis component determines whether a given input field contains sensitive information. This component consists of three major steps.

First, if the input field has been assigned with certain attributes like `android:inputType="textPassword"`, it is directly considered as sensitive. With such attribute, the original inputs on the UI are concealed after users type them. In most cases these inputs are passwords.

Second, if the input field contains any hint (*i.e.,* tooltip), *e.g.,* "Enter Password Here", the words in the hint are checked: if it contains any keyword in our sensitive keyword dataset, the input field is considered sensitive; otherwise, the third step is required to determine its sensitiveness.

Third, SUPOR identifies the text label that describes the purpose of the input field, and analyzes the text in the label to determine the sensitiveness. In order to identify text labels that are close to a given input field, we provide

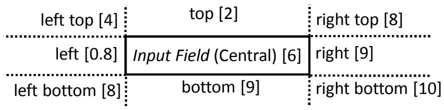


Figure 7: The partition of the UI is based on the boundary of the input field.

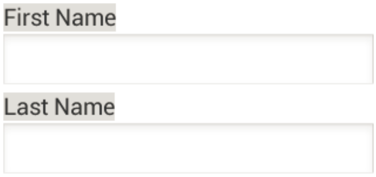


Figure 8: Example for UI widget sensitiveness analysis.

an algorithm to compute correlation scores for each pair of a text label and an input field based on their distances and relative positions.

The details of our algorithm is shown in Algorithm 1. At first, SUPOR divides the UI plane into nine partitions based on the boundaries of the input field. Figure 7 shows the nine partitions divided by an input field. Each text label can be placed in one or more partitions, and the input field itself is placed in the central partition. For a text label, we determine how it is correlated to an input field by computing how each pixel in a text label is correlated to the input field (Line 4). The correlation score for a pixel consists of two parts (Line 5). The first part is the Euclidean distance from the pixel to the input field, computed using the absolute coordinates. The second part is a weight based on their relative positions, *i.e.,* which of the nine partitions the widget is in. We build the position-based weight function based on our empirical observations: if the layout of the apps is top-down and left-right arranged, the text label that describes the input field is usually placed at left or on top of the input field while the left one is more likely to be the one if it exists. We assign smallest weight to the pixels in the left partition and second smallest for the top partition. The right-bottom partition is least possible so we give the largest weight to it. The detailed weights for each partition is shown in Figure 7. Based on the correlation scores of all the pixels, our algorithm uses the average of the correlation scores as the correlation score for the pair of the text label and the input field (Line 7). The label with smaller correlation score is considered more correlated to the input field.

After the correlation scores for all text labels are computed, SUPOR selects the text label that has the smallest score as the descriptive text label for the input field, and uses the pre-defined sensitive keyword dataset to determine if the label contains any sensitive keyword. If yes, the input field is considered as sensitive.

Example. Figure 8 shows an example UI that requires Algorithm 1 for sensitiveness analysis. This example shows a UI that requests a user to enter personal information. This UI contains two input fields and two text labels. Neither can SUPOR determine the sensitiveness through their attributes, nor can SUPOR use any hint to determine the sensitiveness. SUPOR then applies Algorithm 1 on these two input fields to compute the correlation scores for each pair of text labels and input fields. The correlation scores are shown in Table 2. According to the correlation scores, SUPOR associates "First Name" to the first input field and "Last Name" to the second input field. Since our keyword dataset contains keywords "*first name*" and "*last name*" for personal information, SUPOR can declare the two input fields are sensitive.

Table 2: Scores of the text labels in Figure 8.

	First Name	Last Name
1st input field	46.80	218.81
2nd input field	211.29	46.84

Repeating the above steps for every input field in the app, SUPOR obtains a list of sensitive input fields. It assigns an contextual ID to each sensitive input field in the form of `<Layout_ID, Widget_ID>`, where `Layout_ID` is the ID of the layout that contains the input field and `Widget_ID` is the ID of the input field (*i.e.,* the value of the attribute "`android:id`").

3.5 Variable Binding

With the sensitive input fields identified in the previous step, the variable binding component performs context-sensitive analysis to bind the input fields to the variables in the code. The sensitive input fields are identified using contextual IDs, which include layout IDs and widget IDs. These contextual IDs can be used to directly locate input fields from the XML layout files. To find out the variables that store the values of the input fields, SUPOR leverages the binding mechanism provided by Android to load the UI layout and bind the UI widgets with the code. Such a binding mechanism enables SUPOR to associate input fields with the proper variables. We refer to these variables the *widget variables* that are bound to the input fields.

The variable binding component identifies the instances of the input fields in a context-insensitive fashion via searching the code using the APIs provided by the rapid UI development kit of Android. As shown in Section 2.2, `findViewById(ID)` is an API that loads a UI widget to the code. Its argument `ID` is the numeric ID that specifies which widget defined in the XML to load. Thus, to identify the instances of the input fields, SUPOR searches the code for such method calls, and compare their arguments to the widget IDs of the sensitive input fields. If the arguments match any widget ID of the sensitive input fields, the return values of the corresponding `findViewById(ID)` are considered as the widget variables for the sensitive input fields.

One problem here is that developers may assign the same widget ID to UI widgets in different layout files, and thus different UI widgets are associated with the same numeric ID in the code. Our preliminary analysis on 5000 apps discovers that about 22% of the identified sensitive input fields have duplicate IDs within the corresponding apps. Since the context-insensitive analysis cannot distinguish the duplicate widget IDs between layout files inside an app, a lot of false positives will be presented.

To reduce false positives, SUPOR adds context-sensitivity into the analysis, associating widget variables with their corresponding layouts. Similar to loading a widget, the rapid UI development kit provides APIs to load a UI layout into the code. For example, `setContentView(ID)` with a numeric ID as the argument is used to load a UI layout to the code, as shown at Line 6 in Figure 3. Any subsequent `findViewById` with the ID `WID` as the argument returns the UI widget identified by `WID` in the newly loaded UI layout, not the UI widget identified by `WID` in the previous UI layout. Thus, to find out which layout is associated with a given widget variable, SUPOR traces back to identify the closest method call that loads a UI layout[1] along the program paths that lead to the invocation of `findViewById`. We next describe how SUPOR performs context-sensitive analysis to distinguish widget IDs between layout files. For the description below, we use `setContentView()` as an example API.

Given a widget variable, SUPOR first identifies the method call `findViewById`, and computes an interprocedural backward slice [18] of its receiver object, *i.e.,* the activity object. This backward slice traces back from `findViewById`, and includes all statements that may affect the state of the activity object. SUPOR then searches the slice backward for the method call `setContentView`, and uses the argument of the first found `setContentView` as the layout ID. For example, in Figure 3, the widget variable `txtUid` is defined by the `findViewById` at Line 7, and the activity object of this method call is an instance of `LoginActivity`. From the backward slice of the activity object, the first method call `setContentView` is found at Line 6, and thus its argument `R.layout.login_activity` is associated with `txtUid`, whose widget ID is specified by `R.id.uid`. Both `R.layout.login_activity` and `R.id.uid` can be further resolved to identify their numeric IDs, and match with the contextual IDs of sensitive input fields to determine whether `txtUid` is a widget variable for a sensitive input field.

3.6 Keyword Dataset Construction

To collect the sensitive keyword dataset, we crawl all texts in the resource files from 54,371 apps, including layout files and string resource files. We split the collected texts based on newline character (`\n`) to form a list of texts, and extract words from the texts to form a list of words. Both of these lists are then sorted based on the frequencies of text lines and words, respectively. We then systematically inspect these two lists with the help of the adapted NLP techniques. Next we describe how we identify sensitive keywords in detail.

First, we adapt NLP techniques to extract nouns and noun phrases from the top 5,000 frequent text lines. Our technique first uses Stanford parser [36] to parse each text line into a syntactic tree as discussed in Section 2.4, and then traverses the parse tree level by level to identify nouns and noun phrases. For the text lines that do not contain any noun or noun phrase, our technique filters out these text lines, since such text lines usually consist of only prepositions (*e.g.,* to), verbs (*e.g.,* update please), or unrecognized symbols. From the top 5,000 frequent text lines, our technique extracts 4,795 nouns and noun phrases. For the list of words, our technique filters out words that are not nouns due to the similar reasons. From the top 5,000 frequent words, our technique obtains 3,624 words. We then manually inspect these two sets of frequent nouns and noun phrases to identify sensitive keywords. As phrases other than noun phrases may indicate sensitive information, we further extract consecutive phrases consisting of two and three words from the text lists and manually inspect the top 200 frequent two-word and three-word phrases to expand our sensitive keyword set.

Second, we expand the keyword set by searching the list of text lines and the list of words using the identified words. For example, we further find "cvv code" for credit card by searching the lists using the top-ranked word "code", and find "national ID" by searching the lists using the top-ranked word "id". We also expand the keywords using synonyms of the keywords based on WordNet [11].

Third, we further expand the keywords by using Google Translate to translate the keywords from English into other languages. Currently we support Chinese and Korean besides English.

These keywords are manually classified into 10 categories, and part of the keyword dataset is presented in Table 3. Note that we do not use "Address" for the category "Personal Info". Although personal address is sensitive information, our preliminary results show that this

[1] SUPOR considers both `Activity.setContentView()` and `LayoutInflater.inflate()` as the methods to load UI layouts due to their prevalence.

Table 3: Part of keyword dataset.

Category	Keywords
Credential	pin code, pin number, password
Health	weight, height, blood type, calories
Identity	username, user ID, nickname
Credit Card	credit card number, cvv code
SSN	social security number, national ID
Personal Info	first name, last name, gender, birthday
Financial Info	deposit amount, income, payment
Contact	phone number, e-mail, email, gmail
Account	log in, sign in, register
Protection	security answer, identification code

keyword also matches URL address bars in browsers, causing many false positives. Also, we do not find interesting privacy disclosures based on this keyword in our preliminary results, and thus "Address" is not used in our keyword dataset. Although this keyword dataset is not a complete dataset that covers every sensitive keyword appearing in Android apps, our evaluation results (in Section 5) show that it is a relatively complete dataset for the ten categories that we focus on in this work.

4 Implementation

In this section, we provide the details of our implementation of SUPOR, including the frameworks and tools we built upon and certain tradeoffs we make to improve the effectiveness.

SUPOR accepts APK files as inputs, and uses a tool built on top of Apktool [1] to extract resource files and bytecode from the APK file. The Dalvik bytecode is translated into an intermediate representation (IR), which is based on dexlib in Baksmali [3]. The IR is further converted to WALA [4] static single assignment format (SSA). WALA [4] works as the underlying analysis engine of SUPOR, providing various functionalities, *e.g.,* call graph building, dependency graph building, and point-to analysis.

The UI rendering engine is built on the UI rendering engine from the ADT Eclipse plug-ins Besides improving the engine to better render custom widgets, we also make the rendering more resilient using all available themes. Due to SDK version compatibility, not every layout can be rendered in every theme. We try multiple themes until we find a successful rendering. Although different themes might make UI slightly different, the effectiveness of our algorithm should not be affected. The reason is that apps should not confuse users in the successfully rendered themes, and thus our algorithm designed to mimic what users see the UIs should work accordingly.

To demonstrate the usefulness of SUPOR, we implement a privacy disclosure detection system by combining SUPOR with static taint analysis. This system enables us to conduct a study on the disclosures of sensitive user inputs. We build a taint analysis engine on top of Dalysis [24] and make several customizations to improve the effectiveness. The details of the customizations can be found at Appendix A.2.

To identify sensitive user inputs, SUPOR includes totally 11 source categories, including the 10 categories listed in Section 3.6 and an additional category *PwdLike* for the input fields identified as sensitive using their attributes such as `inputType`. The *PwdLike* category is prioritized if it has some overlapping with the other categories. Once the widget variables of the sensitive input fields are found, we consider any subsequent method calls on the variables that retrieve values from the input fields as source locations, such as `getText()`. To identify privacy disclosures of the sensitive user inputs, SUPOR mainly focuses on the information flows that transfer the sensitive data to the following two types of sinks: (1) the sinks of output channels that send the information out from the phone (*e.g.,* SMS and Network) and (2) the sinks of public places on the phone (*e.g.,* logging and content provider writes). More details are shown in Appendix A.1.

Our implementation, excluding the underlying libraries and the core taint analysis engine, accounts for about 4K source lines of code (SLoC) in Java.

5 Evaluations and Experiments

We conducted comprehensive evaluations on SUPOR over a large number of apps downloaded from the official Google Play store. We first evaluated the performance of SUPOR and demonstrated its scalability. We then measured the accuracy of the UI sensitiveness analysis and the accuracy of SUPOR in detecting disclosures of sensitive user inputs. In addition, our case studies on selected apps present practical insights of sensitive user input disclosures, which are expected to contribute to a community awareness.

5.1 Evaluation Setup

The evaluations of SUPOR were conducted on a cluster of eight servers with an Intel Xeon CPU E5-1650 and 64/128GB of RAM. During the evaluations, we launched concurrent SUPOR instances on 64-bit JVM with a maximum heap space of 16GB. On each server 3 apps were concurrently analyzed, so the cluster handled 24 apps in parallel.

In our evaluations, we used the apps collected from the official Google Play store in June 2013. We applied SUPOR to analyze 6,000 apps ranked by top downloads, with 200 apps for each category. Based on the results of the 6,000 apps, we further applied SUPOR on another 10,000 apps in 20 selected categories. Each of the 20 categories is found to have at least two apps with sensi-

Table 4: Statistics of 16,000 apps.

	#Apps	Percentage
Without Layout Files	625	3.91%
Without Input Fields	5,711	35.69%
Without Sensitive Input Fields	4,731	29.57%
With Sensitive Input Fields	4,922	30.76%
Parsing Errors	11	0.07%
TOTAL	16,000	100.00%

tive user input disclosures.

For each app, if it contains at least one input field in layout files, the app is analyzed by the UI sensitiveness analysis. If SUPOR identifies any sensitive input field of the app, the app is further analyzed by the taint analysis to detect sensitive user input disclosures. Table 4 shows the statistics of these apps. A small portion of the apps do not contain any layout files and about 1/3 of the apps do not have any input field in layout files. This is reasonable because many Game apps do not require users to enter information. 35% of the apps without layout files and 17% of the apps without input fields belong to different sub-categories of games. 11 apps (0.07%) cannot be analyzed by SUPOR due to various parsing errors in rendering their layout files. In total, 60.33% of the apps contain input fields in their layout files, among which more than half of the apps are further analyzed because sensitive input fields are found via the UI sensitiveness analysis.

As not every layout containing input fields is identified with sensitive input fields, we show the statistics of the layouts for the 4,922 apps identified with sensitive input fields. Among these apps, 47,885 layouts contain input fields and thus these layouts are rendered. Among the rendered layouts, 19,265 (40.2%) are found to contain sensitive keywords (no matter whether the keywords are associated with any input field). This is the upper bound of the number of layouts that can be identified with sensitive input fields. In fact, 17,332 (90.0%) of the 19,265 layouts with sensitive keywords are identified with sensitive input fields.

5.2 Performance Evaluation

The whole experiment for 16,000 apps takes 1439.8 minutes, making a throughput of 11.1 apps per minutes on the eight-server cluster. The following analysis is only for the 4,922 apps identified with sensitive input fields, if not specified.

The UI analysis in SUPOR includes decompiling APK files, rendering layouts, and performing UI sensitiveness analysis. For each app with sensitive input fields, SUPOR needs to perform the UI analysis for at least 1 layout and at most 190 layouts, while the median number is 7 and the average number is 9.7. Though the largest execution time required for this analysis is about 2 minutes. 96.3% of the apps require less than 10 seconds to render all layouts in an app. The median analysis time is 5.2 seconds and the average time is 5.7 seconds for one app. Compared with the other parts of SUPOR, the UI analysis is quite efficient, accounting for only 2.5% of the total analysis time on average. Also, the UI sensitiveness analysis, including the correlation score computation and keyword matching, accounts for less than 1% of the total UI analysis time, while decompiling APK files and rendering layouts take most of the time.

To detect sensitive user input disclosures, our evaluation sets a maximum analysis time of 20 minutes. 18.1% of the apps time out in our experiments but 73.7% require less than 10 minutes. The apps with many entry points tend to get stuck in taint analysis, and are more likely to timeout. Scalability of static taint analysis is a hard problem, but we are not worse than related work. The timeout mechanism is enforced for the whole analysis, but the system will wait for I/O to get partial results. In practice, we can allow a larger maximum analysis time so that more apps can be analyzed. Among the apps finished in time, the median analysis time is 1.9 minutes and the average analysis time is 3.7 minutes.

The performance results show that SUPOR is a scalable solution that can statically analyze UIs of a massive number of apps and detect sensitive user input disclosures on these apps. Compared with existing static taint analysis techniques, the static UI analysis introduced in this work is highly efficient, and its performance overhead is negligible.

5.3 Effectiveness of UI Sensitiveness Analysis

To evaluate the accuracy of the UI sensitiveness analysis, we randomly select 40 apps and manually inspect the UIs of these 40 apps to measure the accuracy of the UI sensitiveness analysis.

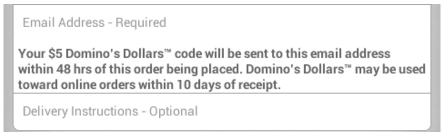
First, we randomly select 20 apps reported *without* sensitive input fields, and manually inspect these apps to measure the false negatives of SUPOR. In these apps, the largest number of layouts SUPOR renders is 5 and the total number of layouts containing input fields is 39 (1.95 layouts per app). SUPOR successfully renders 38 layouts and identifies 57 input fields (2.85 input fields per app). SUPOR fails to render 1 layout due to the lack of necessary themes for a third-party library. By analyzing these 57 input fields, we confirm that SUPOR has only one false negative (FN), *i.e.,* failing to mark one input field as sensitive in the app *com.strlabs.appdietas*. This input field requests users to enter their weights, belonging to the Health category in our keyword dataset. However, the text of the descriptive text label for the input field is “Peso de hoy”, which is “Today Weight” in Spanish. Since our keyword dataset focuses on sensitive keywords in English, SUPOR has a false negative. Such false negatives can be reduced by expanding our keyword dataset to support more languages.

Second, we randomly select 20 apps reported *with* sensitive input fields. Table 5 shows the detailed analysis results. Column "#Layouts " counts the number of layouts containing input fields in each app, while Column "#Layouts with Sensitive Input Fields" presents the number of layouts reported with sensitive input fields. Column "#Input Fields" lists the total number of input fields in each app and Column "#Reported Sensitive Input Fields" gives the detailed information about how many input fields are identified by checking the `inputType` attribute, by matching the hint text, and by analyzing the associated text labels. Sub-Column "Total" presents the total number of sensitive input fields identified by SUPOR in each app. Columns "FP" and "FN" show the number of false positives and the number of false negatives produced by SUPOR in classifying input fields. Column "Duplicate ID" shows if an app contains any duplicate widget ID for sensitive input fields. These duplicate IDs belong to either sensitive input fields (represented by ○) or non-sensitive input fields (●). For all the layouts in these 20 apps, SUPOR successfully renders the layouts except for App 18, which has 29 layouts containing input fields but SUPOR renders only 17 layouts. The reason is that Apktool fails to decompile the app completely.

The results show that for these 20 apps, SUPOR identifies 149 sensitive input fields with 4 FPs and 3 FNs, and thus the achieved true positives (TP) is 145. Combined with the 20 apps identified without sensitive input fields (0 FP and 1 FN), SUPOR achieves an average precision of 97.3% (precision = $\frac{TP}{TP+FP}$ = 145/149) and an average recall of 97.3% (recall = $\frac{TP}{TP+FN}$ = 145/(145+(1+3)).

We next describe the reasons for the FNs and the FPs. SUPOR has two false negatives in App 1, in which the text label "Answer" is not identified as a sensitive keyword. But according to the context, it means "security answer", which should be sensitive. Although this phrase is modeled as a sensitive phrase in our keyword dataset, SUPOR cannot easily associate "Answer" with the phrase, resulting in a false negative. In App 8, SUPOR marks an input field as sensitive because the associated text label containing the keyword "Height". However, based on the context, the app actually asks the user to enter the expected page height of a PDF file. Such issues can be alleviated by employing context-sensitive NLP analysis [19].

SUPOR also has two FPs in App 6 and App 8 due to the inaccuracy of text label association. In App 6 shown in Figure 9, the hint of the "Delivery Instructions" input field does not contain sensitive keywords, and thus SUPOR identifies the close text label for determining its sensitiveness. However, SUPOR incorrectly associates a description label of "Email" to the "Delivery Instructions" input field based on their close distances. Since this description contains sensitive keywords such as email, SUPOR considers the "Delivery Instructions" input field as sensitive, causing a false positive. Finally, SUPOR has both FPs and FNs for App 14, since its arrangements of input fields and their text labels are not accurately captured by our position-based weights that give preferences for left and top positioned text labels.



Email Address - Required

Your $5 Domino's Dollars™ code will be sent to this email address within 48 hrs of this order being placed. Domino's Dollars™ may be used toward online orders within 10 days of receipt.

Delivery Instructions - Optional

Figure 9: False positive example in UI sensitiveness analysis.

To evaluate the effectiveness of resolving duplicate IDs, We instrumented SUPOR to output detailed information when identifying the widget variables. We did not find any case where SUPOR incorrectly associates the widget variables with the input fields based on the contextual IDs, but potentially SUPOR may have inaccurate results due to infeasible sequences of entry points that can be executed. We next present an example to show how backward slicing help SUPOR distinguish duplicate widget IDs. App 17 has two layouts with the same hierarchy. Layout A contains a sensitive input field with the ID $w1$ while Layout B contains a non-sensitive input field with the same ID $w1$. Both layouts are loaded via `LayoutInflater.inflate` and then `findViewById` is invoked separately to obtain the enclosed input fields. Without the backward slicing, SUPOR considers the input field with the ID $w1$ in the Layout B as sensitive, which is a false positive. With the backward slicing, SUPOR can distinguish the input field with the ID $w1$ in Layout B with the input field with the ID $w1$ in Layout A, and correctly filter out the non-sensitive input field in Layout B.

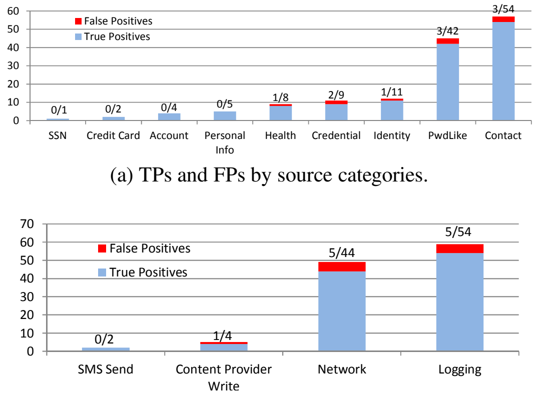
5.4 Accuracy of Detecting Sensitive User Input Disclosures

In our experiments, 355 apps are reported with sensitive user input disclosures. The reported apps belong to 25 out of the 30 categories in Google Play Store and 20 categories have at least 2 apps reported. We next report the accuracy of detecting sensitive user input disclosures.

Figure 10 shows the number of true positives and the number of false positives by taint source and sink categories. If an app is reported with multiple disclosure flows and one of them is a false positive, the app is considered as a false positive. Through manually evaluating the 104 apps reported cases from the first 6,000 analyzed apps, we find false positives in 9 apps. Therefore, the overall false positive rate is about 8.7%, *i.e.,* the accu-

Table 5: UI analysis details for 20 randomly chosen apps.

App ID	#Layouts	#Input Fields	#Layouts with Sensitive Input Fields	#Reported Sensitive Input Fields				FP	FN	Duplicate ID
				Password	Hint	Label	Total			
1	8	18	4	6	0	3	9		2	○
2	37	77	2	0	0	8	8			
3	3	3	1	0	1	0	1			
4	4	9	3	0	0	6	6			○
5	5	7	1	1	0	0	1			
6	17	52	10	6	12	12	30	1		○
7	4	5	2	0	0	3	3			
8	15	22	9	8	3	2	13	1		
9	3	7	1	1	1	0	2			
10	7	16	1	0	0	1	1			
11	5	6	1	1	1	0	2			
12	17	33	8	8	9	0	17			○ ●
13	26	60	10	0	0	12	12			○ ●
14	2	8	2	1	0	4	5	2	1	
15	14	26	5	2	3	0	5			○ ●
16	4	7	1	1	0	0	1			
17	4	8	3	2	3	0	5			●
18	29	25	4	4	0	6	10			○
19	24	37	8	9	6	1	16			○
20	1	2	1	0	2	0	2			
Total	229	428	77	50	41	58	149	4	3	



(a) TPs and FPs by source categories.

(b) TPs and FPs by sink categories.

Figure 10: True positives and false positives by source/sink categories for the reported apps.

racy of privacy disclosure detection is 91.3%. We investigated the false positives and found that these false positives were mostly resulted from the limitations of the underlying taint analysis framework, such as the lack of accurate modeling of arrays.

5.5 Case Studies

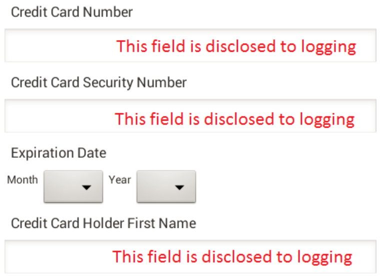
To improve the community's awareness and understanding of sensitive user input disclosures, we conducted cases studies on four selected apps from the source categories SSN, PwdLike, Credit Card, and Health. These case studies present interesting facts of sensitive user input disclosures, and also demonstrate the usefulness of SUPOR. We also inform the developers of the apps mentioned in this section about the detected disclosures.

com.yes123.mobile is an app for job hunting. The users are required to register with their national ID and a password to use the service. When the users input the ID and password, and then click log in[2], the app sends both their national IDs and passwords via Internet without any protection (*e.g.,* hashing or through HTTPS channel). Since national ID is quite sensitive (similar as Social Security Number), such limited protection in transmission may lead to serious privacy disclosure problems.

The second example app (*craigs.pro.plus*) shows a legitimate disclosure that uses HTTPS connections to send user sensitive inputs to its server for authentication. Even though the password itself is not encoded (*e.g.,* hashing), we believe HTTPS connections provide a better protection layer to resist the disclosures during communications. Also we find that popular apps developed by enterprise companies are more likely to adopt HTTPS, providing better protection for their users.

To better understand whether sensitive user inputs are properly protected, we further inspect 104 apps, of which 44 apps send sensitive user inputs via network. Among these 44 apps, only 10 of them adopt HTTPS connections, while the majority of apps transmit sensitive user inputs in plain text via HTTP connections. Such study

[2] The UI is shown in Figure 12 in Appendix B.1.



Credit Card Number

This field is disclosed to logging

Credit Card Security Number

This field is disclosed to logging

Expiration Date

Credit Card Holder First Name

This field is disclosed to logging

Figure 11: Case study: credit card information disclosure example.

results indicate that most developers are still unaware of the risks posed by sensitive user input disclosures, and more efforts should be devoted to provide more protections on sensitive user inputs.

Our last example app (*com.nitrogen.android*) discloses credit card information, a critical financial information provided by the users. Figure 11 shows the rendered UI of the app. The three input fields record credit card number, credit card security number, and the card holder's name. Because these fields are not decorated with `textPassword` input type and they do not contain any hints, SUPOR uses the UI sensitiveness analysis to compute correlation scores for each text label. As we can see from the UI, the text label "Credit Card Number" and the text label "Credit Card Security Number" are equally close to the first input field. As our algorithm considers weights based on the relative positions between text labels and input fields, SUPOR correctly associates the corresponding text labels for these three input fields, and the taint analysis identifies sensitive user input disclosures for all these three input fields to logging. SUPOR also identifies apps that disclose personal health information to logging, and the example app is shown in Appendix B.2.

Although Google tries to get rid of some of the known sinks that contribute most of the public leaks by releasing new Android versions, many people globally may still continue using older Android releases for a very long time (about 14.2% of Android phones globally using versions older than Jelly Bean [2]). If malware accesses the logs on these devices, all the credit card information can be exploited to malicious adversaries. Thus, certain level of protection is necessary for older versions of apps. Also, SUPOR finds that some apps actually sanitize the sensitive user inputs (*e.g.,* hashing) before these inputs are disclosed in public places on the phone, indicating that a portion of developers do pay attention to protecting sensitive user input disclosures on the phone.

6 Discussion

SUPOR is designed as an effective and scalable solution to screening a large number of apps for sensitive user inputs. In this work, we have demonstrated that SUPOR can be combined with static taint analysis to automatically detect potential sensitive user input disclosures. Such analysis can be directly employed by app markets to raise warnings, or by developers to verify whether their apps accidentally disclose sensitive user inputs. Also, SUPOR can be paired with dynamic taint analysis to alert users before the sensitive user inputs escape from the phones.

SUPOR focuses on input fields, a major type of UI widgets to collect user inputs. Such UI widgets record what user type and contain high entropy, unlike yes/no buttons which contain low entropy. It is quite straightforward to extend our current approach to handle more diverse widgets.

SUPOR chooses the light-weight keyword-based technique to determine the sensitiveness of input fields since the texts contained in the associated text labels are usually short and straightforward to understand. Our evaluations show that in general these keywords are highly effective in determining the sensitiveness of input fields. Certain keywords may produce false positives since these keywords have different meanings under different contexts. To alleviate such issues, we may leverage more advanced NLP techniques that consider contexts [19].

7 Related Work

Many great research works [6, 8–10, 12, 14, 16, 23, 24, 34, 38] focus on privacy leakage problems on predefined sensitive data sources on the phone. SUPOR identifies sensitive user inputs, and may enable most of the existing research on privacy studies to be applied to sensitive user inputs. As a result, our research compliments the existing works. FlowDroid [6] also employs a limited form of sensitive input fields—password fields. Compared with FlowDroid, we leverage static UI rendering and NLP techniques to identify different categories of sensitive input fields in an extensible manner. Susi [33] employs a machine learning approach to detect pre-defined source/sinks from Android Framework. In contrast, SUPOR focus on a totally different type of sensitive sources–user inputs through GUI.

Moreover, a few approaches are designed for controlling the known privacy leaks. AppFence [17] employs fake data or network blocking to protect privacy leaks to Internet with user supplied policies. Nadkarni *et al.*provide new OS mechanisms for proper information sharing cross apps [28].

NLP techniques have been used to study app descriptions [13, 30, 31]. WHYPER [30] and AutoCog [31] leverages NLP techniques to understand whether the application descriptions reflect the permission usage. CHABADA [13] also applies topic modelling, an NLP technique to detecting malicious behaviors of Android apps. It generates clusters according to the topic, which consists of a cluster of words that frequently occur together. Then, it tries to detect the outliers as malicious behaviors. CHABADA does not focus on detecting privacy leaks. On the other hand, SUPOR leverages NLP techniques to identify sensitive keywords and further use those keywords to classify the descriptive text labels and the associated input fields.

Furthermore, there are a few important related works using UI related information to detect different types of vulnerabilities and attacks. AsDroid [20] checks UI text to detect the contradiction between expected behavior inferred from the UI and the program behavior represented by APIs. Chen *et al.*study the GUI spoofing vulnerabilities in IE browser [7]. Mulliner *et al.*discover GUI element misuse (GEM), a type of GUI related access control violation vulnerabilities and design GEM Miner to automatically detect GEMs [27]. SUPOR focuses on sensitive user input identification which is different from the problems studied by these existing works.

The closest related work is UIPicker [29], which also focuses on sensitive user input identification. UIPicker uses supervised learning to train a classifier based on the features extracted from the texts and the layout descriptions of the UI elements. It also considers the texts of the sibling elements in the layout file. Unlike UIPicker that uses sibling elements in the layout file as the description text for a UI widget, which could easily include unrelated texts as features, SUPOR selects only the text labels that are physically close to input fields in the screen, mimicking how users look at the UI, and uses the texts in the text labels to determine the sensitiveness of the input fields. Also, their techniques in extracting privacy-related texts could complement our NLP techniques to further improve our keyword dataset construction.

In the software engineering domain, there are quite a few efforts on GUI reverse engineering [25, 26, 32, 35] for GUI testing. GUITAR is a well-known framework for general GUI testing, and GUI ripper [26], a component of GUITAR targets general desktop applications, uses dynamic analysis to extract GUI related information and requires human intervention when the tools cannot fill in proper information in the applications. In [25] and [32], two different approaches have been proposed to convert the hard-coded GUI layout to model-based layout (such as XML/HTML layout). GUISurfer leverages source code to derive the relationships between different given UI widgets. In contract, SUPOR focuses on mobile apps and in particular Android apps, and leverages the facility from existing rapid UI development kits to identify and render UI widgets statically.

8 Conclusions

In this paper, we study the possibility of scalably detecting sensitive user inputs, an important yet mostly neglected sensitive source in mobile apps. We leverage the rapid UI development kits of modern mobile OSes to detect sensitive input fields and correlate these input fields to the app code, enabling various privacy analyses on sensitive user inputs. We design and implement SUPOR, a new static analysis tool that automatically identifies sensitive input fields by analyzing both input field attributes and surrounding descriptive text labels through static UI parsing and rendering. Leveraging NLP techniques, we build mobile app specific sensitive word vocabularies that can be used to determine the sensitiveness of given texts. To enable various privacy analyses on sensitive user inputs, we further propose a context-sensitive approach to associate the input fields with corresponding variables in the app code.

To demonstrate the usefulness of SUPOR, we build a privacy disclosure discovery system by combining SUPOR with static taint analysis to analyze the sensitive information of the variables that store the user inputs from the identified sensitive input fields. We apply the system to 16,000 popular Android apps, and SUPOR achieves an average precision of 97.3% and also an average recall of 97.3% in detecting sensitive user inputs. SUPOR finds 355 apps with privacy disclosures and the false positive rate is 8.7%. We also demonstrate interesting real-world cases related to national ID, username/password, credit card and health information.

9 Acknowledgements

The authors would like to thank the anonymous reviewers for their insightful comments that helped improve the presentation of this paper. Jianjun Huang and Xiangyu Zhang are supported, in part, by National Science Foundation (NSF) under grants 0845870, 1320444, 1320326 and 1409668. Any opinions, findings, and conclusions or recommendations in this paper are those of the authors and do not necessarily reflect the views of NSF.

References

[1] Android-ApkTool: A tool for reverse engineering Android apk file. https://code.google.com/p/android-apktool.

[2] Android Dashboards. https://developer.android.com/about/dashboards/index.html. Accessed: 20 Feb 2015.

[3] Baksmali: a disassembler for Android's dex format. https://code.google.com/p/smali.

[4] WALA: T.J. Watson Libraries for Analysis. http://wala.sourceforge.net.

[5] ARP, D., SPREITZENBARTH, M., HUBNER, M., GASCON, H., AND RIECK, K. DREBIN: Effective and explainable detection of Android malware in your pocket. In *NDSS* (2014).

[6] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI* (2014).

[7] Chen, S., Meseguer, J., Sasse, R., Wang, H. J., and Wang, Y.-M. A systematic approach to uncover security flaws in GUI logic. In *S&P (Oakland)* (2007).

[8] Egele, M., Kruegel, C., Kirda, E., and Vigna, G. PiOS: Detecting privacy leaks in iOS applications. In *NDSS* (2011).

[9] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI* (2010).

[10] Enck, W., Octeau, D., McDaniel, P., and Chaudhuri, S. A study of Android application security. In *USENIX Security* (2011).

[11] Fellbaum, C., Ed. *WordNet An Electronic Lexical Database*. The MIT Press, 1998.

[12] Gibler, C., Crussell, J., Erickson, J., and Chen, H. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *TRUST* (2012).

[13] Gorla, A., Tavecchia, I., Gross, F., and Zeller, A. Checking app behavior against app descriptions. In *ICSE* (2014).

[14] Grace, M., Zhou, Y., Wang, Z., and Jiang, X. Systematic detection of capability leaks in stock Android smartphones. In *NDSS* (2012).

[15] Grace, M., Zhou, Y., Zhang, Q., Zou, S., and Jiang, X. Riskranker: Scalable and accurate zero-day Android malware detection. In *MobiSys* (2012).

[16] Han, J., Yan, Q., Gao, D., Zhou, J., and Deng, R. Comparing mobile privacy protection through cross-platform applications. In *NDSS* (2013).

[17] Hornyack, P., Han, S., Jung, J., Schechter, S., and Wetherall, D. These aren't the droids you're looking for: Retrofitting Android to protect data from imperious applications. In *CCS* (2011).

[18] Horwitz, S., Reps, T., and Binkley, D. Interprocedural slicing using dependence graphs. *SIGPLAN Not. 23*, 7 (June 1988).

[19] Huang, E. H., Socher, R., Manning, C. D., and Ng, A. Y. Improving word representations via global context and multiple word prototypes. In *ACL* (2012).

[20] Huang, J., Zhang, X., Tan, L., Wang, P., and Liang, B. Asdroid: Detecting stealthy behaviors in Android applications by user interface and program behavior contradiction. In *ICSE* (2014).

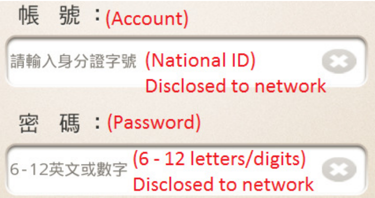
[21] Jurafsky, D., and Martin, J. H. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, 1st ed. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[22] Klein, D., and Manning, C. D. Accurate unlexicalized parsing. In *ACL* (2003).

[23] Lu, K., Li, Z., Kemerlis, V., Wu, Z., Lu, L., Zheng, C., Qian, Z., Lee, W., and Jiang, G. Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting. In *NDSS* (2015).

[24] Lu, L., Li, Z., Wu, Z., Lee, W., and Jiang, G. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In *CCS* (2012).

[25] Lutteroth, C. Automated reverse engineering of hard-coded GUI layouts. In *AUIC* (2008).



帳　號：(Account)

請輸入身分證字號 (National ID)
Disclosed to network

密　碼：(Password)

6-12英文或數字 (6 - 12 letters/digits)
Disclosed to network

Figure 12: Case study: national ID and password disclosure example without protection.

[26] Memon, A., Banerjee, I., and Nagarajan, A. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *WCRE* (2003).

[27] Mulliner, C., Robertson, W., and Kirda, E. Hidden GEMs: Automated discovery of access control vulnerabilities in graphical user interfaces. In *S&P (Oakland)* (2014).

[28] Nadkarni, A., and Enck, W. Preventing accidental data disclosure in modern operating systems. In *CCS* (2013).

[29] Nan, Y., Yang, M., Yang, Z., Zhou, S., Gu, G., and Wang, X. UIPicker: User-input privacy identification in mobile applications. In *USENIX Security* (2015).

[30] Pandita, R., Xiao, X., Yang, W., Enck, W., and Xie, T. WHYPER: Towards automating risk assessment of mobile applications. In *USENIX Security* (2013).

[31] Qu, Z., Rastogi, V., Zhang, X., Chen, Y., Zhu, T., and Chen, Z. AutoCog: Measuring the description-to-permission fidelity in Android applications. In *CCS* (2014).

[32] Ramón, Ó. S., Cuadrado, J. S., and Molina, J. G. Model-driven reverse engineering of legacy graphical user interfaces. *Automated Software Engineering 21*, 2 (2014).

[33] Rasthofer, S., Arzt, S., and Bodden, E. A machine-learning approach for classifying and categorizing Android sources and sinks. In *NDSS* (2014).

[34] Rastogi, V., Chen, Y., and Enck, W. AppsPlayground: Automatic security analysis of smartphone applications. In *ASIACCS* (2013).

[35] Silva, J. C., Silva, C., Gonçalo, R. D., Saraiva, J., and Campos, J. C. The GUISurfer tool: towards a language independent approach to reverse engineering GUI code. In *EICS* (2010).

[36] The Stanford Natural Language Processing Group, 1999. `http://nlp.stanford.edu/`.

[37] Socher, R., Bauer, J., Manning, C. D., and Ng, A. Y. Parsing with compositional vector grammars. In *ACL* (2013).

[38] Yang, Z., Yang, M., Zhang, Y., Gu, G., Ning, P., and Wang, X. S. AppIntent: Analyzing sensitive data transmission in Android for privacy leakage detection. In *CCS* (2013).

[39] Zhou, Y., and Jiang, X. Dissecting Android malware: Characterization and evolution. In *S&P (Oakland)* (2012).

[40] Zhou, Y., and Jiang, X. Detecting passive content leaks and pollution in Android applications. In *NDSS* (2013).

Appendix

A Taint Analysis

The details of sinks and customizations of the taint analysi engine are shown in this section.

A.1 Sink Dataset

The sink dataset includes five categories of sink APIs, among which two categories are SMS send (*e.g.,*`SmsManager.sendTextMessage()`) and Network (*e.g.,*`HttpClient.execute()`). The other three are related to local storage: logging (*e.g.,*`Log.d()`), content provider writes (*e.g.,*`ContentResolver.insert()`), and local file writes (*e.g.,*`OutputStream.write()`). Totally there are 236 APIs.

A.2 Customizations of Taint Analysis

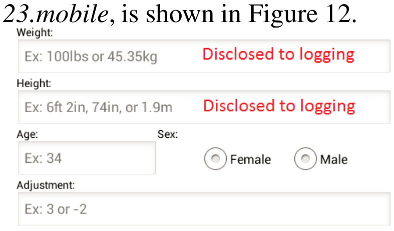
Our taint analysis engine constraints the taint propagation to only variables and method-call returns of `String` type. Therefore, method calls that return primitive types (*e.g.,*`int`) are ignored. There are two major reasons for making this tradeoff. The first is that the sensitive information categories we focus on are passwords, user names, emails, and so on, and these are usually not numeric values. The second is that empirically we found a quite number of false positives related to flows of primitive types due to the incompleteness of API models for the Android framework. This observation-based refinement suppresses many false positives. For example, one false warning we observed is that the length of a tainted string (`tainted.length()`) is logged, and tracking such length causes too many false positives afterwards. Since such flow does not disclose significant information of the user inputs, removing the tracking of such primitive values reduces the sources to track and improves the precision of the tracking.

To further suppress false warnings, we model data structures of key-value pairs, such as `Bundle` and `BasicNameValuePair`. `Bundle` is widely used for storing an activity's previously frozen state, and `BasicNameValuePair` is usually used to encode name-value pairs for HTTP URL parameters or other web transmission parameters, such as JSON. For each detected disclosure flow, we record the keys when the analysis finds method calls that insert values into the data structures, *e.g.,*`bundle.put("key1", tainted)`. For any subsequent method call that retrieves values from the data structures, *e.g.,*`bundle.get("key2")`, we compare the key for retrieving values `key2` with the recorded keys. If no matches are found, we filter out the disclosure flow.

B Example Apps in Case Studies

B.1 Example App for Disclosing National IDs

The UI for the first example app described in Section 5.5, *com.yes123.mobile*, is shown in Figure 12.



Weight:

Ex: 100lbs or 45.35kg — Disclosed to logging

Height:

Ex: 6ft 2in, 74in, or 1.9m — Disclosed to logging

Age: Ex: 34

Sex: ○ Female ○ Male

Adjustment: Ex: 3 or -2

Figure 13: Case study: health information disclosure.

B.2 Example App for Disclosing Health Information

Figure 13 shows the rendered UI of the layout *dpacacl* in app *com.canofsleep.wwdiary*, which belongs to the category HEALTH && FITNESS. This app discloses personal health information through the user inputs collected from the UI. As we can see, even though all input fields on the UI hold hint texts, these texts do not contain any sensitive keywords. Therefore, SUPOR still needs to identify the best descriptive text label for each input field. Based on the UI sensitiveness analysis, SUPOR successfully marks the first three input fields as sensitive, *i.e.,* the input fields that accept *weight*, *height* and *age*. But based on the taint analysis, only the first two input fields are detected with disclosure flows to logging. Similar to financial information, such health information about users' wellness is also very sensitive to the users.

UIPicker: User-Input Privacy Identification in Mobile Applications

Yuhong Nan [1], Min Yang [1], Zhemin Yang [1], Shunfan Zhou [1], Guofei Gu [2], and XiaoFeng Wang [3]

[1]School of Computer Science, Fudan University
[1]Shanghai Key Laboratory of Data Science, Fudan University
[2]SUCCESS Lab, Texas A&M University
[3]Indiana University at Bloomington
{*nanyuhong, m_yang, yangzhemin, 11300240020*}*@fudan.edu.cn*
guofei@cse.tamu.edu, xw7@indiana.edu

Abstract

Identifying sensitive user inputs is a prerequisite for privacy protection. When it comes to today's program analysis systems, however, only those data that go through well-defined system APIs can be automatically labelled. In our research, we show that this conventional approach is far from adequate, as most sensitive inputs are actually entered by the user at an app's runtime: in our research, we inspect 17, 425 top apps from Google Play, and find that 35.46% of them involve sensitive user inputs. Manually marking them involves a lot of effort, impeding a large-scale, automated analysis of apps for potential information leaks. To address this important issue, we present *UIPicker*, an adaptable framework for automatic identification of sensitive user inputs. UIPicker is designed to detect the semantic information within the application layout resources and program code, and further analyze it for the locations where security-critical information may show up. This approach can support a variety of existing security analysis on mobile apps. We further develop a runtime protection mechanism on top of the technique, which helps the user make informed decisions when her sensitive data is about to leave the device in an unexpected way. We evaluate our approach over 200 randomly selected popular apps on Google-Play. UIPicker is able to accurately label sensitive user inputs most of the time, with 93.6% precision and 90.1% recall.

1 Introduction

Protecting the privacy of user data within mobile applications (*apps* for short) has always been at the spotlight of mobile security research. Already a variety of program analysis techniques have been developed to evaluate apps for potential information leaks, either dynamically [19, 23, 41] or statically [15, 26]. Access control mechanisms [27, 22, 33, 17] have also been proposed to enforce fine-grained security policies on the way that private user data can be handled on a mobile system. These techniques are further employed by mobile app marketplaces like Google Play (e.g., Bouncer [7]) to detect the apps that conduct unauthorized collection of sensitive user data.

Identifying sensitive user inputs. Critical to those privacy protection mechanisms is the labeling of sensitive user data. Some of the data are provided by the operating system (OS), e.g., the GPS locations that can be acquired through system calls like `getLastKnownLocation()`. Protection of such information, which we call *System Centric Privacy* data, can leverage relevant data-access APIs to set the security tags for the data. More complicated here is the content the user enters to a mobile app through its user interface (UI), such as credit-card information, username, password, etc. Safeguarding this type of information, called *User-Input Privacy* (UIP) data in this paper, requires understanding its semantics within the app, before its locations can be determined, which cannot be done automatically using existing techniques.

Just like the system-controlled user data (e.g., GPS), the private content entered through the UI is equally vulnerable to a variety of information-leak threats. It has been reported [5, 10, 4, 6] that adversaries can steal sensitive user inputs through exploiting the weaknesses inside existing protection mechanisms. For example, fraud banking apps to steal user's financial credentials with very similarity UIs. Besides, less security-savvy developers often inadvertently disclose sensitive user data, for example, transmitting plaintext content across public networks, which subjects the apps to eavesdropping attacks. Recent work further shows that side channels [18] and content-pollution vulnerabilities [42] can be leveraged to steal sensitive user inputs as well. In our research, we found that among 17,425 top Google-Play apps, 35.46% require users to enter their confidential information.

Given its importance, UIP data urgently needs protection. However, its technical solution is by no means trivial. Unlike system-managed user data, which can be easily identified from a few API functions, sensitive user inputs cannot be found without interpreting the context and semantics of UIs. A straightforward approach is to mark all the inputs as sensitive [15], which is clearly an overkill and will cause a large number of false positives. Prior approaches [43, 15, 36, 38, 40] typically rely on users, developers or app analysts to manually specify the contents within apps that need to be protected. This requires intensive human intervention and does not work when it comes to a large-scale analysis of apps' privacy risks.

To protect sensitive user inputs against both deliberate and inadvertent exposures, it is important to automatically recognize the private content the user enters into mobile apps. This is challenging due to the lack of fixed structures for such content, which cannot be easily recovered without analyzing its semantics.

Our work. To address this issue, we propose our research *UIPicker*, a novel framework for automatic, large-scale User-Input Privacy identification within Android apps. Our approach leverages the observation that most privacy-related UI elements are well-described in layout resource files or annotated by relevant keywords on UI screens. These UI elements are automatically recovered in our research with a novel combination of several natural language processing, machine learning and program analysis techniques. More specifically, UIPicker first collects a training corpus of privacy-related contents, according to a set of keywords and auto-labelled data. Then, it utilizes the content to train a classifier that identifies sensitive user inputs from an app's layout resources. It also performs a static analysis on the app's code to locate the elements that indeed accept user inputs, thus filtering out those that actually do not contain private user data, even though apparently they are also associated with certain sensitive keywords, e.g., a dialog box explaining how a strong password should be constructed.

Based on UIPicker, we further develop a runtime privacy protection mechanism that warns users whenever sensitive data leave the device. Using the security labels set by UIPicker, our system can inform users of what kind of information is about to be sent out insecurely from the device. This enables the user to decide whether to stop the transmission. UIPicker can be used by the OS vendors or users to protect sensitive user data in the presence of untrusted or vulnerable apps. It can be easily deployed to support any existing static and dynamic taint analysis tools as well as access control frameworks for automatic labeling of private user information.

To the best of our knowledge, UIPicker is the first approach to help detect UIP data in a large scale. Although the prototype of UIPicker is implemented for Android, the idea can be applied to other platforms as well. We implemented UIPicker based on FlowDroid [15] and built our identification model using 17,425 popular Google Play apps. Our evaluation of UIPicker over 200 randomly selected popular apps shows that it achieves a high precision (93.6%) and recall (90.1%).

Contributions. In summary, this paper makes the following contributions.

- We measure the distribution of UIP data based on 17,425 classified top free applications from different categories. The results show that in some categories, more than half of applications contain UIP data. Further protection of these UIP data is in urgent need.
- We propose UIPicker, a series of techniques for automatically identifying UIP data in large scale. Lots of existing tools can benefit from UIPicker for better privacy recognition in mobile applications.
- Based on UIPicker, we propose a runtime security enhancement mechanism for UIP data protection, which helps user to make informed decisions when such data prepare to leave the device with insecure transmission.
- We conduct a series of evaluation to show the effectiveness and precision of UIPicker.

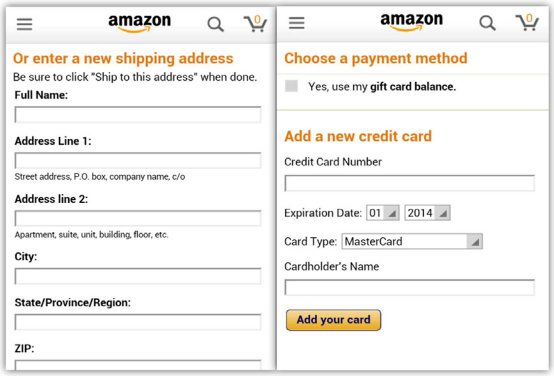
Roadmap. The rest of this paper is organized as follows. Section 2 gives the motivation, challenges and identification scope of UIP data, then introduces some background knowledge about Android layout resources. Section 3 gives an overview of UIPicker and illustrates the key techniques applied for identifying UIP data. Section 4 describes the identification approach step by step. Section 5 describes the runtime security enhancement framework based on UIPicker's identification results. Section 6 gives some implementation details about UIPicker. Section 7 gives evaluation and Section 8 discusses the limitation of UIPicker. Section 9 describes related work, and Section 10 concludes this work.

2 Problem Statement

In this section, we first provide a motivating example of users' sensitive input in two UI screens, then we investigate challenges in identifying such data and clarify our identification scope of UIP data. We also give some background knowledge about Android layout resources for further usage.

2.1 Motivating Example

Figure 1 shows two UI screens that contain some critical sensitive information in the Amazon Online Store [1]



amazon

Or enter a new shipping address

Be sure to click "Ship to this address" when done.

Full Name:

Address Line 1:

Street address, P.O. box, company name, c/o

Address line 2:

Apartment, suite, unit, building, floor, etc.

City:

State/Province/Region:

ZIP:

amazon

Choose a payment method

□ Yes, use my **gift card balance.**

Add a new credit card

Credit Card Number

Expiration Date: 01 2014

Card Type: MasterCard

Cardholder's Name

Add your card

(a) User's detailed address (b) User's credit card credential

Figure 1: Examples of User-Input Privacy (UIP) Data

app. In Figure 1(a), the user is required to input his/her detailed address for delivering products. Figure 1(b) requires user to input the credit card credential to accomplish the payment process. Many apps in mobile platform would require such sensitive data for various functional purposes. Most of such data are personal information that users are unwilling to expose insecurely to the public.

Although UIP data can be highly security-sensitive and once improperly exposed, could have serious consequences, little has been done so far to identify them at a large scale. The key issue here is how to automatically differentiate sensitive user inputs from other inputs. In our research, we check the top 350 free apps on Google Play, and find that on average each of them contains 11 fields across 6 UI screens to accept user inputs; however many of these fields do not accommodate any sensitive data. Static analysis tools like FlowDroid [15] only provide options to taint all user inputs as sensitive sources (e.g. *Element.getText()*). Analyzing in this way would get fairly poor results because sensitive user inputs we focus are mixed in lots of other sources we do not care. Such problem also exists in runtime protection on users' sensitive inputs. For example, in order to prevent sensitive user inputs insecurely leaking out, an ideal solution would be warning users when such data leave the device. Alerting all user inputs in this way would greatly annoy the users and reduce the usability because many normal inputs do not need to be treated as sensitive data.

2.2 Challenges

UIP data can be easily recognized by human. However, it is quite challenging for the machine to automatically identify such data with existing approaches in large-scale.

First, UIP data can not be identified through runtime monitoring. As these sensitive data are highly unstructured, they can not be simply matched by regex expressions when users input them. Besides, like any normal inputs, privacy-related inputs are sparsely distributed in various layouts in a single app, and most UI screens contain such private data require login or complex trigger conditions, which makes it very difficult for automatic testing tools like [8, 34] to traverse such UI screens exhaustively without manual intervention.

Identifying UIP data by traditional static analysis approaches is also impractical. In program code's semantic, sensitive input does not have explicit difference compared to normal input. Specifically, all of such input data can be accepted by apps, then transmitted out or saved in local storage in the same way, which makes it difficult to distinguish them through static analysis approaches.

UIPicker identifies UIP data in apps from another perspective, it analyzes texts describing sensitive inputs other than data themselves. This is because texts in UI screens usually contain semantic information that describes the sensitive input. Besides, layout description texts in layout files also contain rich semantic information to reveal what the specific element is intended to be in the UI screen by developers. UIPicker is primarily designed to help identify UIP data in *benign* apps. The identification results can be further used for security analysis or protection of users' sensitive data. Note that in this work we do not deal with malicious apps that intentionally evade our analysis, e.g., malware that constructs its layout dynamically or uses pictures as labels to guide users to input their sensitive data.

2.3 Identification Scope

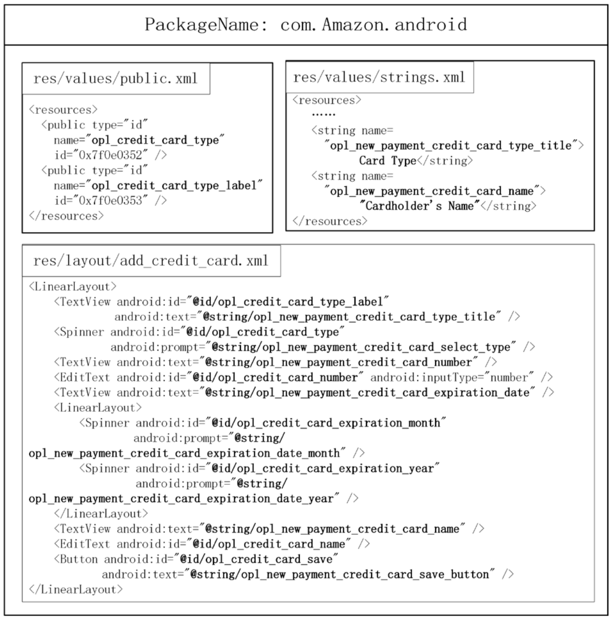
UIP data could be any piece of data that users consider to be sensitive from inputs. In the current version of UIPicker, we consider the following 3 categories as they cover most existing UIP data in current apps:

- **Account Credentials and User Profiles:** Information that reveals users' personal characters when they login or register, which includes but not limited to data such as username, user's true name, password, email address, phone number, birth date.
- **Location:** Plain texts that represent address information related to users. Different from system derived location (latitude and longitude), what we focus here is location data from users' input, e.g., the delivering address in shopping apps or the billing address for credit cards.
- **Financial:** Information related to users' financial activities, e.g., credit card number, expire date and security code.

The objective of UIPicker is to automatically identify such data from app resources in large-scale. Note that UIP data might not be limited to items listed here. UIPicker is capable of expanding its identification scope easily, as further discussed in Section 4.2.

2.4 Android Layout Background

Here we give some background knowledge about Android layout resources which UIPicker will use in our identification approach.



PackageName: com.Amazon.android

res/values/public.xml

```
<resources>
  <public type=″id″
    name=″opl_credit_card_type″
    id=″0x7f0e0352″ />
  <public type=″id″
    name=″opl_credit_card_type_label″
    id=″0x7f0e0353″ />
</resources>
```

res/values/strings.xml

```
<resources>
    ……
    <string name=
      ″opl_new_payment_credit_card_type_title″>
          Card Type</string>
    <string name=
      ″opl_new_payment_credit_card_name″>
          ″Cardholder's Name″</string>
</resources>
```

res/layout/add_credit_card.xml

```
<LinearLayout>
    <TextView android:id=″@id/opl_credit_card_type_label″
              android:text=″@string/opl_new_payment_credit_card_type_title″ />
    <Spinner android:id=″@id/opl_credit_card_type″
             android:prompt=″@string/opl_new_payment_credit_card_select_type″ />
    <TextView android:text=″@string/opl_new_payment_credit_card_number″ />
    <EditText android:id=″@id/opl_credit_card_number″ android:inputType=″number″ />
    <TextView android:text=″@string/opl_new_payment_credit_card_expiration_date″ />
    <LinearLayout>
        <Spinner android:id=″@id/opl_credit_card_expiration_month″
                 android:prompt=″@string/
opl_new_payment_credit_card_expiration_date_month″ />
        <Spinner android:id=″@id/opl_credit_card_expiration_year″
                 android:prompt=″@string/
opl_new_payment_credit_card_expiration_date_year″ />
    </LinearLayout>
    <TextView android:text=″@string/opl_new_payment_credit_card_name″ />
    <EditText android:id=″@id/opl_credit_card_name″ />
    <Button android:id=″@id/opl_credit_card_save″
            android:text=″@string/opl_new_payment_credit_card_save_button″ />
</LinearLayout>
```

Figure 2: Android Layout Description Resources

Layout resources define what will be drawn in the UI screen(s) of the app. In Android, a User Interface is made up of some basic elements (e.g., *TextView*, *EditText*, *Button*) to display information or receive input. Android mainly uses XML to construct app layouts, thus developers can quickly design UI layouts and screen elements they wish to contain, with a series of elements such as buttons, labels, or input fields. Each element has various attributes or parameters which are made up of name-value pairs to provide additional information about the element.

Android layout resources are distributed in different folders in the app package. Layout files for describing UI screens are located in folder *res/layout*. The unique hex digit IDs for identifying each element in layout files are in *res/value/public.xml* and texts showed in UI screens to users are in *res/values/strings.xml*. Resources in *res/values/* are referenced by texts with specific syntax (e.g. *@String, @id*) in the layout files in *res/layout* for ease of development and resource management.

Figure 2 shows some layout resources used for constructing the UI in Figure 1(b). The entry is a layout file named *add_credit_card.xml*. It contains two *EditText* elements to accept the credit card number and the card holder's name, three *Dropdown list* elements (named as *spinner* in Android) to let user select card type and expiration date. In the *EditText* for requesting the card number, it uses *@id/opl_credit_card_number* to uniquely identify this element for the app. Syntax like *android:inputType=number* suggests that this *EditText* only accepts digital input. There is also a *TextView* before *EditText* with attribute *android:text=@string/opl_new_payment_credit_card_number*, which means the content showed in this label will be string referenced to *opl_new_payment_credit_card_number* in */res/values/stings.xml*.

3 System Overview

In this section, we give an overview of UIPicker and describe the key techniques applied in our identification framework.

Overall Architecture. Figure 3 shows the overall architecture of UIPicker. UIPicker is made up of four components to identify layout elements which contain UIP data step by step. The major components can be divided into two phases: model-training and identification. In the model-training phase (Stage 1,2,3), UIPicker takes a set of apps to train a classifier for identifying elements contain UIP data from their textual semantics. In the identification Phase (Stage 1,3,4), UIPicker uses both the trained classifier (Stage 3) and program behavior (Stage 4) to identify UIP data elements.

Pre-Processing. In the Pre-Processing module, UIPicker extracts the selected layout resource texts and reorganizes them through natural language processing (NLP) for further usage. This step includes word splitting, redundant content removal and stemming for texts. Pre-Process can greatly reduce the format variations of texts in layout resources caused by developers' different coding practice.

Privacy-related Texts Analysis. For identifying UIP data from layout resources, the first challenge is how to get privacy-related texts. One can easily come up with a small set of words about UIP data, but it is very difficult to get a complete dictionary to cover all such semantics. In our case, leveraging an English dictionary like WordNet [14] for obtaining semantically related words is limited in the domain of our goals. Many words that are semantically related in privacy may not be semantically related in English, and many words that are semantically related in English may not appear in layout resource texts as well. For example, both "signup" and "register" represent to create a new account in an app's login screen, but

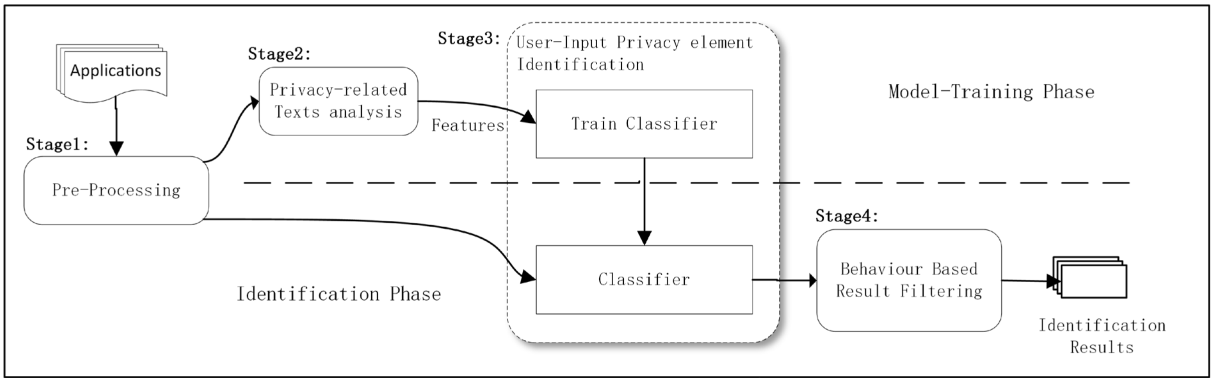


Figure 3: System Overview of UIPicker

they can not be correlated from a dictionary like WordNet.

Besides, UIP data in apps are often described with a single word or very short phrases (e.g. "password" or "input password") in layout resources. Due to the lack of complete sentences describing UIP data, natural language processing techniques [35] like dependency relation pattern extraction is not suitable in our scenario.

UIPicker expands UIP semantic texts with a few privacy-related seeds based on a specific feature extraction approach. It first automatically labels a subset of layouts which could contain UIP data by heuristic rules, then extracts privacy-related semantics from such layouts by applying clustering algorithms. It helps us to automatically extract privacy-related texts with little manual effort. As a result, these inferred texts can be used as features for identifying whether an element is privacy-related or not in the next step.

UIP Data Element Identification. Based on the given set of privacy-related textual semantics from the previous step, to what extent an element contains privacy-related texts can be identified as sensitive? As previous work [37] showed, purely relying on keyword-based search would result in a large number of false positives. For example, sensitive item "username" could always be split into "user" and "name" as two words in apps, and none of the single word can represent "username". Besides, certain words like "address" have a confounding meaning. For instance, such phrase showed in a layout screen "address such problem" does not refer to location information.

In this step, UIPicker uses a supervised machine learning approach to train a classifier based on a set of semantic features generated in the previous stage. Besides, it fully takes the element's context in the whole layout into consideration for deciding whether the element is privacy-related or not. With this trained model, for any given layout element with description texts, UIPicker can tell whether it is related to UIP from its textual semantics.

Behavior Based Result Filtering. Besides identifying elements that contain UIP data from their textual semantics, we also need to check whether a privacy-related element is actually accepting user input. In other words, we need to distinguish user inputs from other static elements such as buttons or labels for information illustration in layout screens. Although Android defines *EditText* for accepting user input, developers can design any type of element by themselves (e.g. customized input field named as *com.abc.InputBox*). Besides, apps also receive user inputs in an implicit way through other system defined elements without typing the keyboard by users. For example, in Figure 1(b), the expire date of credit card is acquired by selecting digits from the *Spinner* element.

We observe that for each privacy-related element identified by UIPicker in the previous stage, the data should be acquired by the app with user's consent if it is actually accepting user input. For example, the user clicks a button "OK" to submit data he/she inputs. When reflected in the program code, the user input data should be acquired by the system under certain event trigger functions. We use static code analysis to check whether an arbitrary element can be matched with such behavior, thus filter out irrelevant elements we do not expect.

4 IDENTIFICATION APPROACH

In this section, we explain the details of four stages in UIPicker's identification approach.

4.1 Stage 1: Pre-Processing

Resource Extraction. We first decode the Android APK package with apktool [2] for extracting related resource files we need. Our main interest is in UI-related content, thus for each app, we extract **UI Texts** and **Layout Descriptions** from its decompiled layout files.

Layout Resource	Sample
UI Texts	Add a new credit card, Credit Card Number Expiration Date, Card Type, Cardholder's name
Layout Descrioptions	@id/opl_credit_card_number @string/opl_new_credit_card_expiration_date_month @string/opl_new_credit_card_save_buton

Table 1: Selected resources of Amazon Online's "Add Credit Card" screen

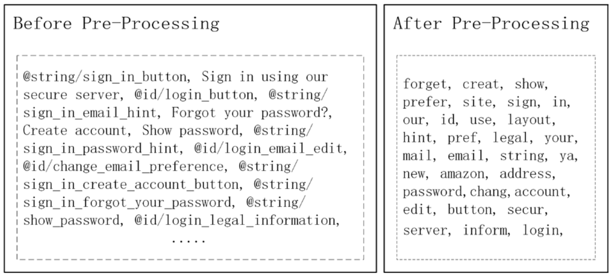
- **UI Texts**. UI texts are texts showed to users in the layout screen. In Android, most of such texts are located in */res/values/strings.xml* and referenced by syntax *@String/[UI text identifier]* in element's attribute. Some *UI texts* are directly written in layout files as attribute values of UI elements, e.g., *android:hint='Please input your home address here'*.
- **Layout Descriptions**. Layout Descriptions are texts only showed in layout files located in */res/layout/*. For these texts, we consider all strings starting with syntax @*id* and @*String* to reflect what the element is intended to be from their textual semantics.

The main difference between *UI texts* and *layout descriptions* is that *UI texts* are purely made up of natural language while *layout descriptions* are mainly name identifiers (both formatted and unformatted) with semantic information. As developers have different naming behaviors when constructing UIs, in most cases, semantic information in *layout descriptions* is more ambiguous than that in *UI texts*.

We extract these groups of resources for further analysis because these selected targets can mostly reflect the actual content of the app's layout. For example, the selected resources about Amazon's "Add Credit Card" screen in Figure 1(b) are showed in Table 1.

Word Splitting. Although most of *layout descriptions* are meaningful identifiers for ease of reading and program development, normally they are delimiter-separated words or letter-case separated words. For example, "phone number" can be described as "phone_number" or "PhoneNumber". Thus we split such strings into separated word sets. Besides, some of *layout descriptions* are concatenated by multiple words without any separated characters. For these data, we split them out by iteratively matching the maximum length word in WordNet [14] until the string cannot be split any more. For example, string "confirmpasswordfield" will be split into "confirm", "password", and "field".

Redundant Content Removal. For all *UI texts* we extracted, we remove non-English strings through encoding analysis. For each word, we also remove non-text characters from all extracted resources such as digits, punctuation. After this, we remove stop words. Stop words are some of the most common words like "the", "is", "have". We remove such contents because they can not provide meaningful help in our analysis process.



```
Before Pre-Processing

@string/sign_in_button, Sign in using our
secure server, @id/login_button, @string/
sign_in_email_hint, Forgot your password?,
Create account, Show password, @string/
sign_in_password_hint, @id/login_email_edit,
@id/change_email_preference, @string/
sign_in_create_account_button, @string/
sign_in_forgot_your_password, @string/
show_password, @id/login_legal_information,
                  .....
```

```
After Pre-Processing

forget, creat, show,
prefer, site, sign, in,
our, id, use, layout,
hint, pref, legal, your,
mail, email, string, ya,
new, amazon, address,
password, chang, account,
edit, button, secur,
server, inform, login,
```

Figure 4: After Pre-Processing, texts in left are transformed into formats in right

Stemming. Stemming is the process for reducing inflected (or sometimes derived) words to their stem, base or root form. Stemming is essential to make words such as "changed", "changing" all match to the single common root "change". Stemming can greatly improve the results of later identification processes since they reduce the number of words in our resources. We implement *Porter Stemmer* [11] with python NLTK module [9].

Figure 4 shows part of texts before and after preprocessing for Amazon's "Add credit card" layout file. As we can see, all texts concatenated by '_' are split into separated words, "edthomephonecontact" is split into "edt", "home", "phone" and "contact" instead. We also transform words like "forgot", "forget" into a single uniformed format as "forget".

4.2 Stage 2: Privacy-related Texts Analysis

In this stage, we use Chi-Square test [39] to extract privacy-related texts from a subset of specific layouts. The intuition here is that privacy-related words prefer to be correlated in specific UIs such as the login, registration or settings page of the app. If some words appear together in these UI, they are likely to have semantic relevance to users' sensitive information. Thus, we use such layouts to extract privacy-related texts in contrast to other normal layouts.

Chi-Square Based Clustering. Chi-Square (Chi^2) test is a statistical test that is widely used to determine whether the expected distributions of categorical variables significantly differ from those observed. Specifically in our case, it is leveraged to test whether a specific term on UI screens is privacy-related or not according to its occurrences in two opposite datasets (privacy-related or non privacy-related).

Here we choose *UI texts* rather than *layout descriptions* to generate privacy-related texts due to the follow-

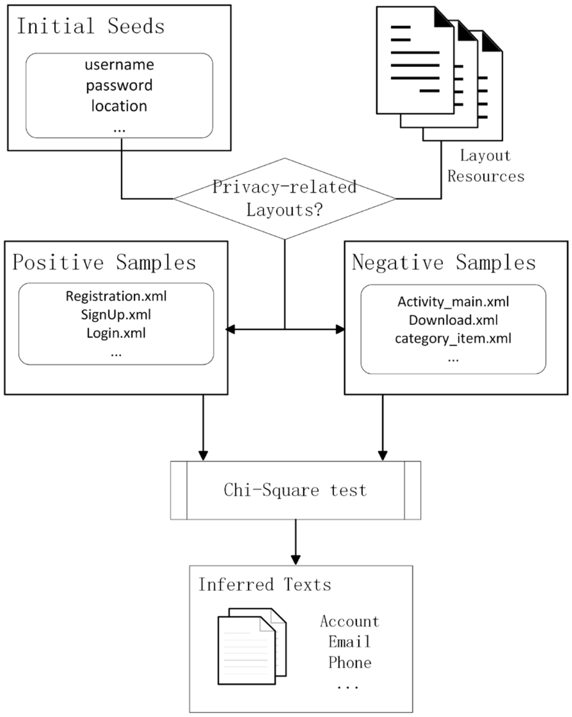


Figure 5: For each word assigned as the initial seed, UIPicker calculates its Chi-Square test for each word in positive samples and appends part of its top results into the privacy-related texts feature set.

ing two reasons: First, *layout descriptions* are not well structured as the naming behaviors vary very differently between apps (or developers), while *UI texts* are in a relatively uniformed format, thus making it easy to extract privacy-related texts from them. For example, a layout requesting a user's password must contain term "password" in the UI screen, while in layout descriptions it could be text like "pwd", "passwd", "pass". Second, as layout descriptions aim for describing layout elements, it may contain too much noisy texts like "button", "text" which would bring negative impact to the privacy-related text extraction.

Figure 5 shows how UIPicker generates privacy-related texts. First, we give a few words that can explicitly represent users' sensitive input we focus (e.g., email, location, credit card), and we call them *initial seeds*. Each layout sample is made up of a set of *UI texts* in its layout screen. Then, the initial seeds will be used to identify whether a specific layout sample is privacy-related or not based on the following two patterns:

- Logical relation between sensitive noun phrase (initial seed) and verb phrase, e.g., the pair (save, password).
- Logical relation between possessive (mainly word "your") and sensitive noun phrase (initial seed), e.g., the pair (your, username).

As such patterns strongly imply actions that the app is requesting the user's sensitive input, for those layout samples satisfying one of these two patterns, we label them as privacy-related (positive samples). On the other hand, for layout samples that do not contain any of texts in the pattern (both noun phrase and verb, possessive phase), we label them as negative samples. Note that we do not label those layouts only containing initial seeds as positive or negative because a single word is insufficient for us to identify whether the layout is privacy-related or not.

Based on the two classified sample sets, for all distinct words appearing in positive samples, we use Chi-Square test and rank their results in a descending order. As a result, texts with higher Chi-Square scores mean they are more representative as privacy-related, which can easily be picked up from the top-ranked words in the test results.

The following example explains our analysis approach for finding financial-related textual semantics. We set "credit card" as an initial seed, then the layout shown in Figure 1(b) will be identified as a positive sample because both "credit card" and verb phrase "add" are included in this layout. Thus in our dataset, other similar layouts will be labeled as positive if it requires users to input credit card information as well. As a result, the positive sample will include more texts such as "expire", "date", "year", "month", which are also related to financial credentials and ranked in top of the Chi-Square test results.

Noisy Text Removal. Although Chi-Square test aims to cluster privacy-related texts, it still unavoidably introduces some irrelevant texts into its clustering results. This is mainly because not all texts in privacy-related layout are necessarily related to privacy. In order to generate a highly precise cluster of privacy-related texts to eliminate false positives in the UIP data element identification process, we introduce a little manual effort here for filtering out such irrelevant texts from the clustering result. Since Chi-Square test already helps us extract texts that are most probably related to privacy, looking through such a list is quite simple and effortless.

Alternative Approaches. In our research, we compared the Chi-Square test with two popular alternatives, frequency based text extraction and TF-IDF [31], both of which are found to be less effective. They all bring in more irrelevant contents than the Chi-Square test, more susceptible to the limitation of the layout level samples, that is, privacy related UI screens often contain a lot of normal texts, which become noises in our sensitive term identification. Also, using these two approaches, we need to continuously adjust their thresholds for selecting privacy-related text when their sample sizes change. This can be avoided when using the Chi-Square test. Nevertheless, we acknowledge that there may exist other feature extraction mechanisms that could perform better, which is one of our future work.

4.3 Stage 3: UIP Data Element Identification

In this stage, we explain the details of our machine-learning approach, which automatically identifies UIP data elements based on their textual semantics. UIPicker uses supervised learning to train a classifier based on a subset of element samples with privacy-related semantic features. As a result, for a given unclassified UI element, this step could identify whether it is semantically privacy-related from its description texts.

Feature Selection. We use privacy-related texts inferred from the previous stage as features for the identification module. A single word alone usually does not provide enough information to decide whether a given element is privacy-related. However, all such features in combination can be used to train a precise classifier. The main reason why these features work is that both *UI texts* and *layout descriptions* do in fact reveal textual semantic information, which a machine learning approach such as ours can discover and utilize. Note that in *layout descriptions*, it is often the case that developers use text abbreviations for simplicity when naming identifiers. For example, “address” in layout descriptions could be “addr”. For this, we construct a mapping list of such texts we visited during the manual analysis. Thus, for each word in layout descriptions, we transform the abbreviation into complete one if it is contained by any privacy-related texts.

Besides, we also take semantic features of layout structure into consideration: the texts of this element’s siblings. We observe that many elements are described by texts in its siblings. For example, In Figure 1(b), most of input fields are described by static labels which contain privacy-related text as instructions for requesting user inputs. As a result, texts from sibling elements can bring more semantic information for better identification results.

The classifier works on a matrix organized by one column per feature (one word) and one row per instance. The dimension for each instance is the size of our feature set (the number of texts from the previous step). The additional column indicates whether or not this instance is a privacy-related element.

Training Data. Since text fields can have different input types for determining what kind of characters are allowed inside the field, Android provides the *android:inputType* attribute to specify what kind of characters are allowed for *EditText*. For example, an element with *inputType* valued *textEmailAddress* means only email address is accepted in this input field. There are several input types explicitly reflect the element containing UIP data we focus on, which can be used as the training data of the identification module. We list such sensitive attribute values[1] in the first column of Table 2.

Privacy Category	Attribute Value
Account Credentials & User Profile	textEmailAddress textPersonName textPassword textVisiblePassword password/email/phoneNumber
Location	textPostalAddress

Table 2: Sensitive attribute values in layout descriptions

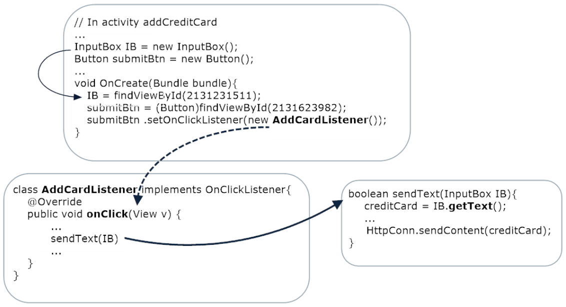
The training data is constructed as follows: First, we automatically label all elements with sensitive attributes as positive samples since they are a subset of UIP data elements. We further manually label a set of elements involving financial information from the category “Financial” because such elements are covered by sensitive attributes Android provides. Besides, a set of negative samples are picked out through human labeling after filtering out the elements that contain any of the privacy-related texts we generated in Stage 2.

Classifier Selection. We utilize the standard support vector machine (SVM) as our classifier. SVM is widely used for classification and regression analysis. Given a set of training examples with two different categories, the algorithm tries to find a hyper-plane separating the examples. As a result, it determines which side of hyper-plane the new test examples belong to. In our case, for an unclassified unknown layout element with corresponding features (whether or not containing privacy-related texts extracted in the previous step) the classifier can decide whether it contains UIP data or not from its textual semantics.

4.4 Stage 4: Behavior Based Result Filtering

As a non-trivial approach for identifying UIP data, for each element identified as privacy-related from its layout descriptions, UIPicker inspects the behaviors reflected in its program code to check whether it is accepting user inputs, thus filtering out irrelevant elements from the identification results in the previous step.

[1] In some older apps, developers also use specific attribute like “android:password=True” to achieve the same goal as *inputType*. We list them in Table 2 and call them sensitive attribute values as well for simplicity.



```
// In activity addCreditCard
...
InputBox IB = new InputBox();
Button submitBtn = new Button();
...
void OnCreate(Bundle bundle){
  IB = findViewById(2131231511);
  submitBtn = (Button)findViewById(2131623982);
  submitBtn .setOnClickListener(new AddCardListener());
}
```

```
class AddCardListener implements OnClickListener{
   @Override
   public void onClick(View v) {
        ...
        sendText(IB)
        ...
   }
}
```

```
boolean sendText(InputBox IB){
   creditCard = IB.getText();
   ...
    HttpConn.sendContent(creditCard);
}
```

Figure 6: Sample codes for requesting a user's credit card number

User input data is generated based on a user's interactions with the app during runtime. In other words, the data will be acquired by the app under the user's consent. In Android, to get any data from a UI screen is achieved by calling specific APIs. Getting such data under user consent means these APIs are called under user-triggered system callbacks. For example, code fragments in Figure 6 shows the behavior reflected in the program when the app gets the user's credit card number in Figure 1(b). Here, the input field *IB* is defined by *IB=findViewById(21...1)* in activity *addCreditCard*. When the user clicks the "Add your card" button, in the program code, the *OnClick()* function in class *AddCardListener()* will be triggered by pre-registered system callback *submitBtn.setOnClickListener()*. Then, it invokes *sendText(IB)*, which sends the inputBox's object by parameter, and finally gets the user's card number by *IB.getText()*. One might consider why don't catch UIP data simply by checking whether the element is invoked by *getText()* API. The reason is that sometimes developers may also get values from UI screens like static text labels as well as user inputs, resulting in false negatives for our identification approach.

5 Runtime Security Enhancement with UIPicker

The security implications about UIP data are rooted from the fact that users have to blindly trust apps when they input sensitive data. With the help of UIPicker differentiating UIP data from other normal inputs, we can use taint tracking techniques to trace users' sensitive inputs and enable users to make informed decisions with a popup window when such data insecurely leave the device, thus effectively mitigating the potential threats posed by apps.

For UIP data, we consider the following two situations as insecure and should inform users to let them decide whether to proceed or not.

Plain Text Transmission. We consider any piece of UIP data should not be transmitted in plain text. Such situation can be easily identified by checking if the tainted sink is HTTP connection in runtime.

Insecure SSL Transmission. Previous works [34] showed that a large number of apps implement SSL with inadequate validations (e.g., app contains code that allows all hostnames or accepts all certificates). Insecure SSL transmission could be more dangerous because they may carry over critical sensitive data in most cases. UIP data should not be transmitted in this way as well.

Since UIPicker is deployed in off-line analysis by customized system vendors, one can also check whether the apps have securely implemented SSL off-line at the same time. We integrate a static analysis framework named MalloDroid [20] with UIPicker to automatically check SSL security risks by evaluating the SSL usage in apps. As MalloDroid can only find broken SSL usage regardless what data is transmitted via this channel, we also use FlowDroid to check if there exists data/control flow intersections between UIP data sources and SSL library invocation sinks in the app, thus confirming whether the UIP data in the app will be transmitted with security risks.

6 IMPLEMENTATION

Dataset. We crawled apps from Google Play Store based on its pre-classified 35 categories in Oct. 2014. For each category, we downloaded the top 500 apps. Excepting some connection errors occurred in the crawling process, totally we collected 17,425 apps as our dataset. This dataset will be used in both model training and evaluation of UIPicker.

Identification Approach. We implement the prototype of UIPicker as a mix of Python Scripts and Java code. The first three steps of UIPicker are developed using Python with 3,624 lines of code (LOC). The last step, static analysis for result filtering, is implemented in Java, which extends FlowDroid[15] and introduces additional 985 LOCs. All experiments are performed on a 32 core Debian server with Linux 2.6.32 kernel and 64GB memory.

For privacy-related text analysis, the initial seeds are assigned as texts in the second column of Table 3 for each privacy category. For each initial seed, we run the Chi-Square test using apps in our dataset. Since Android allows developers to use nested layout structures for flexibility, we also group sub-layout *UI texts* into their root layouts. For each round, we collect the top 80 words from the test results, this threshold is determined by balancing between the number of privacy-related terms that can be detected and the amount of noisy text introduced. After 7 (7 initial seeds) rounds of the Chi-Square test, we collect 273 words from layout samples (some texts are overlapped in different round of Chi-Square test). We then remove 45 words as the noisy text by manual analysis within less than 3 minutes. As a result, UIPicker extracts 228 privacy-related terms from 13,392 distinct words. We list part of them in the third column of Table 3 corresponding to the privacy category they belong to. Such data are used as features for privacy-related element identification in the follow-up step.

Privacy Category	Initial Seeds	Representative Inferred Texts(Stemmed)
Login Credenticals &User Profile	username, password, email	mobil phone middl profile cellphon account nicknam firstnam lastnam person birth login confirm detail regist
Location	address, location	zip citi street postal locat countri
Financial	credit card, bank	secur month date pay year bill expir debit transact mm yy pin code

Table 3: Initial seeds and part of inferred privacy-related texts from Chi-Square test

The SVM classifier is implemented with scikit-learn[13] in poly kernel. We optimize the classifier parameters (gamma=50 and degree=2) for performing the best results.

For each element identified as privacy-related by the machine learning classifier, UIPicker conducts static taint analysis using FlowDroid[15] to check whether it satisfies specific behavior described in Section 4.4. Since FlowDroid successfully handles android life cycle (system event based callbacks) and UI widgets, the dataflow results should be both precise and complete. We set FlowDroid's layout mode as "ALL" to get each element's propagation chain that starts with function *findViewById([elementId])* and ends in *getText()*. As a result, for any element's info-flow path which contains system event function like *OnClick()*, the element can be identified as accepting user input.

Runtime Enhancement For each app, we use a list of elements containing UIP data identified from UIPicker with their unique IDs as the taint sources of TaintDroid[19] build in Android 4.1. Since TaintDroid allows 32 different taint markings through a 32-bit bitvector to encode the taint tag, for those UIP data elements involved in insecure SSL usage, we label them as "SSL Insecure" in the taint source list, thus provide warnings to users when such data leave the device as well. We add a pop-up window for showing the leaked information to users when sensitive data leave the device. Our modification to TaintDroid is implemented with 730 LOCs in total.

7 Evaluation

In this section, we present our evaluation results. We first show the performance of UIPicker in Section 7.1, then we discuss its effectiveness and precision in Section 7.2 and Section 7.3. Then we evaluate our runtime security enhancement mechanism in Section 7.4.

7.1 Performance

During our experiment, the training phase of the classifier takes about 2.5 hours on average, the identification phase for the whole dataset takes 30.5 hours (6.27 seconds per app). Pre-Processing time for apps is included in both of these two phases. The static analysis for behaviour based result filtering is proceeded in 32 threads concurrently. Since UIPicker mainly targets for customized system vendors or security analysts, we consider such overhead quite acceptable.

7.2 Effectiveness

UIP Data Distribution. We show the general identification results of UIPicker in Table 4. In 17,425 apps, UIPicker finds that 6,179 (35.46%) contain UIP data. We list our results in a descending order of the identified total app amounts. As we can see, in 9 out of 35 categories, more than half of apps contain UIP data.

We make the following observations from this table. First, application categories such as BUSINESS, FINANCE, SHOPPING, COMMUNICATION and SOCIAL are more likely to request Account Credentials and User Profile information, which showed that these apps are closely related to users' personal activities. APP_WIDGETS (54.08%) is also ranked among top of the table. It is a set of apps which have small UIs embedded in the home screen of the device, e.g., Facebook, Youtube, Twitter. Since most of such apps provide login and account-specific functions, they prefer to request more UIP data as well. The SHOPPING category contains many location-related elements (1,605, 37%) because the delivering address are always generated from user inputs. It is also reasonable that both FINANCE and SHOPPING apps require many financial-related sensitive inputs. We believe such apps containing rich UIP data should be treated more carefully in both developing and security vetting process in order to make sure that sensitive data are well protected in both transmission and storage.

Comparative Results. We illustrate the effectiveness of UIPicker from two aspects. First, UIPicker identifies privacy data that system defined APIs do not touch but still be sensitive to users. Second, UIPicker achieves far better coverage than simply identifying UIP data by specific sensitive attribute values from the Android design specification.

Comparison with System Defined Sensitive APIs. As previously mentioned, specific sensitive resources

Application Category	Account Credentials & User Profile		Location		Financial		Total	
	#element	%app	#element	%app	#element	%app	#element	%app
BUSINESS	4,314	61.52%	1,112	38.28%	399	18.04%	5,825	62.73%
WEATHER	1,102	46.18%	1,086	59.24%	32	3.01%	2,220	62.45%
FINANCE	4,821	50.90%	1,106	33.47%	1,815	30.46%	7,742	55.31%
COMMUNICATION	2,756	53.83%	439	21.77%	213	14.31%	3,408	55.24%
SHOPPING	3,380	51.80%	1,605	37.00%	609	24.80%	5,594	54.60%
APP_WIDGETS	3,161	51.22%	816	31.43%	352	15.71%	4,329	54.08%
NEWS_AND_MAGAZINES	1,994	47.38%	529	34.68%	133	12.50%	2,656	54.03%
SOCIAL	2,889	52.62%	555	27.42%	146	8.27%	3,590	54.03%
TRAVEL_AND_LOCAL	2,826	49.00%	1,494	41.16%	452	16.87%	4,772	52.21%
PRODUCTIVITY	1,923	45.45%	394	18.59%	113	9.29%	2,430	48.69%
LIFESTYLE	2,243	43.29%	853	28.66%	341	14.03%	3,437	45.29%
TRANSPORTATION	1,634	39.00%	750	28.60%	273	11.00%	2,657	44.60%
SPORTS_GAMES	2,023	41.70%	509	22.67%	151	6.68%	2,683	43.32%
MEDICAL	1,478	40.04%	302	15.49%	169	7.04%	1,949	40.24%
HEALTH_AND_FITNESS	1,795	39.56%	344	15.06%	165	8.43%	2,304	39.96%
MEDIA_AND_VIDEO	1,079	37.15%	170	13.05%	72	3.61%	1,321	38.55%
TOOLS	1,110	36.36%	252	16.16%	121	8.08%	1,483	38.38%
MUSIC_AND_AUDIO	1,053	37.20%	219	11.40%	91	3.20%	1,363	38.00%
PHOTOGRAPHY	1,008	26.65%	205	9.82%	122	5.21%	1,335	28.46%
ENTERTAINMENT	973	27.71%	249	9.24%	215	5.62%	1,437	28.31%
BOOKS_AND_REFERENCE	924	26.80%	213	9.80%	156	5.60%	1,293	27.40%
EDUCATION	1,753	20.68%	461	9.84%	83	5.02%	2,297	21.69%
COMICS	390	16.60%	84	4.00%	69	3.00%	543	17.20%
PERSONALIZATION	440	16.23%	77	3.85%	32	1.83%	549	16.43%
CARDS	360	14.20%	40	3.20%	58	4.60%	458	15.80%
GAME_WIDGETS	302	13.25%	17	2.01%	56	4.42%	375	13.45%
ARCADE	390	12.22%	66	3.61%	24	0.80%	480	12.42%
LIBRARIES_AND_DEMO	302	10.84%	89	3.61%	136	3.01%	527	11.24%
GAME_WALLPAPER	242	11.00%	21	2.00%	55	4.20%	318	11.00%
BRAIN	396	10.60%	102	4.00%	71	2.20%	569	10.80%
GAME	302	9.82%	53	3.81%	16	0.80%	371	10.22%
SPORTS	209	10.22%	26	1.40%	15	0.80%	250	10.22%
CASUAL	267	9.60%	23	2.60%	10	0.40%	300	9.60%
APP_WALLPAPER	187	6.25%	34	2.42%	20	1.61%	241	6.65%
RACING	82	4.60%	16	0.40%	20	0.60%	118	4.60%
TOTAL	50,108	30.59%	14,311	16.26%	6,805	7.57%	71,224	35.46%

Table 4: UIP data distribution. #element denotes the number of UIP data elements in each category by different privacy type. %app denotes the percentage of apps in which these elements appear (500 per category). The last column shows the total number of UIP data elements and apps that contain UIP data.

Privacy Category	Android System Defined APIs
Account Credentials & User Profile	android.tel...TelephonyManager getLine1Number() android.accounts.AccountManager getAccounts()
Location	and...LocationManager getLastKnownLocation() android.location.Location: getLongitude() android.location.Location: getLatitude()

Table 6: System defined sensitive APIs related to UIPicker's identification scope

such as phonenumber, account and location can be regulated by fixed system APIs which we list in Table 6. We compare the amount of UIPicker's identification results with Android system derived sensitive data, which can help us understand to what extent, system defined sensitive APIs are insufficient to cover users' privacy.

As Table 5 shows, in our dataset, 4,900 apps use system defined APIs for requesting Account Credentials and Profile Information while UIPicker identifies 5,330 (30.59%) apps containing UIP. UIPicker identifies 2,883 (16.26%) apps in the whole dataset that request location privacy data from user inputs. Besides, 1,318 (7.57%) apps request financial privacy data from users, and none of system defined APIs can regulate such data. In general, UIPicker identifies 6,179 (35.46%) apps containing at least one category of UIP data, which have been largely neglected by previous work in privacy security analysis and protection.

As Column 4 in Table 5 shows, there is some overlap between system defined APIs and UIP data (1,340 for Account Credentials & User Profile, 2,282 for Location respectively). For each app, we check whether it contains both the system defined APIs and the UIP data in the same privacy category, e.g., invoking the getLastKnownLocation() API and requesting address information from the user input of the same app. In some cases, the same piece data may come from either UI input or API call. For example, using a phone number as the login account of the app. However in most cases, the overlapped data in the same privacy category may come from different sources without overlapping in code paths. For example, the invocation of get-location APIs is used for realtime geographic locating, while some location input could be

Privacy Category	System Defined APIs (#Apps)			Elements with Sensitive Attribute Values (#Elements)		
	API	UIPicker	Overlap	InputType	UIPicker	Incremental
Account Credentials & User Profile	4,900	5,330	1,340	24,021	46,227	26,087
Location	15,221	2,883	2,282	941	14,311	13,370
Financial	-	1,318	-	-	6,353	-
Total	15,632	6,179	-	24,962	71,224	46,262

Table 5: We compare UIPicker's identification results with apps containing system defined sensitive APIs (column 2-4) and elements containing sensitive attribute values (column 5-7).

a shopping address for delivering goods. Since precisely analyzing which input element may overlap with system defined APIs requires additional information-flow analysis, which is beyond this paper's scope, we leave it as future work for measuring the relationship between these two types of sensitive data.

Comparing to sensitive attribute values. In Section 4.3, we use elements containing sensitive attribute values as part of training data for our identification module. However, they can only cover a portion of UIP data because they are not intended for this purpose. Here we compare the amount of UIPicker's identification results with elements containing sensitive attribute values to show the effectiveness of UIPicker.

As Table 5 shows, in general, UIPicker identifies 46,262 more UIP data elements than simply identifying them by sensitive attribute values (e.g. textPassword). Especially for the Location category, UIPicker identifies 14,311 elements, which is nearly 15 times more than simply identifying them based on attribute "textPostalAddress".

Types of UIP Elements. We list the identification results of UIP data elements other than *EditText* in Table 7. In general, UIPicker finds 18,403 (25.84%) elements other than *EditText* to accept users' sensitive inputs. It is interesting to note that UIPicker also finds a large portion of *TextView* as UIP data elements. In most cases, although data in *TextViews* are not editable, they could be generated by users from other layouts and dynamically filled in *TextView* later. For example, the data from previous steps of a registration form, or fetched from the server after users' login. There are 5,075 (7.13%) customized input elements and 1,962 (2.75%) dropdown lists (*Spinners*) containing UIP data. Type "Others" in table contains elements such as *RadioButton*, *CheckBox*.

7.3 Precision

For evaluating the precision of UIPicker, we perform the evaluation of classifier based on the machine-leaning dataset mentioned in Section 4. We also conduct a manual validation for two reasons. First, since the training data of classifier is not absolutely randomly selected (part of them are labeled by sensitive attributes automatically), a manual validation is required to confirm that the identification results of the classifier carries over the entire dataset. Second, the classifier is only capable of distinguishing UIP data elements from their textual semantics, the manual validation can be used to check whether static text labels are effectively excluded by UIPicker after behaviour based result filtering.

Type	# Elements	% in UIP Data
TextView	10,582	14.86%
Customized	5,075	7.13%
Spinner	1,962	2.75%
Others	784	1.10%
Total	18,403	25.84%

Table 7: Types of UIP Elements Other than EditText

Evaluation of Classifier. The training set contains 53,094 elements in total, which includes 24,962 labeled by sensitive attribute values and financial-related elements, with 25,331 negative samples labeled by manual efforts.

We use ten-fold cross validation which is the standard approach for evaluating machine-learning classifiers. We randomly partition the entire set of sample elements into 10 subsets, and we train the classifier on nine of them and then test the remaining 1 subset. The process is repeated on each subset for 10 times. In the end, the average precision and recall is 92.5% and 85.43% respectively.

As shown in Table 8, we also compare the average precision and recall with other two classifiers, i.e., One-Class Support Vector Machine learning (OC-SVM) [32] and Naive Bayes [30]. The results show that the standard SVM performs the best. We tried OC-SVM with only positive samples (elements containing sensitive attributes) to train the classifier. OC-SVM generated more false negatives than the standard SVM due to the lack of negative samples. Naive Bayes, a traditional probabilistic learning algorithm, also produced very imprecise results. This happens especially when it deals with elements that contain low-frequency privacy-related texts.

Classifier	Avg.Precision	Avg.Recall
SVM	92.50%	85.43%
OC-SVM	93.74%	68.48%
Naive Bayes	95.42%	26.70%

Table 8: Classifier Comparison

Manual Validation. We envision UIPicker to be used as an automated approach for labeling elements that contain UIP data. UIPicker achieves this by using some easily available UIP data (elements containing sensitive attributes or hand-annotated) and then using the classifier to automatically explore larger parts of UIP data. Measuring precision is hard in this setting as there is no entire pre-annotated elements (labeling sensitive or insensitive for all of them) for a set of apps that could compare with UIPicker's identification results.

As a best-effort solution, we randomly select 200 apps from top 10 categories (20 in each) ordered by %apps which UIP data appear most in Table 4 as the manual validation dataset. As such categories may contain much more UIP data than others, it provides the opportunity that our experts can walk through less apps (and activities) to validate more UIP elements. The selected apps are excluded from the classifier's training process to avoid overlap. Such way can greatly improve the effectiveness of the manual validation. Since the subset of apps is randomly picked, we believe that the evaluation results can provide a reasonable accuracy estimation on the entire dataset. For each element that UIPicker identifies as UIP data, we check their corresponding descriptions in XML layout files with some automated python scripts for efficiency (quickly locating the element in layout files and trying to understand it from descriptions). If this is still insufficient for us to identify whether it is a UIP data element, we confirm them by launching the app and find the element in the layout screen. The manual validation over 200 apps shows that UIPicker identifies 975 UIP data elements with 67 false positives and 107 false negatives.

False Positives: The false positive rate is 6.4% (67/1042 elements UIPicker identifies). In most cases, this is caused by the element's neighbors. That is, the element's neighbors contain privacy-related texts while the element itself is not privacy-related. Consider the following example, an *EditText* with only one description "message" while its previous element requires the user to input username with many sensitive textual phrases. As UIPicker takes neighbor elements' texts into consideration for better identification results, the privacy-related texts in its neighbor make UIPicker falsely identify the current element as UIP data. We consider such false alarm as acceptable because once such false alarm happens, their neighbor elements (the actual UIP data elements) are very possible to be identified by UIPicker as well.

False Negatives: We manually inspect each app in the evaluation dataset by traversing their UI screens as much as possible to see whether there exists UIP data elements that missed by UIPicker. In 200 apps, we find 107 elements not identified by UIPicker as privacy-related, and we conclude the reasons as follows: (1) Some very low-frequency texts representing UIP were not inferred from UIPicker by the privacy-related text analysis module. For example, "CVV" represents the credit card's security code, however we find this only happened in 4 Chinese apps. The low occurrence frequency of texts like "CVV" in our croups makes UIPicker fail to add them as features for the identification process. (2) In static analysis for behavior-based element filtering, due to FlowDroid's limitations, the call trace of some element was broken in inter-procedural analysis which makes UIPicker miss such elements in the final output.

Based on the total number of TPs, FPs and FNs (975, 67, 107), we compute the precision and recall of UIPicker as follows:

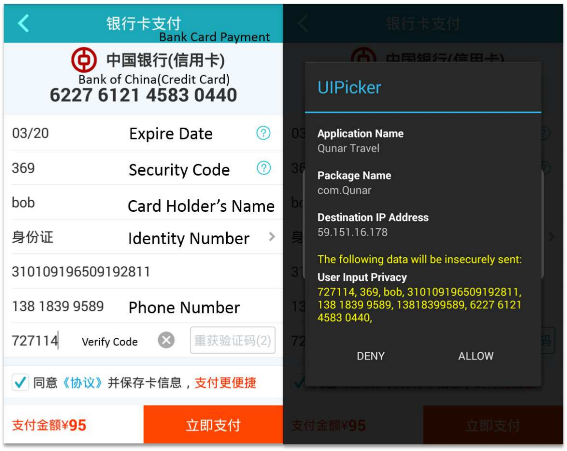
$$Precision = \frac{TP}{TP+FP} \qquad Recall = \frac{TP}{TP+FN}$$

Overall, UIPicker precisely identified most of UIP data, with 93.6% precision and 90.1% recall.

7.4 Runtime Enhancement Evaluation

System Overhead. We compare the performance overhead with TaintDroid using Antutu Benchmark [3]. We run Antutu 10 times in both systems under a Nexus Prime device, and the average scores are basically the same. This is reasonable because our mechanism only provides additional UIP data sources. We conclude that the security enhancement mechanism does not introduce noticeable additional performance overhead to TaintDroid.

Case Study. We find that some critical UIP data are under threats in Android apps. In Figure 7, a popular travel app "Qunar", which has 37 million downloads in China [12], sends users' credit card information with vulnerable SSL implementation during the payment process. The insecure transmission is reported to the user with a pop-up window when such data leave the device, thus the user can decide whether to proceed or use an alternative payment method to avoid the security risk.



中国银行(信用卡)
Bank of China(Credit Card)
6227 6121 4583 0440

03/20 Expire Date
369 Security Code
bob Card Holder’s Name
身份证 Identity Number
310109196509192811
138 1839 9589 Phone Number
727114 Verify Code 重获验证码(2)
☑ 同意《协议》并保存卡信息，支付更便捷
支付金额¥95

UIPicker
Application Name
Qunar Travel
Package Name
com.Qunar
Destination IP Address
59.151.16.178
The following data will be insecurely sent:
User Input Privacy
727114, 369, bob, 310109196509192811, 138 1839 9589, 13818399589, 6227 6121 4583 0440,

Figure 7: Insecure Transmission of UIP data. We use faked sensitive data in the experiment.

8 Discussion

In this section, we discuss the general applicability of UIPicker, as well as limitations and future work.

UIPicker is able to efficiently handle UIP data which previous work does not concentrate on, nor be able to cover. Compared with existing approaches that focus on System-Centric Privacy data, UIPicker rethinks privacy from a new perspective: sensitive data generated from user inputs, which is largely neglected for a long period. UIPicker provides an opportunity for users to make informed decisions in a timely manner when sensitive data leave the device insecurely, instead of letting users assume the app can be trusted.

UIPicker uses not only texts in UI screens but also texts in layout descriptions for UIP data identification. This framework is generic to all kinds of apps without locality limitation. The way UIPicker correlates UIP data from layout descriptions could also be leveraged by existing work [37, 28] that attempts to map the permission usage with app descriptions.

UIPicker has the following limitations. (1) UIPicker does not consider dynamically generated UI elements, although we have not found any UIP data element being generated at runtime in our experiments. Dynamic UI elements could be analyzed through more sophisticated static/dynamic analysis with the app’s program code, which is our future work. (2) Currently, UIPicker can not handle sensitive user inputs in *Webview* because they are not included in app layout resources. In the future, we plan to download such webpages by extracting their URLs from the app, then analyze their text contents as well.

9 RELATED WORK

Privacy source identification. Existing work [16, 29] focuses on mapping Android system permissions with API calls. PScout [16] proposes a version-independent analysis tool for complete permission-to-API mapping through static analysis. SUSI [29] uses a machine learning approach to classify and categorize more Android sources and sinks which are missed by previous info-flow taint tracking systems. The most similar work with UIPicker is SUPOR [25], which also aims to automatically identify sensitive user inputs using UI rendering, geometrical layout analysis and NLP techniques. SUPER mainly focuses on specific type of UI elements (EditText) while UIPicker is not limited to this.

Text analysis in Android app. Several studies utilize UI text analysis for different security proposes. AsDroid [24] detects stealthy behaviors in Android app by UI textual semantics and program behavior contradiction. However, it only uses a few keywords to cover sensitive operations such as “send sms”, “call phone”. CHABADA [21] checks application behaviors against application descriptions. It groups apps that are similar with each other according to their text descriptions. The machine learning classifier OC-SVM is used in CHABADA to identify apps whose used APIs differ from the common use of the APIs within the same group. Whyper [37] uses natural language processing (NLP) techniques to identify sentences that describe the need for a given permission in the app description. It uses Stanford Parser to extract short phrases and dependency relation characters from app descriptions and API documents related to permissions. AutoCog [28] improves Whyper’s precision and coverage through a learning-based algorithm to relate descriptions with permissions. UIPicker could potentially leverage their techniques to generate more complete privacy-related texts for UIP data identification.

Static analysis. There are lots of work [24, 26, 15, 20, 34] on using static analysis to detect privacy leakage, malware or vulnerabilities in Android apps. AsDroid takes control flow graphs and call graphs to search intent from API call sites to top level functions (Activities). UIPicker’s behavior-based result filtering is similar to AsDroid while they have different goals. SMV-HUNTER [34] uses static analysis to detect possible MITM vulnerabilities in large scale. The static analysis extracts input information from layout files and identifies vulnerable entry points from the application program code, which can be used to guide dynamic testing for triggering the vulnerable code.

10 CONCLUSION

In this paper, we propose UIPicker, a novel framework for identifying UIP data in large scale based on a novel combination of natural language processing, machine learning and program analysis techniques. UIPicker takes layout resources and program code to train a precise model for UIP data identification, which overcomes existing challenges with both good precision and coverage. With the sensitive elements identified by UIPicker, we also propose a runtime security enhancement mechanism to monitoring their sensitive inputs and provide warnings when such data insecurely leave the device. Our evaluation shows that UIPicker achieves 93.6% precision and 90.1% recall with manual validation on 200 popular apps. Our measurement in 17,425 top free apps shows that UIP data are largely distributed in market apps and our run-time monitoring mechanism based on UIPicker can effectively help user to protect such data.

Acknowledgements

We thank the anonymous reviewers and our shepherd Franziska Roesner for their insightful comments that helped improve the quality of the paper. We also thank Cheetah Mobile Inc. (NYSE:CMCM), Antiy labs, Li Tan and Yifei Wu for their assistance in our experiments. This work is funded in part by the National Program on Key Basic Research (NO. 2015CB358800), the National Natural Science Foundation of China (61300027, 61103078, 61170094), and the Science and Technology Commission of Shanghai Municipality (13511504402 and 13JC1400800). The TAMU author is supported in part by the National Science Foundation (NSF) under Grant 0954096 and the Air Force Office of Scientific Research (AFOSR) under FA-9550-13-1-0077. Also the IU author is supported in part by the NSF (1117106, 1223477 and 1223495). Any opinions, findings, and conclusions expressed in this material do not necessarily reflect the views of the funding agencies.

References

[1] Amazon online store. https://goo.gl/jYdVPr.

[2] Android-apktool. https://goo.gl/UgmPXp.

[3] Antutu benchmark. https://goo.gl/78W9xL.

[4] Av-comparatives : Mobile security review - september 2014. http://goo.gl/JfmcYh.

[5] Bank app users warned over android security. http://goo.gl/PWcqUy.

[6] Cm security : A peek into 2014's mobile security. http://goo.gl/i58ihW.

[7] Google bouncer. http://goo.gl/ET4JDW.

[8] Monkeyrunner. http://goo.gl/AQsIQu.

[9] Natural language toolkit. http://goo.gl/qzWulA.

[10] Phishing attack replaces android banking apps with malware. http://goo.gl/cJqqyX.

[11] Python implementations of various stemming algorithms. https://goo.gl/kdxkqv.

[12] Qunaer 7.3.8. http://goo.gl/1vB2k7.

[13] scikit-learn. http://goo.gl/mBzGUZ.

[14] Wordnet, a lexical database for english. http://goo.gl/KwzO0r.

[15] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014), ACM, p. 29.

[16] AU, K. W. Y., ZHOU, Y. F., HUANG, Z., AND LIE, D. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 217–228.

[17] BUGIEL, S., DAVI, L., DMITRIENKO, A., FISCHER, T., SADEGHI, A.-R., AND SHASTRY, B. Towards taming privilege-escalation attacks on android. In *NDSS* (2012).

[18] CHEN, Q. A., QIAN, Z., AND MAO, Z. M. Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In *Proc. 23rd USENIX Security Symposium (SEC14), USENIX Association* (2014).

[19] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones. vol. 57, ACM, pp. 99–106.

[20] FAHL, S., HARBACH, M., MUDERS, T., BAUMGÄRTNER, L., FREISLEBEN, B., AND SMITH, M. Why eve and mallory love android: an analysis of android ssl (in)security. In *Proceedings of the 2012 ACM SIGSAC Conference on Computer & Communications Security(CCS)* (2012).

[21] GORLA, A., TAVECCHIA, I., GROSS, F., AND ZELLER, A. Checking app behavior against app descriptions. In *ICSE* (2014), pp. 1025–1035.

[22] HEUSER, S., NADKARNI, A., ENCK, W., AND SADEGHI, A.-R. Asm: A programmable interface for extending android security. In *Proc. 23rd USENIX Security Symposium (SEC14)* (2014).

[23] HORNYACK, P., HAN, S., JUNG, J., SCHECHTER, S., AND WETHERALL, D. These arent the droids youre looking for. *Retrofitting Android to Protect Data from Imperious Applications. In: CCS* (2011).

[24] HUANG, J., ZHANG, X., TAN, L., WANG, P., AND LIANG, B. Asdroid: detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *ICSE* (2014), pp. 1036–1046.

[25] JIANJUN HUANG, PURDUE UNIVERSITY; ZHICHUN LI, X. X., AND WU, Z. Supor: Precise and scalable sensitive user input detection for android apps. In *Proc. of 24rd USENIX Security Symposium* (2015).

[26] LU, L., LI, Z., WU, Z., LEE, W., AND JIANG, G. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 229–240.

[27] NAUMAN, M., KHAN, S., AND ZHANG, X. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security* (2010), ACM, pp. 328–332.

[28] QU, Z., RASTOGI, V., ZHANG, X., CHEN, Y., ZHU, T., AND CHEN, Z. Autocog: Measuring the description-to-permission fidelity in android applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 1354–1365.

[29] RASTHOFER, S., ARZT, S., AND BODDEN, E. A machine-learning approach for classifying and categorizing android sources and sinks. In *2014 Network and Distributed System Security Symposium (NDSS)* (2014).

[30] RISH, I. An empirical study of the naive bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence* (2001), vol. 3, IBM New York, pp. 41–46.

[31] SALTON, G., WONG, A., AND YANG, C.-S. A vector space model for automatic indexing. *Communications of the ACM 18*, 11 (1975), 613–620.

[32] SCHÖLKOPF, B., PLATT, J. C., SHAWE-TAYLOR, J., SMOLA, A. J., AND WILLIAMSON, R. C. Estimating the support of a high-dimensional distribution. *Neural computation 13*, 7 (2001), 1443–1471.

[33] SMALLEY, S., AND CRAIG, R. Security enhanced (se) android: Bringing flexible mac to android. In *The 20th Annual Network and Distributed System Security (NDSS)* (2013).

[34] SOUNTHIRARAJ, D., SAHS, J., GREENWOOD, G., LIN, Z., AND KHAN, L. Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps. In *Proceedings of the 19th Network and Distributed System Security Symposium. San Diego, California, USA* (2014).

[35] SPYNS, P. Natural language processing. *Methods of information in medicine 35*, 4 (1996), 285–301.

[36] WEI, F., ROY, S., OU, X., ET AL. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 1329–1341.

[37] XU, R., SADI, H., AND ANDERSON, R. Whyper: Towards automating risk assessment of mobile applications. In *USENIX Security Symposium* (2013), pp. 539–552.

[38] XU, R., SAÏDI, H., AND ANDERSON, R. Aurasium: Practical policy enforcement for android applications. In *USENIX Security Symposium* (2012), pp. 539–552.

[39] YANG, Y., AND PEDERSEN, J. O. A comparative study on feature selection in text categorization. In *ICML* (1997), vol. 97, pp. 412–420.

[40] YANG, Z., YANG, M., ZHANG, Y., GU, G., NING, P., AND WANG, X. S. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 1043–1054.

[41] ZHANG, Y., YANG, M., XU, B., YANG, Z., GU, G., NING, P., WANG, X. S., AND ZANG, B. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 611–622.

[42] ZHOU, Y., AND JIANG, X. Detecting passive content leaks and pollution in android applications. In *The 20th Annual Network and Distributed System Security (NDSS)* (2013).

[43] ZHOU, Y., SINGH, K., AND JIANG, X. Owner-centric protection of unstructured data on smartphones. In *Trust and Trustworthy Computing* (2014), Springer, pp. 55–73.

Cloudy with a Chance of Breach: Forecasting Cyber Security Incidents

Yang Liu[1], *Armin Sarabi*[1], *Jing Zhang*[1], *Parinaz Naghizadeh*[1]
Manish Karir[2], *Michael Bailey*[3], *Mingyan Liu*[1,2]
[1] *EECS Department, University of Michigan, Ann Arbor*
[2] *QuadMetrics, Inc.*
[3] *ECE Department, University of Illinois, Urbana-Champaign*

Abstract

In this study we characterize the extent to which cyber security incidents, such as those referenced by Verizon in its annual Data Breach Investigations Reports (DBIR), can be predicted based on externally observable properties of an organization's network. We seek to proactively forecast an organization's breaches and to do so without cooperation of the organization itself. To accomplish this goal, we collect 258 externally measurable features about an organization's network from two main categories: mismanagement symptoms, such as misconfigured DNS or BGP within a network, and malicious activity time series, which include spam, phishing, and scanning activity sourced from these organizations. Using these features we train and test a Random Forest (RF) classifier against more than 1,000 incident reports taken from the VERIS community database, Hackmageddon, and the Web Hacking Incidents Database that cover events from mid-2013 to the end of 2014. The resulting classifier is able to achieve a 90% True Positive (TP) rate, a 10% False Positive (FP) rate, and an overall 90% accuracy.

1 Introduction

Recent data breaches, such as those at Target [35], JP Morgan [25], and Home Depot [49] highlight the increasing social and economic impact of such cyber incidents. For example, the JP Morgan Chase attack was believed to be one of the largest in history, affecting nearly 76 million households [25]. Often, by the time a breach is detected, it is already too late and the damage has already occurred. As a result, such events call into the question whether these breaches could have been predicted and the damage avoided. In this study we seek to understand the extent to which one can forecast if an organization may suffer a cyber security incident in the near future.

Machine learning has been used extensively in the cyber security domain, most prominently for *detection* of various malicious activities or entities, e.g., spam [44, 45] and phishing[39]. It has been used far less for the purpose of *prediction*, with the notable exception of [51], where textual data is used to train classifiers to predict whether a currently benign webpage may turn malicious in the near future. The difference between detection and prediction is analogous to the difference between diagnosing a patient who may already be ill (e.g., by using biopsy) vs. projecting whether a presently healthy person may become ill based on a variety of relevant factors. The former typically relies on identifying known characteristics of the object to be detected, while the latter on factors believed to correlate with the prediction objective.

To explore the effectiveness of forecasting security incidences we begin by collecting *externally* observed data on Internet organizations; we do not require information on the internal workings of a network or its hosts. To do so, we tap into a diverse set of data that captures different aspects of a network's security posture, ranging from the *explicit* or *behavioral*, such as externally observed malicious activities originating from a network (e.g., spam and phishing) to the *latent* or *relational*, such as mismanagement and misconfigurations in a network that deviate from known best practices. From this data we extract 258 features and feed them to a Random Forest (RF) classifier. We train and test the classifier on these features and more than 1,000 incident reports taken from the VERIS community database [55], Hackmageddon [42], and the Web Hacking Incidents Database [31] that cover events from mid-2013 to 2014. The resulting classifier can be configured over a wide range of operating points including one with 90% True Positive (TP) rate, 10% False Positive (FP) rate and an overall accuracy of 90%.

We posit that such cyber incident forecasting offers a completely different set of characteristics as compared to detection techniques, which in turn enables entirely new classes of applications that are not feasible with detection techniques alone. First and foremost, prediction allows *proactive* policies and measures to be adopted rather than *reactive* measures following the detection of an incident. Effective proactive actions can substantially reduce the potential cost incurred by an incident; in this sense prediction is complementary to detection. Cyber incident prediction also enables the development of effective risk management schemes such as cyber insurance, which introduces monetary incentives for the adoption of better cyber security policies and technologies. In the wake of recent breaches, the market for such policies has soared, with current written annual premiums estimated to be between $500M and $1B [47].

The remainder of the paper is organized as follows. Section 2 introduces the datasets used in this study and details the rationale for their use as well as our processing methodology. We then define the features we use in constructing the classifier and show why they are relevant in predicting security incidents in Section 3. We present the main prediction results as well as their implications in Section 4. In Section 5 we discuss a number of observations and illustrate several major data breaches in 2014 in the context of this prediction methodology. Related work is detailed in Section 6, and Section 7 concludes the paper.

2 Data Collection and Processing

Our study draws from a variety of data sources that collectively characterize the security posture of organizations, as well as security incident reports used to determine their security outcomes. These sources are summarized in Table 1 and detailed below; a subset of these has been made available at [7].

2.1 Security Posture Data

An organization's network security posture may be measured in various ways. Here, we utilize two families of measurement data. The first is measurements on a network's misconfigurations or deviations from standards and other operational recommendations; the second is measurements on malicious activities seen to originate from that network. These two types of measurements are related. In particular, in [58] Zhang et al. quantitatively established varying degrees of correlation between eight different mismanagement symptoms and the amount of malicious activities from an organization. The combination of both of these datasets represents a fairly comprehensive view of an organization's externally discernible security posture.

2.1.1 Mismanagement Symptoms

We use the following five mismanagement symptoms in our study, a subset of those studied in [58].

Open Recursive Resolvers: Misconfigured open DNS resolvers can be easily used to facilitate massive amplification attacks that target others. In order to help the network operations community address this wide spread threat, the Open Resolver Project [14] actively sends a DNS query to every public IPv4 address in port 53 to identify misconfigured DNS resolvers. In this study, we use a data snapshot collected on June 2, 2013. In total, 27.1 million open recursive resolvers were identified.

DNS Source Port Randomization: In order to minimize the threat of DNS cache poisoning attacks [13], current best practice (RFC 5452 [34]) recommends that DNS servers implement both source port randomization and a randomized query ID. Many servers however have not been patched to implement source port randomization. In [58], over 200,000 misconfigured DNS resolvers were detected based on the analysis over a set of DNS queries seen by VeriSign's .com and .net TLD name server on February 26, 2013. This is the data used in this study.

BGP Misconfiguration: BGP configuration errors or reconfiguration events can cause unnecessary routing protocol updates with short-lived announcements in the global routing table [40]. Zhang et. al detected 42.4 million short-lived routes with BGP updates from 12 BGP listeners in the Route Views project [32] during the first two weeks of June 2013 [58]; this data is used in our study.

Untrusted HTTPS Certificates: Secure websites utilize X.509 certificates as part of the TLS handshake in order to prove their identity to clients. Properly configured certificates should be signed by a browser-trusted certificate authority. It is possible to detect misconfigured websites by validating the certificate presented during the TLS handshake [33]. An Internet scan performed on March 22, 2013 found that only 10.3 million out of a total of 21.4 million sites presented browser-trusted certificates [58]. We use this dataset in our study.

Open SMTP Mail Relays: Email servers should perform filtering on the message source or destination to only allow users in their own domain to send email messages. This is documented in current best practice (RFC 2505 [38]), and misconfigured servers can be used in large scale spam campaigns. Though small in number, these represent a severe misconfiguration in an organizations' infrastructure. In this study, we use data collected on July 23, 2013, which detected 22,284 open mail relays [58].

None of the datasets mentioned above is necessarily directly related to a vulnerability. The presence of misconfigurations in an organization's networks and infras-

Category	Collection period	Datasets
Mismanagement symptoms	February 2013 - July 2013	Open Recursive Resolvers, DNS Source Port Randomization, BGP misconfiguration, Untrusted HTTPS Certificates, Open SMTP Mail Relays [58]
Malicious activities	May 2013 - December 2014	CBL[4] , SBL[22], SpamCop[19], WPBL[24], UCEPROTECT[23], SURBL[20], PhishTank[16], hpHosts[11], Darknet scanners list, Dshield[5], OpenBL[15]
Incident reports	August 2013 - December 2014	VERIS Community Database [55], Hackmageddon [42], Web Hacking Incidents [31]

Table 1: Summary of datasets used in this study. Mismanagement and malicious activity data are used to extract features, while incident reports are used to generate labels for the training and testing of a classifier.

tructure is, however, an indicator of the lack of appropriate policies and technological solutions to detect such failures. The latter increases the potential for a successful data breach.

Also, note that all of the above datasets were collected during roughly the first half of 2013. As we shall be using the mismanagement symptoms as features in constructing a classifier/predictor, it is important that these features reflect the condition of a network *prior* to the incidents. Consequently, our incident datasets (detailed in Section 2.2) cover incidents that occurred between August 2013 and December 2014. Note also that we use only a single snapshot of each of the symptoms; this is because such symptomatic data is relatively slow-changing over time, as systems are generally not reconfigured on a daily or even weekly basis.

2.1.2 Malicious Activity Data

Another indicator of the lack of policy or technical measures to improve security at an organization is the level of malicious activities observed to originate from its network assets and infrastructure. Such activity is often observed by well-established monitoring systems such as spam traps, darknet monitors, or DNS monitors. These observations are then distilled into blacklists. We use a set of reputation blacklists to measure the level of malicious activities in a network. This set further breaks down into three types: (1) those capturing spam activities, including CBL[4] , SBL[22], SpamCop[19], WPBL[24], and UCEPROTECT[23], (2) those capturing phishing and malware activities, including SURBL[20], PhishTank[16], and hpHosts[11], and (3) those capturing scanning activities, including the Darknet scanners list, Dshield[5], and OpenBL[15]. We use reputation blacklists that have been collected over a period of more than a year, starting in May 11, 2013 and ending in December 31, 2014. Each blacklist is refreshed on a daily basis and consists of a set of IP addresses seen to be engaged in some malicious activity. This longitudinal dataset allows us to characterize not only the presence of malicious activities from an organization, but also its dynamic behavior over time.

2.2 Security Incident Data

In addition to the security posture data described in the previous section, we require data on reported cyber-security incidents to serve as ground-truth in our study; such data is needed for the purpose of training the classifier, as well as for assessing its accuracy in predicting incidents (testing). In general, we believe such incidents are vastly under reported. In order to obtain a good coverage, we employ three collections of publicly available incident datasets. These are described below.

VERIS Community Database (VCDB) [55]: This dataset represents a broad ranging public effort to gather cyber security incident reports in a common format [55]. The collection is maintained by the Verizon RISK Team, and is used by Verizon in its highly publicized annual Data Breach Investigations Reports (DBIR) [56]. The current repository contains more than *5,000* incident reports, that cover a variety of different types of events such as server breach, website defacements, and physically stolen assets. Table 7 (in the Appendix) provides some example reports from this repository; a majority (64.99%) is from the US.

Of the full set, roughly 700 unique incidents were relevant to our study: we include only incidents that occurred after mid-2013 so that they are aligned with the security posture data, and those directly reflecting cyber-security issues. We therefore exclude those due to physical attacks, robbery, deliberate mis-operation by internal actors (e.g. disgruntled employees) and the like, as well as unnamed or unverified attack targets. We show several such examples in Table 2. Also note that even though the same IPs may appear in both the malicious and incident data, the independence of the features from ground-truth data is maintained because malicious activities only reveal botnet presence, which is *not* considered an incident type by or reported in any of our incident datasets.

Incident report	Reason to exclude
Student of a college changed score	Unknown target
Road construction sign hacked	Physical tampering
Praxair Healthcare Inc. asset stolen	Physical theft
Lucile Packard Child. Hosp.l asset stolen	Physical theft
Medicare Privilege Misuse	Deliberate internal misuse

Table 2: Examples of excluded VCDB incidents.

Hackmageddon [42]: This is an independently maintained cyber incident blog that aggregates and documents various public reports of cyber security incidents on a monthly basis. From the overall set we extract 300 incidents, in which the reported dates are aligned with our security posture data, between October 2013 and February 2014, and for which we are able to clearly identify the affected organizations.

The Web Hacking Incidents Database (WHID) [31]: This is an actively maintained cyber security incident repository; its goal is to raise awareness of cyber security issues and to provide information for statistical analysis. From the overall dataset we identify and extract roughly 150 incidents, for which the reported dates are aligned with our security posture data, between January 2014 and November 2014.

A breakdown of the incidents by type from each of these datasets is given in Table 5. Note that Hackmageddon and WHID have similar categories while VCDB has much broader categories.

Incident type	SQLi	Hijacking	Defacement	DDoS
Hackmageddon	38	9	97	59
WHID	12	5	16	45
Incident type	Crimeware	Cyber Esp.	Web app.	Else
VCDB	59	16	368	213

Table 3: Reported cyber incidents by category. Only the major categories in each set are shown. The "Else" category by VCDB represents incidents lacking sufficient detail for better classification.

2.3 Data Pre-processing

Though our diverse datasets give us substantial visibility into the state of security at an organizational level, the diversity also presents substantial challenges in aligning the data in both time and space. All of the security posture datasets – mismanagement and malicious activities – record information at the host IP-address level; e.g., they reveal whether a particular IP address is blacklisted on a given day, or whether a host at a specific IP address is misconfigured. On the other hand, a cyber incident report is typically associated with a company or organization, not with a specific IP address within that domain.

Conceptually, it is more natural to predict incidents for an organization for the following reasons. Firstly, our interest is in predicting incidents broadly defined as a way to assess organizational cyber risk. Secondly, while some IP addresses are statically associated with a machine, e.g., a web server, others are dynamically assigned due to mobility, e.g., through WiFi. In the latter case predicting for specific IP addresses no longer makes sense.

This mismatch in resolution means that we will have to (1) map an organization reported in an incident to a set of IP addresses and (2) aggregate mismanagement and maliciousness information over this set of addresses. To address the first step we will first retrieve a *sample IP address* in the network of the compromised organization, which is then used to identify an *aggregation unit* – a set of IP addresses – that allows us to recover the network asset involved in the incident. Sample IP addresses are obtained by manually processing each incident report, and the aggregation units are identified by using registration information from Regional Internet Registries (RIR) databases. These databases are collected separately from ARIN [3], LACNIC [12], APNIC [2], AFRINIC [1] and RIPE [18], who keep records of IP address blocks/prefixes that are allocated to an organization. ARIN, APNIC, AFRINIC and RIPE databases keep track of the IP addresses that have been allocated, along with the organizations they have been allocated to, labeled with a maintainer ID. LACNIC provides a less detailed database, only keeping track of allocated blocks and not the owners. In this case, we take the last allocation that contains our sample IP address – note a single IP address might be reallocated several times, as part of different IP blocks – i.e., the smallest block, as its owner.

2.3.1 Mapping Process

In the following paragraphs we explain in detail the manual process of (1): (1a) extracting sample IP addresses through a number of examples, and (1b) identifying the aggregation unit using the sample IP address. The general outline of the process for (1a) is that we first read the report concerning each incident, and extract the website of the company involved. If the website is the intrusion point in the breach, or indicative of the compromised network, then we take the address of this website to be our sample IP address. The website is determined to be indicative of the compromised network when the owner ID for the sample IP address matches the reported name of the victim network. Occasionally the victim network can be identified separately regardless of the website address, but in most cases this is found to be an effective way of quickly obtaining the owner ID.

Our first example [21] is a website defacement targeting the official website of the City of Mansfield, Ohio. Since the point of intrusion is clearly the website, we take its address as our sample IP address for this incident. Note that in this case the website might be managed by a 3rd party hosting company, a possibility discussed further when we explain the process to address (1b). The second example [6] is on Evernote resetting all user passwords following an attack on its online system. For this incident we identify said domain (evernote.com), and trace it to an IP block in ARIN's database registered to Evernote Corporation. Since this network is maintained by Evernote itself, we take evernote.com to be our sample IP address. Our final example [10] involves the defacement of Google Kenya and Google Burundi websites. As the report suggests, the hackers altered the DNS records of the domains by hacking into the Kenya and Burundi NICs. Since the attack was not through directly compromising the defaced websites, we excluded this incident – the victim in this incident is neither Google Kenya nor Google Burundi, but the networks owned by the NICs.

The above examples provide insight into the manual process of mapping incident to a network address. For a large portion of the reports the incident descriptor is unique and should therefore be treated as such; this is the main reason that such a mapping is primarily done manually. For a significant portion ($\sim$ 95%) of the reports we are able to identify the compromised network with a high level of confidence – in such cases either the report explicitly cites the website as the intrusion point (first example), or the network identified by the website is registered under the victim organization (second example). When neither of these conditions is satisfied, this incident is excluded unless we can identify the victim network through alternative means; such cases are few. Overall our process is a conservative one: we only include an incident when there is zero or minimal ambiguity. Finally, we also remove duplicate owner IDs in order to avoid a bias against commonly used hosting companies (e.g. Amazon, GoDaddy) in our training and testing process.

We now explain the process used in (1b) to map an obtained sample IP address (as well as the identified owner ID) to network(s) operated by a single entity. The general outline of this process is as follows: we take all the IP blocks that have the same owner ID listed in the RIR databases, excluding sub-blocks that have been reallocated to other organizations, as our aggregation unit. Continuing with the same set of examples, in the case of Evernote (second example) we reverse search ARIN's database and extract all IP blocks registered to Evernote Corporation, giving us a total of 520 IP addresses. For the case of the City of Mansfield website, using records kept by ARIN we see that its web address belongs to Linode, a cloud hosting company. Obviously Linode is also hosting other entities on its network without reported incidents. Nonetheless, in this case we take the network owned by Linode as our aggregation unit, since we cannot further differentiate the source IP address(es) more closely associated with the city. The inclusion of such cases is a tradeoff as excluding them would have left us with too few samples to perform a meaningful study. More on this is discussed in Section 2.4.

2.3.2 A global table of aggregation units

The above explains how we process the incident reports to identify network units that should be given a label of “1”, i.e., victim organizations. For training and testing purposes we also need to identify network units that should be given a label of “0”, i.e., non-victim organizations. To accomplish this, we built a global table using information gathered from the RIRs that provides us with a global aggregation rule, containing both victim and non-victim organizations. Our global table contains 4.4 million prefixes listed under 2.6 million owner IDs. Note that the number of prefixes in the RIR databases is considerably larger than the global BGP routing table size, which includes roughly 550,000 unique prefixes [41]. This is partly due to the fact that the prefixes in our table can overlap for those that have been reallocated multiple times. In other words, the RIR databases can be viewed as a tree indicating all the ownership allocations and reallocations over the IP address space. On the other hand, the BGP table tends to combine prefixes that are located within the same Autonomous System (AS), in order to reduce routing table sizes. Therefore, the RIR databases provide us with a finer-grained look into the IP address space. By taking all the IP addresses that have been allocated to an organization, and have not been further reallocated, we can break the IP address space into mutually exclusive sets, each owned and/or maintained by a single organization. Out of the 4.4 million prefixes, 300,000 of them are assigned by LACNIC and therefore have no owner ID. Combined with the 2.6 million owner IDs from the other registries, the IP address space is broken, by ownership (or LACNIC prefixes), into 2.9 million sets. Each set constitutes an aggregation unit that is given a label of “0”, except for those already identified and labeled as “1” by the previous process.

2.3.3 Aggregation Process

Once these aggregation units are identified, the second step (2) is relatively straightforward. For each mismanagement symptom we simply calculate the fraction of symptomatic IPs within such a unit. For malicious activities, we count the number of unique IP addresses listed on a given day (by a single blacklist, or by blacklists monitoring the same type of malicious activities) that belong to this unit; this results in one or more time series for each unit. This step is carried out in the same way for both victim and non-victim organizations.

2.4 A Few Caveats

As already alluded to, our data processing consists of a series of rules of thumb that we follow to make the data useable, some perhaps less clear-cut than others. Below we summarize the typical challenges we encounter in this process and their possible implications on the prediction performance.

As described in Section 2.3, the aggregation units are defined using ownership information from RIR databases. One issue with the use of ownership information is that big corporations tend to register their IP address blocks under multiple owner IDs, and in our processing these IDs are treated as separate organizations. In principle, as long as each of the aggregation units is non-trivial in size, each can have its own security posture assessed. Furthermore, in some cases it is more accurate to treat such IDs separately, since they might represent different sections of an organization under different management. The opposite issue also exists, where it may be impossible to distinguish between the network assets of multiple organizations; recall, e.g. our first example where multiple organizations are hosted on the same network. As mentioned before, we have chosen in such cases to use the owner ID as the aggregation unit. While this mapping process is clearly non-ideal, it is a best-effort attempt at the problem, and will instead provide the classifier with the average value of the features over all organizations hosted on the identified network.

The labels for our classifier are extracted from real incident reports, and we can safely assume that the amount of false positives in these reports, if any, is negligible. However data breach incidents are only reported when an external source detects the data breach (e.g. website defacements), or an organization is obligated to report the incident due to private customer information getting compromised. In general, organizations tend not to announce incidents publicly, and security incidents remain largely under-reported. This will affect our classifier in two ways: First, by failing to incorporate all incidents in our training set, we may fail to identify all of the factors that might affect an organization's likelihood of suffering a breach. Second, when choosing non-victim organizations, it is possible that we select some of them from unreported victims, which could further impact the accuracy of our classifier. We have tried to overcome this challenge by using three independently maintained incident datasets. Ultimately, however, this can only be addressed when timely incident reporting becomes the norm; more on this is discussed in Section 5.

Last but not least, all the raw security posture data (mismanagement symptoms and blacklists) could contain error, which we have no easy way of calibrating. However, two aspects of the present study help mitigate the potential impact of these noises. Firstly, we use many different datasets from independent sources; the diversity and the total volume generally have a dampening effect on the impact of the noise contained in any single source. Secondly and perhaps more importantly, our ultimate verification and evaluation of the prediction performance are not based on the security posture data, but on the incident reports (with their own issues as noted above). In this sense, as long as the prediction performance is satisfactory, the noise in the input data becomes less relevant.

3 Forecasting Methodology

The key to our prediction framework is the construction of a good classifier. We will primarily focus on the Random Forest (RF) method [37], which is an ensemble classifier and an enhancement to the classical random decision tree method. It uses randomly selected subsets of samples to construct different decision trees to form a forest, and is generally considered to work well with large and diverse feature sets. In particularly, it has been observed to work well in several Internet measurement studies, see e.g., [57]. As a reference, we will also provide performance comparison by using the Support Vector Machine (SVM) [27], one of the earliest and most common classifiers. To train a classifier, we need to identify a set of features from the measurement data. Below, we first detail the set of features used, and then present the training and testing procedures.

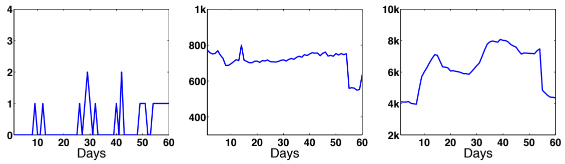
3.1 Feature Set

We shall use two types of features, a primary set and a secondary set. The primary set of features consists of the raw data, while the secondary set is derived or extracted from the raw data, i.e., in the form of various statistics. In all, 258 features are used, including 5 mismanagement features, 180 primary features, 72 secondary features, and a last feature on the organization size.

3.1.1 Primary Features (186)

Mismanagement symptoms (5). There are five symptoms; each is measured by the ratio between the number of misconfigured systems and the total number of systems in an organization. For instance, for the untrusted HTTPS certificates, this ratio is between the number of misconfigured certificates over the total number of certificates discovered in an organization. Similarly, for open SMTP mail relay this ratio is between the number of misconfigured mail servers and the total number of mail servers. The only exception is in the case of open recursive resolver: since we do not know the total number of open resolvers, this ratio is between the number of misconfigured open DNS resolvers and the total number of IPs in an organization. These ratios are denoted as $\mathbf{m}_i \in [0,1]^5$ for organization i.

Malicious activity time series (60 × 3). For each organization we collect three separate time series, one for each malicious activity type, namely spam, phishing, and scan. Accordingly, for organization i, its time series data are denoted by $\mathbf{r}_i^{SP}, \mathbf{r}_i^{PH}, \mathbf{r}_i^{SC}$. These time series data are directly fed in their entirety into the classifier. Several examples of $\mathbf{r}_i^{SP}$ are given in Fig. 1; these are collected over a two-month (60 days) period and show the total number of unique IPs blacklisted on each day over all spam blacklists in our dataset.



(a) Org. 1 (b) Org. 2 (c) Org. 3

Figure 1: Examples of malicious activity time series of three organizations; Y-axis is the number of unique IP addresses listed on all spam blacklists in each day over a 60-day period.

Size (1). This refers to the size of an organization in terms of the number of IP addresses identified within that organization's aggregation unit as outlined in the previous section. For organization i, this is denoted by s_i.

The relevance of these symptoms to an organization's security posture is examined more closely by comparing their distributions among the victim and the non-victim populations, as shown in Fig. 2. We see a clear difference between the two populations in their untrusted HTTPS and Openresolver distributions. This difference suggests that these symptoms are meaningful distinguishers, and thus hold predictive power. This is indeed verified later when these two symptoms emerge as the most indicative of the five. By contrast, the other three mismanagement symptoms appear much less powerful.

The relevance of the malicious activity time series will be examined more closely in the next section, within the context of their secondary features. Lastly, the organization size can to some extent capture the likelihood of an organization becoming a target of intentional attacks, and is therefore included in the feature set.

3.1.2 Secondary Features (72)

In determining what type of statistics to extract to serve as secondary features, we aim to capture distinct behavioral patterns in an organization's malicious activities, particularly concerning their dynamic changes. To illustrate, the three examples given in Fig. 1 show drastically different behavior: Org. 1 shows a network with consistently low level of observed malicious IPs (and possibly within the noise inherent in the blacklists), while Examples 2 and 3 show much higher levels of activity in general. These two, however, differ in how persistent they are at those high levels. Example 2 shows a network with high levels throughout this period, while Example 3 shows a network that fluctuates much more wildly. Intuitively, such dynamic behavior reflects to a large degree how responsive the network operators are to blacklisting, i.e., time to clean up, time to resurfacing of malicious activities, and so on.

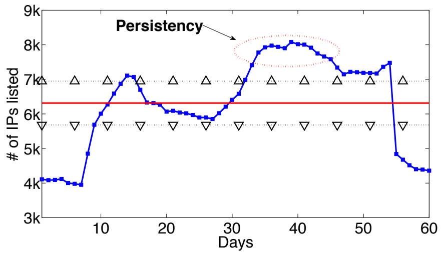


Figure 3: Extracting secondary features. The solid red line indicates time-average of the signal while the two dotted lines denote the boundary of different regions. The region above is "bad" with higher-than-average malicious activities, while the region below is "good" with lower-than-average activities. Persistency refers to the duration the time series persist in the same region.

These observed differences motivate us to collect statistics summarizing such behavioral patterns by measuring their persistence and change, e.g., how big is the change in the magnitude of malicious activities over time and how frequently does it change. To balance the expressiveness of the features and their complexity, we shall do so by first value-quantizing a time series into three regions relative to its time average: "good", "normal" and "bad". An illustration is given in Fig. 3 using one of the examples shown earlier (Org. 3). The solid line marks the average magnitude of the time series over the observation period; the dotted lines then outline the "normal" region, i.e., a range of magnitude values that are relatively close (either from above or below) to its time-average. The region above the top dotted line is accordingly referred to as the "bad" region, showing large number of malicious IPs, and the region below the bottom dotted line the "good" region, with a smaller number of malicious IPs, both relative to its average[1].

An additional motivation behind this quantization step is to capture certain onset and departure of "events", such as a wide-area infection, or scheduled patching and software update, etc. Viewed this way, the duration an orga-

[1] The choice on the size of the normal region may lead to differences in classifier performance, which is discussed in more detail in Section 5.3. In most of our experiments ±20% of the time average is used.

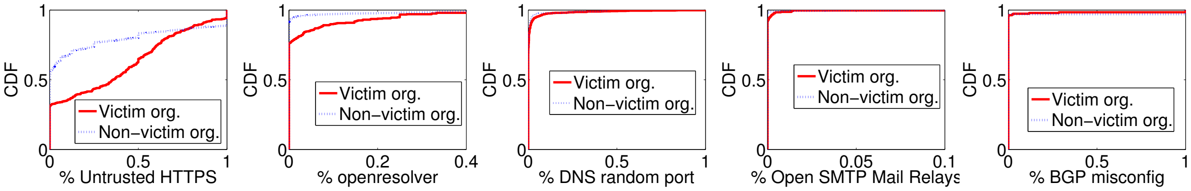


Figure 2: Comparison of mismanagement symptoms between the victim and non-victim populations. There is a clear separation under the first two, while the other three appear to be much weaker predictors.

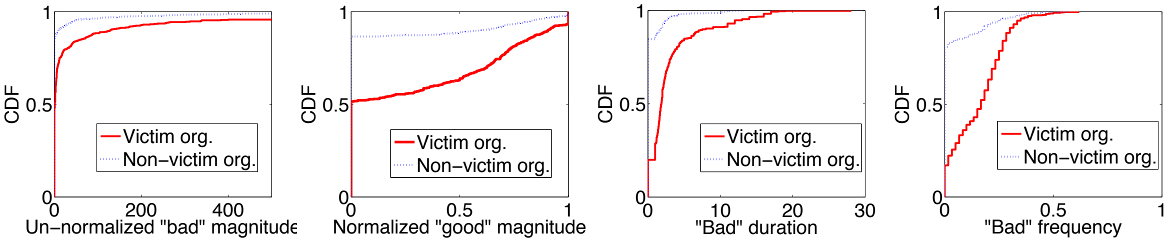


Figure 4: Profile of selected temporal features extracted from the scanning time series over the period Nov. 13-Dec.13.

nization spends in a "bad" region could be indicative of the delay in responding to an event, and similarly, how frequent it re-enters a "bad" region could be indicative of the effectiveness of the solutions taken in remaining clean.

Accordingly, for each region we then measure the average magnitude (both normalized by the total number of IPs in an organization and unnormalized), the average duration that the time series persists in that region upon each entry (in days), and the frequency at which the time series enters that region. This results in four summary statistics for each region, thus 12 values for each time series. Since each organization has three time series, one for each malicious activity type, we obtain a total of 36 derived features per organization i. These will be collectively denoted by the feature vector F_i. Note that the set of 36 values are collected from time series of a certain duration. Here we further distinguish between statistics extracted from a longer period of time vs. from a shorter, most recent period of time. In this study we use two such feature vectors, one referred to as *Recent-60* features that are collected over a period of 60 days (typically leading up to the time of incident) and the other *Recent-14* features collected over a period of 14 days (leading up to the time of incident).

To give a sense of why these features may be expected to hold predictive power, we similarly compare the distribution of these feature values among the victim and non-victim populations. Fig. 4 shows this comparison for four examples: un-normalized magnitude in a bad period, normalized magnitude in a good period, average duration during bad periods, and the frequency of entering a bad period. We see that in each case there is a clear difference between the two populations in how these feature values are distributed, e.g., victim organizations tend to have longer bad periods, indicative of slow response time, and also higher bad/good magnitudes, etc. As we discuss further in Section 4.4, these features have varying degrees of influence over the prediction outcome.

3.2 Training and Testing Procedure

We now describe the construction of the predictor using the set of features defined above. This consists of a training step and a testing step. The training step uses the following two sets of subjects.

A subset of incident or victim organizations. This will be referred to as Group(1) or the incident group. Depending on the experiments, this subset may be selected from one of the three incident datasets (if we train the classifier and conduct testing based solely on one incident dataset), or from the union of all three. This subset is selected based on the time stamps of the reported incidents, and its size is determined by a training-testing ratio, e.g., 70-30 split or 50-50 split of the given dataset. If we use a 50-50 split, it means that we select the first half (in terms of time of occurrence) of the incidents as Group(1); a 70-30 split means using the first 70% of incidents as Group(1). The remaining victim organizations are used in the testing step.

A randomly selected set of non-victim organizations (with size comparable to that of Group(1) in any given experiment). These are taken from the global table described in Section 2.3.2. This will be referred to as

Group(0), or the non-incident group. As mentioned earlier, since there are close to three million non-victim organizations compared to less than a thousand victim organizations, the random sub-sampling is necessary to avoid the common problem of imbalance in the machine learning literature[2]; this issue has also been discussed in [51]. This random selection of non-victim organizations is repeated numerous times, each time training a different classifier. The reported testing results are averages over all these versions.

For a victim organization i in Group(1), its complete feature set $\mathbf{x}_i$ includes the mismanagement symptoms $\mathbf{m}_i$, the three time series $\mathbf{r}_i^{SP}, \mathbf{r}_i^{PH}, \mathbf{r}_i^{SC}$ over the two months prior to the month in which the incident in i occurred[3], secondary features F_i collected over the same time period as the time series, namely Recent-60, and that collected over the two weeks prior to the month of the incident occurrence, namely Recent-14. Each such feature set is associated with the label (or ground-truth or group information in machine learning) $L_i = 1$ for incident. For a non-victim organization j in Group(0), its complete feature set $\mathbf{x}_j$ consists of exactly the same components listed above, with the only difference that the time series and the secondary features are for the two months prior to the month of the first incident in Group(1). It is also associated with the label $L_j = 0$ for non-incident.

The collections of $\{[\mathbf{x}_i, L_i]\}$ and $\{[\mathbf{x}_j, L_j]\}$ constitute the training data used to train the classifier. The testing step then uses the following two inputs: (1) The subset of victim organizations not included in Group(1); denote this group by Group(1^c). (2) A randomly selected set of non-victim organizations not used in training. Unlike in training where we try to keep a balance between the victim and non-victim sets, during testing we use a much larger set of non-victim organizations to better characterize the classifier performance.

For these two subjects their complete feature sets $\mathbf{x}_i$ are obtained in exactly the same way as for those used in training. For the non-victim organizations selected for testing, the features are collected over the two months prior to the incident month of the first incident in Group(1^c). For the victim organization used for testing we further consider two scenarios. In the *short-term forecast* scenario, we collect these features over the two months prior to the incident month for an organization in Group(1^c), while in the *long-term forecast* scenario, we collect these features over the two months prior to the incident month of the first incident in Group(1^c). In the short-term forecast scenario, since in each incident test case the incident occurred within a month of collecting the features, the TP rate is essentially for a forecasting window of one month. In the long-term forecast scenario, an incident may occur months after collecting the features (up to 12 months in the case of VCDB), thus the TP rate is for a forecasting window of up to a year. Note that the short-term forecast can be repeatedly done over time to produce prediction for the immediate future. The differences between these two forecast schemes are also illustrated in Fig. 5.

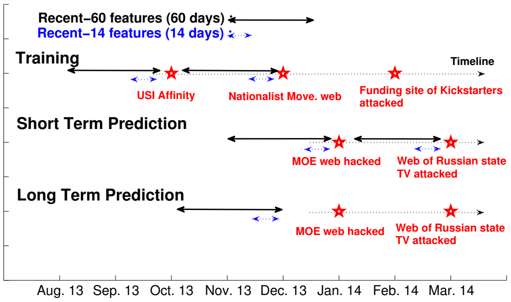


Figure 5: Feature extraction, short-term and long-term forecasting. In training, features are extracted from the most recent period leading up to an incident. In testing, the same is done when we perform short-term forecast. In long-term forecast, features are extracted from periods leading up to the time of the first incident used in testing.

These inputs are then fed into the classifier to produce a label (or prediction). The output of Random Forest is actually a risk probability; a threshold is then imposed to obtain a binary label. For instance, if we set the threshold at 0.5, then all output > 0.5 means a label of 1. By moving this threshold we obtain different prediction performances, which constitute a ROC curve.

4 Incident Prediction

In this section, we present our main prediction results and investigate their various implications.

4.1 Main Results

Using the methodology outlined in the previous section, we performed prediction using the three incident datasets separately, as well as collectively. When used collectively, we removed duplicate reports of the same incident whenever applicable. The separation between training and testing for each dataset is done chronologically, as shown in Table 4. For each dataset, these separations

[2] If we use all three million non-victims in training, the resulting classifier will simply label all of them as non-victims, and achieve performance very close to 100% overall. But clearly this classifier would be of little use, as it will also have 0 true positive probability.

[3] Most incident occurrences in our dataset are timestamped with month and year information.

result in an approximate 50-50 split of the victim set between the training and testing sample sizes. In addition, for each test we randomly sample non-victim test cases from the non-victim organization set.

	Hackmageddon	VCDB	WHID
Training	Oct 13 – Dec 13	Aug 13 – Dec 13	Jan 14 – Mar 14
Testing	Jan 14 – Feb 14	Jan 14 – Dec 14	Apr 14 – Nov 14

Table 4: Chronological separation between training and testing samples for each incident dataset; the split is roughly 50-50 among the victim population.

There is one point worth clarifying. When processing non-sequential data, the split of samples for the purpose of training and testing is often done randomly in the machine learning literature. In our context this would mean to choose a later incident for training and use an earlier incident for testing. Due to the sequential nature of our data, we intentionally and strictly split the data by time: earlier ones are for training and later ones for testing. Because of this, our testing results are indeed “prediction” results; for the same reason, we did not set aside a third, separate dataset for the purpose of “more testing” as is sometimes done in the literature, as this purpose is already served by the second, test dataset.

The prediction results are summarized in the set of ROC (receiver operating characteristic) curves shown in Fig. 6. Recall that the RF classifier outputs a probability of incident for each input sample. To test its accuracy, a threshold is adopted that maps this value into a binary prediction: 1 if it exceeds the threshold and 0 otherwise. This binary prediction is then compared against the ground-truth: a sample from an incident dataset has a true label of 1, while a sample from the non-victim organization set has a true label of 0. Since our non-victim set for training (to balance) is randomly selected from the total non-victim population, the above test is repeated 20 times for a given threshold value, each time for a different random non-victim set. The average TP and FP over these repeated tests form one point on the ROC curve.

We see the prediction performance varies slightly between the datasets, but remain very satisfactory, generally achieving combined (TP, FP) values of $(90\%, 10\%)$ or $(80\%, 5\%)$. In particular, when we combine the three datasets, we can achieve an accuracy level of $(88\%, 4\%)$. A summary of some of the most desirable operating points are given in Table 5.

The above prediction results substantially outperform what has been shown in the literature to date; e.g., the web maliciousness prediction study in [51] reported a combination of $(66\%, 17\%)$ for (TP, FP). It is also worth pointing out that TP and FP values are independent of the sizes of the respective populations of the victim and non-victim organizations (these are conditional probability estimates), whereas the overall accuracy does depend on the two population sizes as it is the unconditioned probability of making correct predictions. Since based on our dataset we have a minuscule victim population (accounting for $\ll 1\%$ of the overall population), the overall accuracy is simply $\sim$ (1-FP). Therefore, if the overall accuracy is of interest, the best classifier would be a naive one that simply labels all inputs as “0”. This would lead to 0% TP, 0% FP, and an overall accuracy of $> 99\%$. However, despite achieving maximum overall accuracy, such a classifier is clearly useless. This point is also emphasized in [51] for similar reasons. Additionally, in the context of forecasting, where the goal is to facilitate preventative measures at an organizational level, having a high TP is perhaps more relevant than having a low FP; this is in contrast to spam detection, where the cost of FP is much higher than a missed detection. Therefore, the three measures in Table 5 should be taken as a whole.

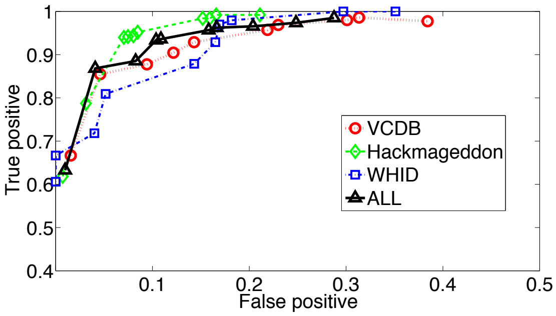


Figure 6: Prediction results. There are variations between the datasets, but an operating point – combined (TP, FP) values – of $(90\%, 10\%)$ or $(80\%, 5\%)$ is achievable. In particular, when we use all three datasets together, we can achieve an accuracy level of $(88\%, 4\%)$.

Accuracy	Hackmageddon	VCDB	WHID	All
True Positive (TP)	96%	88%	80%	90%
False Positive (FP)	10%	10%	5%	10%
False Negative (FN)	4%	12%	20%	10%
Overall Accuracy	90%	90%	95%	90%

Table 5: Best operating points of the classifier for the best combinations of (TP, FP) values.

4.2 Impact of Training:Testing Ratio

The results in Fig. 6 are obtained under a 50-50 split of the victim set into training and testing samples, based on the incident time. Furthermore, they are obtained using the short-term forecasting method described in Section 3.2. In general, one can improve the prediction performance by increasing the training sample size. There is no exception in our study, as shown in Fig. 7 where we compare results from a 70-30 training and testing sample split of the victim set to that from the 50-50 split, for the VCDB data. A best operating point is now around $(94\%, 10\%)$, indicating a clear improvement. Note that, a 70-30 split is not generally regarded high in the machine learning literatures, see e.g., in [57] a 90-10 split was used. We however believe a 50-50 split gives a more objective measure of the prediction performance.

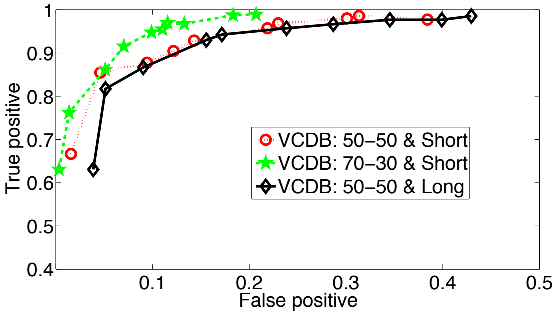


Figure 7: The impact of larger training set and size of forecasting window; all results are obtained using VCDB. The three curves: (1) using a 50-50 split of the victim set between training and testing under the short-term forecasting scenario (this curve is identical to the one in Fig 6); (2) using a 70-30 split of the victim set between training and testing under the short-term forecasting scenario; (3) using a 50-50 split of the victim set between training and testing under the long-term forecasting scenario.

4.3 Short-term vs. Long-term Forecast

Also shown in Fig. 7 are our long-term forecasting results under a 50-50 training and testing sample split of the victim set, again for VCDB. As seen, the prediction performance holds even when we move from a one-month to a 12-month forecasting window. The use of mismanagement symptoms and long-term malicious behaviors in the features contributes to this: they generally remain stable over time and have relatively high importance in the prediction, discussed in greater detail in the next section.

4.4 Relative Importance of the Features

In addition to the prediction output, the RF classifier also outputs a normalized relevance score for each feature used in training [17]; the higher the value, the more important the feature in the prediction. In this section, we examine these scores more closely. This study will further help us understand the extent to which different features determine the chance of an organization becoming breached in the near future. For brevity, the experiments presented in this section are based on a combination of all three datasets.

The importance of each category of features is summarized in Table 6. We make a number of interesting observations. First, note that the mismanagement features stand out as the most important category in prediction. Second, the Recent-60 secondary features are almost as important as the time series data, despite the fact that the former are derived from the latter. This is because the use of time series data has the effect of capturing synchronized behavior between malicious activities of different organizations, while the secondary features are aimed at capturing the dynamic behavior over time within an organization itself. That the latter adds value to the predictor is thus validated by the above importance comparison. Last but not least, the Recent-60 features appear much more important than Recent-14 features.

Feature category	Normalized importance
Mismanagement	0.3229
Time series data	0.2994
Recent-60 secondary features	0.2602
Organization size	0.0976
Recent-14 secondary features	0.02

Table 6: Feature importance by category. The mismanagement features are the most important category in prediction. Secondly, the Recent-60 secondary features are almost as important as the time series data; the former capture dynamic behavior over time within an organization whereas the latter capture synchronized behavior between malicious activities of different organizations.

A closer look into each category reveals that among the mismanagement features, untrusted HTTPS is by far the most important (0.1531), followed by Openresolver (0.0928), DNS random port (0.0469), Mail relay (0.0169), and BGP misconfig. (0.0132). The more significant role of untrusted HTTPS in prediction as compared to Openresolver is consistent with the bigger difference in distributions seen earlier in Fig. 2; that is, a victim organization tends to have a higher percentage of mis-configured HTTPS for their network assets. A possible explanation is that a majority of the incidents in our dataset are web-page breaches; these correlate with the untrusted HTTPS symptom, which reflect poorly managed web systems.

Similarly, a closer look at the secondary features (both Recent-60 and Recent-14) suggests that the dynamic features (duration and frequency together, totaling 0.1769) are far more important than static features (magnitude, totaling 0.0834). This suggests that dynamic changes over time, or in other words, organizations' response time in terms of cleaning up the origin of their malicious activities, is more indicative of security risks.

4.5 The Power of Dataset Diversity

A question that naturally arises is what if only a single feature category is used to train the classifier. For instance, given the prominent score of mismanagement features in prediction, would it be sufficient to only use these in prediction? The answer, as shown in Fig. 8, turns out to be negative. In this figure, we compare the prediction performance by using the following four categories of features separately to build the classifier: mismanagement, time series data, organization size, and the entire set of secondary features. While it is expected that using only one feature set leads to worse prediction performance, it is somewhat surprising that the secondary features are more powerful than mismanagement features or the time series when used separately. Recall that the secondary features were designed specifically to capture the organizational behavior, including their responsiveness and effectiveness in dealing with malicious acti-vities. One explanation of this result is that the human and process element of an organization is the most slow-changing compared to the change in the threat landscape, and thus holds the most predictive power.

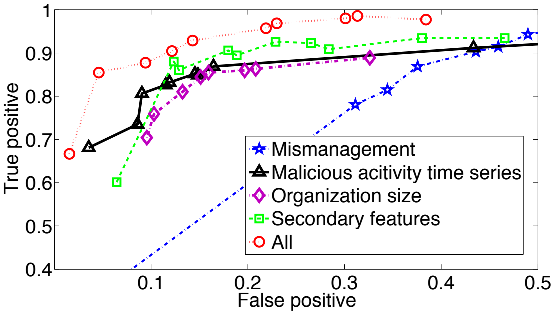


Figure 8: Independent prediction performance using only one set of features. The secondary features are shown to be the most powerful in prediction when used alone. Mismanagement features perform the worst, even though they have the highest importance factor. This is because the factors reflect conditional importance, given the presence of other features. This means that mismanagement features alone are poor predictors but they add valuable information to the other features.

Note that this is not inconsistent with the relative importance given in Table 6, as the latter is a measure of conditional importance of one feature given the presence of other features. In other words, the relative importance suggests how much we lose in performance if we *leave out* a feature, whereas Fig. 8 shows how well we do when using *only* that feature. What's seen here is that mismanagement features add very significant (orthogonal) information to the other features, but they are poor predictors in and by themselves. Perhaps most importantly, the results in Fig. 8 validate the idea of using a diverse set of measurement data that collectively form predictive descriptions of an organization's security risks.

4.6 Comparison with SVM

As a reference, we also trained classifiers using SVM; the prediction results are much poorer compared to using RF. For instance, using the VCDB data, the best operating point under SVM (with a 50-50 training-testing split of the victim population and short-term forecasting) is around (70%, 25%). This observation is consistent with existing literature, see e.g., [57].

5 Discussion

5.1 Top Data Breaches of 2014

In Fig. 9, we plot the distribution (CDF) of the predictor output values for the VCDB victim set and a randomly selected non-victim set used in testing. We use an example threshold of 0.85 for illustration. All points to the right of a threshold is labeled “1”, indicating positive prediction, and all to its left “0”. Three incident examples are also shown, falling into the categories of true-positive (ACME), false-positive (AXTEL), and false-negative (BJP Junagadh).

Also highlighted in Fig. 9 are the top five data breaches of 2014 [43], namely JP Morgan Chase, Sony pictures, Ebay, Home Depot, and Target. Using the suggested threshold value, our prediction method would have correctly labeled four of these incidents, and only narrowly missed the Target incident. It is worth noting that the Target incident was brought on by one of its contractors; however, the fact that Target did not have a more secure vendor policy in place is indicative of something else amiss (e.g., lack of consistent procedure between IT and procurement) that could also have manifested itself in the data and features we examined.

These examples highlight that, in addition to enabling proactive measures by an organization, there are potential business uses of the prediction method presented in this study. The first is in vendor or third party evaluation. Consider Online Tech, the hosting service used by JP Morgan Chase, as an example. As shown in Fig. 9, Online Tech posed very high security risks; this information could have been used in determining whether to use this vendor. Furthermore, information provided by our prediction method can help underwriters better customize terms of a cyber-insurance policy. The insurance

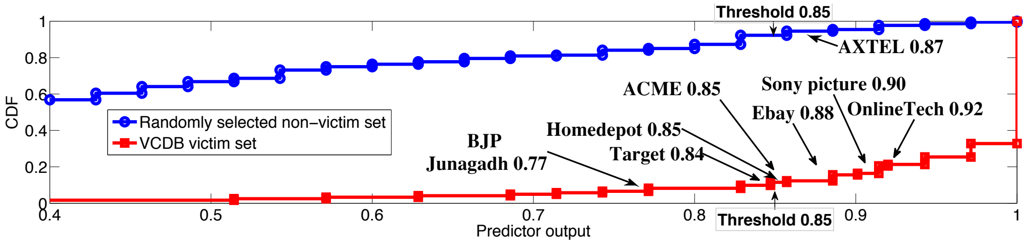


Figure 9: Distribution of predictor outputs with an example threshold value 0.85 (with 91% TP, 10% FP). On the curve with circles (non-victim) to the right of the threshold are FPs; on the curve with squares (victim) to the right of the threshold are TPs. Three types of incidents are shown, presenting true-positive (ACME), false-positive (AXTEL), and false-negative (BJP Junagadh). Also highlighted are top data breach events in 2014.

payout in the case of Target was reported to be around $60M; with this much at stake, it is highly desirable to be able to accurately predict the risk of an insured and adjust terms of the contracts.

5.2 Prediction by Incident Type

For a majority of the incident types, we do not have enough sample points to serve both training and testing purposes, except for the 368 reports of the type "web applications incident" in VCDB. This allows us to train a classifier to predict the probability of an organization being hit with a "web app incident". The corresponding results are similar in accuracy to those obtained earlier (e.g., at (92%, 11%)). This suggests that our methodology has the potential to make more precise predictions as we accumulate more incident data.

Similarly, the current forecasting methodology is not aimed at predicting highly targeted attacks motivated by geo-political reasons (e.g., the Sony Picture breach). Nor does it use explicit business sector information (e.g., a bank may be a bigger target than a public library system). In this sense, our current results represent more the likelihood of an organization falling victim provided it is being targeted. However, an ever increasing swath of the Internet is rapidly under cyber threats to the point that all major organizations should simply assume that they are someone's target. The use of explicit business sector information does allow us to make more fine-grained predictions. In a more recent study [48], we leverage a broad array of publicly available business details on victim organizations reported in VCDB, including business sector, employee count, region of operation and web statistics information from Alexa Web Information Service (AWIS), to generate risk profiles, the conditional probabilities of an organization suffering from specific types of incident, given that an incident occurs.

5.3 Robustness against adversarial data manipulation and other design choices

One design choice we made in the feature extraction process is the parameter δ which determines how a time series is quantized to obtain secondary features. Below we summarize the impact of of having different δ values. In the results shown so far, a value of $\delta = 0.2$ is used. In Fig. 10 we test the cases with $\delta = 0.1$ and $\delta = 0.3$. We see that this parameter choice has relatively minor effect: with $\delta = 0.3$ a desirable TP/FP combination is around $(91\%, 9\%)$, and for $\delta = 0.1$, we have $(86\%, 6\%)$. It appears that having a higher value of δ leads to slightly better performance; a possible explanation is that quantizing using $\delta = 0.2$ retained more noise and fluctuation in the time series, while quantizing using $\delta = 0.3$ may be more consistent with the actual onset of events.

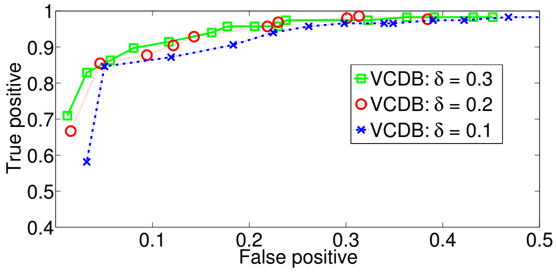


Figure 10: Experiment results under different δ.

Throughout the paper we have assumed the data are truthfully reported (though with noise/error). It is thus reasonable to question how robust is the prediction against possible (malicious) manipulation of the data used for training, a subject of increasing interest and commonly referred to as *adversarial* machine learning. For instance, an entity may attempt to set up fake networks with clean data (no malicious activities) but with fake reported incidents, and vice versa, to mislead the classifier. Without presenting a complete solution, which remains a direction of future research, below we test the robustness of our current prediction technique using two scenarios: (1) In the first we randomly flip the labels of victim organizations from 1 to 0; those flipped to 0 are now part of the non-victim group, thus contaminating the training data. (2) In the second scenario we do the opposite: randomly flip the labels of non-victim organizations, effectively adding them to the victim group. Incidentally, the former scenario is akin to under-reporting by randomly selected organizations.

Experimental results suggest no performance difference for case (1). The reason lies in the imbalance between the victim and non-victim population sizes. Recall that because of this, in our experiment we randomly select a subset of non-victim organizations with size comparable to the victim organizations (on the order of $O(1,000)$). Then in each training instance, the expected number of victims selected as part of the non-victim set is no more than $N_v \cdot O(1,000)/N$, with N_v denoting the number of fake non-victims and N the total number of non-victims. Since $N \sim O(1,000,000)$, even if one is able to inject $N_v \sim O(100)$ victims into the non-victim population, on average no more than one fake non-victim will actually be selected for training, resulting in negligible contamination effect unless such alterations can be done on a scale larger than the actual victim population.

For case (2), we indeed observe performance degradation, albeit slight, in the true positive, as shown in Fig. 11 at the 20% contamination level (20% of non-victim organization labels are flipped).

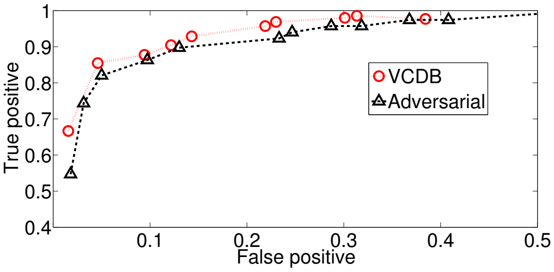


Figure 11: Adversarial case (2) with 20% contamination.

5.4 Incident Reporting

One of the main obstacles in studies of this nature is the acquisition of high quality incident data, without which we can neither train nor verify with confidence. Our result here demonstrates that machine learning techniques have the power to make accurate incident forecasts, but data collection is lagging by comparison. The research community would benefit enormously from more systematic and uniform incident reporting.

6 Related Work

As mentioned in the introduction, a large part of the literature focuses on detection rather than prediction. The work in [44] is one such example. Among others, Lee et al. [36] built sophisticated Hidden Markov Model techniques to detect spam deobfuscation, and in [57] Wang et al. applied (adversarial) machine learning techniques to the detection of malicious accounts on Weibo.

Relatively fewer studies have focused on prediction; even fewer are on the type of prediction presented in this paper where the predicted variable (classifier output) is of a different type from the input variables (feature input). For instance, the predictive IP-based blacklist works in [50, 30] have the same input and output variables (content of the blacklist). Similarly, in [54] the evolution of spam of a certain prefix is predicted using past spam activities as input. Predictive studies similar to ours include the aforementioned [51] that predicts whether a website will turn malicious by using textual and structural analysis of a webpage. The performance comparison has been given earlier. It is worth pointing out that the intended applications are also different: whereas webpage maliciousness prediction can help point to websites needing improvement or maintenance, our prediction on the organizational level can help point to networks facing heightened probability of a broader class of security problems. Also as mentioned earlier, our study [48] examines the prediction of incident types, conditional on an incident occurring, by using an array of industry, business and web visibility/population information. Other predictive studies include [28], where it is shown that by analyzing user browsing behavior one can predict whether a user will encounter a malicious page (attaining a 87% accuracy), [52], where risk factors are identified at the organization level (industry sector and number of employees) and the individual level (job type, location) that are positively or negatively correlated with experiencing spear phishing targeted attacks, and [53], where risk factors for web server compromise are identified through analyzing features from sampled web servers.

Also related are studies on reputation systems and profiling of networks. These include e.g., [26], a reputation assigning system trained using DNS features, reputation systems [8, 9] based on monitoring Internet traffic data, and those studied in [29, 46].

7 Conclusion

In this study, we characterize the extent to which cyber security incidences can be predicted based on externally observable properties of an organization's network. Our method is based on 258 externally measurable features collected from a network's mismanagement symptoms and malicious activity time series. Using these to train a Random Forest classifier, it is shown that we can achieve fairly high accuracy, such as a combination of 90% true positive rate and 10% false positive rate. We further analyzed the relative importance of the features sets in the prediction performance, and showed our prediction outcome for the top data breaches in 2014.

Acknowledgement

This work is partially supported by the NSF under grant CNS 1422211, CNS 1409758, CNS 1111699 and by the DHS under contract number HSHQDC-13-C-B0015.

References

[1] AFRINIC whois database. http://www.afrinic.net/services/whois-query.

[2] APNIC whois database. http://wq.apnic.net/apnic-bin/whois.pl.

[3] ARIN whois database. https://www.arin.net/resources/request/bulkwhois.html.

[4] Composite Blocking List. http://cbl.abuseat.org/.

[5] DShield. http://www.dshield.org/.

[6] Evernote resets passwords after major security breach. http://www.digitalspy.co.uk/tech/news/a462959/evernote-resets-passwords-after-major-security-breach.html.

[7] Global Reputation System. http://grs.eecs.umich.edu//.

[8] Global Security Reports. http://globalsecuritymap.com/.

[9] Global Spamming Rank. http://www.spamrankings.net/.

[10] Google Kenya and Google Burundi hacked by 1337. http://thehackersmedia.blogspot.com/2013/09/google-kenya-google-burundi-hacked-by.html.

[11] hpHosts for your pretection. http://hosts-file.net/.

[12] LACNIC whois database. http://lacnic.net/cgi-bin/lacnic/whois.

[13] Multiple DNS implementations vulnerable to cache poisoning. http://www.kb.cert.org/vuls/id/800113.

[14] Open Resolver Project. http://openresolverproject.org/.

[15] OpenBL. http://www.openbl.org/.

[16] PhishTank. http://www.phishtank.com/.

[17] Rf classifier. http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html.

[18] RIPE whois database. https://apps.db.ripe.net/search/query.html.

[19] SpamCop Blocking List. http://www.spamcop.net/.

[20] SURBL: URL REPUTATION DATA. http://www.surbl.org/.

[21] Syrian hacker Dr.SHA6H hacks and defaces City of Mansfield, OH website. http://hackread.com/syrian-hacker-dr-sha6h-hacks-and-defaces-city-of-mansfield-oh-website-for-free-syria.

[22] The SPAMHAUS project: SBL, XBL, PBL, ZEN Lists. http://www.spamhaus.org/.

[23] UCEPROTECTOR Network. http://www.uceprotect.net/.

[24] WPBL: Weighted Private Block List. http://www.wpbl.info/.

[25] Agrawal, T., Henry, D., and Finkle, J. JPMorgan hack exposed data of 83 million, among biggest breaches in history. http://www.reuters.com/article/2014/10/03/us-jpmorgan-cybersecurity-idUSKCN0HR23T20141003, October 2014.

[26] Antonakakis, M., Perdisci, R., Dagon, D., Lee, W., and Feamster, N. Building a Dynamic Reputation System for DNS. In *Proceedings of the 19th USENIX Security Symposium* (Berkeley, CA, USA, August 2010).

[27] Bishop, C. M., et al. *Pattern Recognition and Machine Learning*, vol. 1. Springer New York.

[28] Canali, D., Bilge, L., and Balzarotti, D. On the Effectiveness of Risk Prediction Based on Users Browsing Behavior. In *ASIA CCS '14* (New York, NY, USA, June 2014), ACM, pp. 171–182.

[29] Chang, J., Venkatasubramanian, K. K., West, A. G., Kannan, S., Lee, I., Loo, B. T., and Sokolsky, O. AS-CRED: Reputation and Alert Service for Interdomain Routing. vol. 7, pp. 396–409.

[30] Collins, M. P., Shimeall, T. J., Faber, S., Janies, J., Weaver, R., De Shon, M., and Kadane, J. Using Uncleanliness to Predict Future Botnet Addresses. In *Proceedings of ACM IMC* (San Diego, California, USA, October 2007), pp. 93–104.

[31] Consortium, T. W. A. S. Web-Hacking-Incident-Database. http://projects.webappsec.org/w/page/13246995/Web-Hacking-Incident-Database.

[32] Durumeric, Z., Kasten, J., Bailey, M., and Halderman, J. A. Analysis of the HTTPS Certificate Ecosystem. In *Proceedings of ACM IMC* (Barcelona, Spain, October 2013), pp. 291–304.

[33] Durumeric, Z., Wustrow, E., and Halderman, J. A. ZMap: Fast Internet-Wide Scanning and its Security Applications. In *Proceedings of the 22nd USENIX Security Symposium* (Washington, D.C., August 2013), pp. 605–620.

[34] Hubert, A., and Mook, R. V. Measures for Making DNS More Resilient against Forged Answers. RFC 5452, January 2009.

[35] Krebs, B. The target breach, by the numbers. http://krebsonsecurity.com/2014/05/the-target-breach-by-the-numbers/, May 2014.

[36] Lee, H., and Ng, A. Y. Spam Deobfuscation using a Hidden Markov Model. In *In Conference on Email and Anti-Spam* (July 2005).

[37] Liaw, A., and Wiener, M. Classification and Regression by randomForest. http://CRAN.R-project.org/doc/Rnews/, 2002.

[38] Lindberg, G. Anti-Spam recommendations for SMTP MTAs. BCP 30/RFC 2505, 1999.

[39] Ma, J., Saul, L. K., Savage, S., and Voelker, G. M. Beyond Blacklists: Learning to Detect Malicious Web Sites from Suspicious URLs. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, June 2009), KDD '09, ACM, pp. 1245–1254.

[40] Mahajan, R., Wetherall, D., and Anderson, T. Understanding BGP misconfiguration. In *Proceedings of SIGCOMM '02* (August 2002), vol. 32, ACM, pp. 3–16.

[41] of Oregon, U. Route Views Project. http://www.routeviews.org/.

[42] Passeri, P. Hackmageddon.com. http://hackmageddon.com/.

[43] Prince, B. Top data breaches of 2014. http://www.securityweek.com/top-data-breaches-2014, December 2014.

[44] Qian, Z., Mao, Z. M., Xie, Y., and Yu, F. On Network-level Clusters for Spam Detection. In *Proceedings of the Network and Distributed System Security Symposium (NDSS '14)* (San Diego, CA, March 2010).

[45] Ramachandran, A., and Feamster, N. Understanding the Network-level Behavior of Spammers. In *Proceedings of SIGCOMM '06* (August 2006), vol. 36, ACM, pp. 291–302.

[46] Resnick, P., Kuwabara, K., Zeckhauser, R., and Friedman, E. Reputation Systems. *Commun. ACM 43*, 12 (December 2000), 45–48.

[47] Romanosky, S. Comments on incentives to adopt improved cybersecurity practices noi. http://www.ntia.doc.gov/federal-register-notice/2013/comments-incentives-adopt-improved-cybersecurity-practices-noi, April 2013.

[48] Sarabi, A., Naghizadeh, P., Liu, Y., and Liu, M. Prioritizing Security Spending: A Quantitative Analysis of Risk Distributions for Different Business Profiles. In *the Annual Workshop on the Economics of Information Security (WEIS)* (June 2015).

[49] Sidel, R. Home depot's 56 million card breach bigger than target's. http://www.wsj.com/articles/home-depot-breach-bigger-than-targets-1411073571, September 2014.

[50] Soldo, F., A., L., and Markopoulou, A. Predictive Blacklisting as an Implicit Recommendation System. In *INFOCOM, IEEE* (March 2010), pp. 1–9.

[51] Soska, K., and Christin, N. Automatically Detecting Vulnerable Websites Before They Turn Malicious. In *Proceedings of the 23rd USENIX Security Symposium* (San Diego, CA, August 2014).

[52] Thonnard, O., Bilge, L., Kashyap, A., and Lee, M. Are You at Risk? Profiling Organizations and Individuals Subject to Targeted Attacks. In *Financial Cryptography and Data Security* (January 2015).

[53] Vasek, M., and Moore, T. Identifying Risk Factors for Webserver Compromise. In *Financial Cryptography and Data Security*. Springer, March 2014, pp. 326–345.

[54] Venkataraman, S., Brumley, D., Sen, S., and Spatscheck, O. Automatically Inferring the Evolution of Malicious Activity on the Internet. In *Proceedings of the Network and Distributed System Security Symposium (NDSS '14)* (San Diego, CA, February 2013).

[55] VERIS. VERIS Community Database (VCDB). http://veriscommunity.net/index.html.

[56] Verizon. Data Breach Investigations Reports (DBIR) 2014. http://www.verizonenterprise.com/DBIR/.

[57] Wang, G., Wang, T., Zheng, H., and Zhao, B. Y. Man vs. Machine: Practical Adversarial Detection of Malicious Crowdsourcing Workers. In *Proceedings of the 23rd USENIX Security Symposium* (San Diego, CA, August 2014), pp. 239–254.

[58] Zhang, J., Durumeric, Z., Bailey, M., Karir, M., and Liu, M. On the Mismanagement and Maliciousness of Networks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS '14)* (San Diego, CA, February 2014).

APPENDIX

Incident Dataset

A snapshot of sample incident reports from VCDB dataset (Table 7).

Incident type	Time	Report summary
Web site defacement	May 2014	”ybs-bank.com” a Malaysian imitation of the real Yorkshire Bank website
Hacking	Apr. 2014	4chan hacked by person targeting information about users posting habits.
Web site defacement	N/A 2013	AR Argentina Military website hacked.
Server breach	N/A 2013	The systems of AdNet Telecom, a major Romania-based telecommunications services provider, have been breached.
Web site hacked	May 2013	Albany International Airport website hacked.
Private key stolen	Mar. 2014	Amazon Web Services, Inc.
Phishing	N/A 2013	Bolivian tourist site was compromised and a fraudulent secret shopper site was installed.

Table 7: Incidents from the VCDB Community Database

WebWitness: Investigating, Categorizing, and Mitigating Malware Download Paths

Terry Nelms[1,2], Roberto Perdisci[3,2], Manos Antonakakis[2], and Mustaque Ahamad[2,4]

[1]Damballa, Inc.
[2]Georgia Institute of Technology
[3]University of Georgia
[4]New York University Abu Dhabi

tnelms@damballa.com, perdisci@cs.uga.edu, manos@gatech.edu, mustaq@cc.gatech.edu

Abstract

Most modern malware download attacks occur via the browser, typically due to social engineering and drive-by downloads. In this paper, we study the "origin" of malware download attacks experienced by real network users, with the objective of improving malware download defenses. Specifically, we study the *web paths* followed by users who eventually fall victim to different types of malware downloads. To this end, we propose a novel *incident investigation system*, named WebWitness. Our system targets two main goals: 1) automatically *trace back and label* the sequence of events (e.g., visited web pages) preceding malware downloads, to highlight how users reach attack pages on the web; and 2) leverage these automatically labeled in-the-wild malware download paths to better understand current attack trends, and to *develop more effective defenses*.

We deployed WebWitness on a large academic network for a period of ten months, where we collected and categorized thousands of *live* malicious download paths. An analysis of this labeled data allowed us to design a new defense against drive-by downloads that rely on injecting malicious content into (hacked) legitimate web pages. For example, we show that by leveraging the incident investigation information output by WebWitness we can decrease the infection rate for this type of drive-by downloads by almost *six times*, on average, compared to existing URL blacklisting approaches.

1 Introduction

Remote malware downloads currently represent the most common infection vector. In particular, the vast majority of malware downloads occur via the browser, typically due to social engineering attacks and drive-by downloads. A large body of work exists on detecting drive-by downloads (e.g., [10, 11, 19, 23, 33, 40]), and a few efforts have been dedicated to studying social engineering attacks [6, 31, 37]. However, very little attention has been dedicated to investigating and categorizing the *web browsing paths* followed by users *before* they reach the web pages from which the attacks start to unfold.

Our Work. In this paper, we study the *web paths* followed by *real users* that become victims of different types of malware downloads, including social engineering and drive-by downloads. We have two primary goals: 1) provide context to the attack by automatically identifying and labeling the sequence of web pages visited by the user *prior to the attack*, giving insight into how users reach attack pages on the web; and 2) leverage these annotated in-the-wild malware download paths to better understand current attack trends and to *develop more effective defenses*.

To achieve these goals we propose a novel malware download *incident investigation system*, named WebWitness, that is designed to be deployed passively on enterprise scale networks. As shown in Figure 1, our system consists of two main components: an *attack path traceback and categorization* (ATC) module and a *malware download defense* (MDD) module. Given all (live) network traffic generated by a user's browsing activities within a time window that includes a malware download event, the ATC module is responsible for identifying and linking together all HTTP requests and responses that constitute the web path followed by the user from an "origin" node (e.g., a search engine) to the actual malware download page, while filtering out all other irrelevant traffic. Afterwards, a statistical classifier automatically divides all collected malware download paths into *update*, *social engineering* and *drive-by* attacks. We refer to the output of the ATC module as *annotated malware download paths* (AMP).

The AMPs are continuously updated as new malware downloads are witnessed in the live traffic, and can therefore be used to aid the study of recent attack trends. Furthermore, the AMP data is instrumental in designing and building new defenses that can be plugged into the MDD

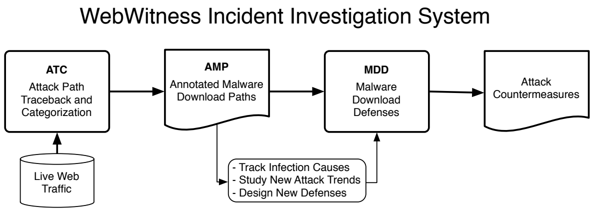

WebWitness Incident Investigation System
ATC
Attack Path Traceback and Categorization
AMP
Annotated Malware Download Paths
MDD
Malware Download Defenses
Attack Countermeasures
Live Web Traffic
- Track Infection Causes
- Study New Attack Trends
- Design New Defenses

Figure 1: WebWitness – high-level system overview.

module (see Figure 1). As an example, by investigating real-world web paths leading to drive-by malware downloads, we found that it is often possible to automatically trace back the domain names typically used in drive-by attacks to inject malicious code into compromised web pages (e.g., via the source of a malicious `script` or `iframe` tag). The injected code is normally used as an attack trigger, directing the browser towards an actual exploit and finally to a "transparent" malware download and execution. We empirically show that automatically discovering and promptly blocking the domain names serving the injected malicious code is a much more effective defense, compared to the more common approach of blacklisting the URLs that directly serve the drive-by browser exploits themselves or the actual malware executables (see Section 4.4).

Main Differences from Previous Work. Most previous works that study the network aspects of malware downloads focus on building malware detection systems, especially for drive-by exploit kits and related attacks (e.g., [13,30,38,40]).

Our work is different from these studies, because our goal is *not* to build a drive-by detection system; rather, we aim to passively trace back and automatically label the network events that *precede* different types of in-the-wild malware downloads, including both drive-by and social engineering attacks. We show that our investigation approach can aid in the design of more effective malware download defenses.

Some recent studies focus primarily on detecting malicious redirection chains as a way to identify possible malware download events [16,18,20,36]. WebWitness is different because we devise a *generic path trace back* approach that does not rely on the properties of redirection chains. Our work aims to provide *context* around malicious downloads by reconstructing the *full web path* (not just redirection chains) that brought the victim from an "origin" page to the download event. In addition WebWitness is able to classify the *cause* of the download (e.g., drive-by or social engineering) and to identify the roles of the domains involved in the attack (e.g., trick page, code injection, exploit, or malware hosting). We further discuss related work in Section 6.

Summary of Contributions. In summary, we make the following contributions:

- We investigate the *web paths* followed by *real network users* who eventually fall victim to different types of malware downloads, including social engineering and drive-by downloads. Through this investigation, we provide quantitative information on attack scenarios that have been previously explained only anecdotally or through limited case studies.

- To enable a continuous collection and study of web paths leading to malware download attacks, we build a system called WebWitness. Our system can automatically trace back and categorize in-the-wild malware downloads. We show that this information can then be leveraged to design more effective defenses against future malware download attacks.

- We deployed WebWitness on a large academic network for a period of ten months, where we collected and categorized thousands of *live* malicious download paths. Using these web paths, we were able to design a new defense against drive-by downloads that rely on injecting malicious content into (hacked) legitimate web pages. For example, we show that by leveraging the incident investigation information output by WebWitness, on average we can decrease the infection rate for this type of drive-by downloads by almost *six times*, compared to existing URL blacklisting approaches.

2 In-The-Wild Malware Download Study

Goals: In this section we report the results of a large study of *in-the-wild* malware downloads captured on a live academic network. Through this study, we aim to create a labeled dataset of download paths that can be used to design (including feature engineering), train, and evaluate the ATC and MDD modules of WebWitness shown in Figure 1. A detailed discussion of ATC and MDD is reported in Sections 3.

2.1 Collecting Executable File Downloads

To collect executable file downloads we use deep packet inspection to perform on-the-fly TCP flow reconstruction, keeping a buffer of all recent HTTP transactions (i.e., request-response pairs) observed on a live network. For each transaction, we check the content of the response to determine if it contains an executable file. If so, we retrieve all buffered HTTP transactions related to the client that initiated the download. Namely, we store all HTTP traffic a client generated preceding (and including) an executable file download; this allows us to study what *web path* users follow *before* falling victim to malware downloads. All data is saved in accordance with the policies set forth by our Institutional Review Board and are protected under a nondisclosure agreement.

2.2 Identifying Malicious Executables

Since many legitimate applications are installed or updated via HTTP (e.g., Windows Update), we immediately exclude all executable downloads from a manually-compiled whitelist of domain names consisting of approximately 120 effective second level domains (e2LDs) of popular benign sites (e.g., `microsoft.com`, `google.com`, etc.). For the remaining downloads, we scan them with more than 40 antivirus (AV) engines, using `virustotal.com`. In addition, we rescan them periodically because many "fresh" malware files are not immediately detected by AV scanners, allowing us to also take into account some "zero-day" downloads. We label a file as malicious if at least one of the top five AV vendors (w.r.t. market share) and a minimum of two other AVs detect it as malicious. The remaining downloads are considered benign until the rescan. In addition, we discard binary samples that are assigned labels that are too generic or based purely on AV detection heuristics.

2.3 Overview of Study Data

To gather our study data we deployed our collection agent (Section 2.1) on a large academic network serving tens of thousands of users for a period of 6 months. Notice that the system was deployed for a total of 10 months, with the study conducted in the first 6 months and the evaluation in the 4 months that followed (see Section 4 details on the evaluation). During these 6 months, we collected a total of $174,376$ executable downloads from domains that were not on our whitelist. Using the malicious executable identification process defined in Section 2.2, we labeled $5,536$ downloads as malicious.

However, many of these malicious downloads were related to *adware*. As we are primarily interested in studying *malware* downloads, because they are potentially the most damaging ones, we devised a number of "best effort" heuristics to separate *adware* from *malware*. For example, given a malicious file, if the majority of AV labels contain the term "adware", or related empirically derived keywords that identify specific unwanted applications (e.g., "not-a-virus", "installer", "PUP", etc.), we label the file as *adware*. The malicious executables not labeled as *adware* by our heuristics were manually reviewed to determine if they were truly *malware*. This resulted in $1,064$ *malware* downloads, with a total of 533 unique samples.

For these 533 unique malware downloads, we performed extensive manual analysis of their download paths, including reverse engineering web pages, heavy javascript deobfuscation, complex plugin content analysis, etc. This time-consuming analysis produced a set of labeled paths, with 164 drive-by, 41 social engineering and 328 update/drop malware download events.

Study Data Limitations: Our collection agent was deployed on an existing production network monitoring sensor. This sensor had limited hardware resources; in addition, our data collection system had to run alongside production software whose functionality could not be disrupted. We therefore collected downloads only during off-peak hours, due to traffic volumes that would oversubscribe the sensor and result in dropped packets during other periods of the day. Thus, the malicious downloads in our study represent only a sample of the ones that occurred during the six month monitoring period. In addition, our system monitors the network in a *purely passive* way; therefore, any malicious downloads preemptively blocked by existing defenses (e.g., URL blacklists such as Google Safe Browsing) were not observed. Yet, based on our extensive manual analysis, we believe the 533 malware downloads to be sufficiently diverse and representative of the overall set of malware downloads that occurred during our study period.

2.4 Download Path Traceback Challenges

One of the goals of our system is to automatically trace back the sequence of steps (i.e., HTTP transactions) that lead victims to be infected via a malware download. One may think that reconstructing the *web path to infection* is fairly easy, because we could rely on the `Referer` and `Location` header fields to link subsequent HTTP transactions together (see RFC2616). For example, a simple strategy would be to start from the download transaction and "walk back" the sequence of transactions by following the `Referer` header found in the HTTP requests.

Unfortunately, in practice download path traceback is much more difficult than it may seem at first. Depending on the particular version of the browser, JavaScript engine, and plugin software running on the client, the `Referer` and/or `Location` headers may be suppressed (e.g., see [14]), resulting in the inability to correctly reconstruct the entire sequence of download path transactions in a given network trace.

Deriving and Measuring Surrogate Features: As part of our study, we reviewed hundreds of malicious download traces. In most cases we cannot rely completely on the `Referer` and `Location` headers, and we therefore derive surrogate "referrer indicator" features and heuristics, which can be used to perform a more complete download path traceback. Next, we define each of the features we observed, and then provide a measure of how prevalent they are for malware download paths. While in this section we simply *measure their prevalence*, we later use these features to automate path traceback (Section 3).

First, let us more precisely define what we mean with *download path traceback*. Let T_d indicate an HTTP transaction carrying an executable file download initiated by client C. Given the recording of all web traffic generated by C during a time window preceding (and including) T_d, we would like to reconstruct the sequence of transactions $(T_1, T_2, \ldots, T_d)$ that led to the download, while filtering out all unrelated traffic. This sequence of transactions may be the consequence of both explicit user interactions (e.g., a click on a link) and actions taken by the browser during rendering (e.g., following a page redirection). Notice that the traffic trace we are given may contain a large number of transactions that are completely unrelated to the download path, simply because the user may have multiple browser tabs open and multiple web-based applications active in parallel. Thus, potentially producing a large amount of overlapping unrelated traffic.

Let T_1 and T_2 be two HTTP transactions. We found that the features/heuristics listed below can be used to determine whether T_1 is a *likely source* of T_2, therefore allowing us to "link" them with different levels of confidence. Table 1 summarizes the prevalence of each feature in both drive-by and social engineering downloads (we discuss how we can distinguish drive-by from social engineering later in Section 2.5). A detailed discussion of how WebWitness uses these features for automated download path traceback is given in Section 3.

(1) **Location**: According to RFC2616, if transaction T_2's URL matches T_1's `Location` header, it indicates that T_2 was reached as a consequence of a server redirection from T_1.

(2) **Referrer**: Similarly, if T_1's URL matches T_2's `Referer` header, this indicates that the request for T_2 originated (either directly or through a redirection chain) from T_1, for example as a consequence of page rendering, a click on a hyperlink, etc.

(3) **Domain-in-URL**: We observed that advertisement URLs often embed the URL of the page that displayed the ad. So, if T_1's domain name is "embedded" in T_2's URL, it is likely that T_1 was the "source" of the request, even though the `Referer` is not present. This is especially true if there is only a small time gap between the transactions.

(4) **URL-in-Content**: If T_1's response content includes T_2's URL (e.g., within an HTML or non-obfuscated JavaScript code), this indicates there is (potentially) a "source of" relationship that links T_1 to T_2.

(5) **Same-Domain**: By investigating numerous drive-by malware downloads, we found that in many cases the exploit code and the malware executable file itself are served from the same domain. This approach is likely chosen by the attackers because if the exploit is successfully served, it means that the related malicious domain is currently reachable and serving the malware file from the same domain helps guarantee a successful infection (a similar observation was made in [13]). Therefore, if T_1 and T_2 share the same domain name and are temporally close, this likely indicates that T_1 is the "source of" T_2.

(6) **Commonly Exploitable Content (CEC)**: In our observations, most drive-by downloads use "commonly exploitable" content (e.g., .jar, .swf, or .pdf files that carry an exploit) to compromise their victims. The exploit downloads the malicious executable; thus, if T_1 contains commonly exploitable content (CEC) and T_2 is an executable download that occurred within a small time delta after T_1, this indicates that T_1 may be the "source of" T_2.

(7) **Ad-to-Ad**: In some cases, we observed chains of ad-related transactions where the `Referer` and `Location` header are missing (e.g., due to JavaScript or plugin-driven redirections). Therefore, if T_1 and T_2 are consecutive ad-related requests (e.g., identified by matching their URLs against a large list of known ad-distribution sites) and were issued within a small time delta, this indicates there may be a "source of" relationship.

Table 1: Success rate of traceback method and "Source-of" relationships in malware download paths. The numbers indicate the percentage of analyzed download paths.

Traceback method success rate	Drive-by	Social Eng.
Only Referrer and Location	0%	53%
All surrogate referrer features	96%	95%

Feature	Drive-by	Social Eng.
Location	69%	73%
Referrer	97%	100%
Domain-in-URL	0%	5%
URL-in-Content	17%	17%
Same-Domain	97%	20%
CEC	5%	0%
Ad-to-Ad	6%	10%

As a confirmation to the fact that tracing back malware download paths is challenging, we found that not a single drive-by download in our dataset could be traced back by relying only on the `Referer` and `Location` headers. For example, even if 97% of the drive-by download paths contained at least one pair of requests linked via the `Referer`, all drive-by paths contained at least some subsequence of the path's transactions that could not be "linked" by simply using the `Referer` header.

For social engineering paths, we found that 53% of the downloads could be traced back using only the `Referer` and `Location` headers. When this was not possible, the main cause was the presence of requests made via JavaScript and browser plugins. In some cases, we were not able to fully trace back the download path. The cause for the majority of the untraceable drive-by (4%) and social engineering (5%) downloads, when using all the features, was missing transactions likely due to our system not observing all related packets.

2.5 Drive-by vs. Social Engineering

We label a malware download path as *social engineering* if explicit user interaction (e.g., a mouse click) is required to initiate a malware download. In contrast, we label as *drive-by* those malware downloads that are transparently delivered to the victim via a browser exploit. As mentioned earlier (Section 2.3), during our study, we were able to manually review and label 164 drive-by and 41 social engineering malware downloads.

What distinguishes drive-by from social-engineering: In the following we report the characteristics that we observed for different types of paths. In particular, some of these characteristics could be leveraged as statistical features to build a classifier that automatically distinguishes between drive-by and social engineering downloads (see Section 3). We also discuss characteristics of malware updates/drops that could be used to filter out download paths that belong neither to the drive-by nor to the social-engineering class. Table 2 summarizes the prevalence of each of the characteristics described below.

Table 2: Download path properties.

Feature	Drive-by	Social eng.
Candidate Exploit Domain Age	0	-
Drive-by URL Similarity	69%	0%
Download Domain Recurrence	0.6%	34%
Download Referrer	0.6%	95%
Download Path Length	6	7
User-Agent Popularity	95%	98%

(1) **Candidate Exploit Domain "Age":** Drive-by download attacks often exploit their victims by delivering exploits via files of popular content types such as `.jar`, `.swf`, or `.pdf` files; we simply refer to these file types as "commonly exploitable" content (CEC). For example, during our study, we found that 94% of the drive-by download paths at some point delivered the exploit via CEC. The domains serving these exploits tend to be short-lived compared to domains serving benign content of the same type. Therefore, CEC served from a recently registered domain is an indicator of a possible drive-by download path. On the other hand, none of the social engineering download paths we observed during our study had this property. Table 2 reports the median domain name "age", computed as the number of days of activities for the domain of a page serving CEC, measured over a very large passive DNS database. The median age is less than one day for drive-by paths, and is not indicated for social engineering paths, because none of the nodes in the social engineering path served content of the type we consider as CEC (the overall traffic traces included HTTP transactions that carried content such as `.swf` files, but none of those were on the download path).

(2) **Drive-by URL Similarity:** The majority of drive-by downloads (about 70% of our observations) are served by a small number of exploit kits. Therefore, in many cases the exploit delivery URLs included in drive-by download paths share a structural URL similarity to known exploit kit URLs. Table 2 reports the fraction of drive-by download paths that had a similarity to known exploit kit URLs greater than 0.8, measured using the approach proposed in [26].

(3) **Download Domain Recurrence:** Most domains serving drive-by and social engineering malware download are contacted rarely, and often only once by one particular client at the time of the attack. On the other hand, malicious software regularly checks for executable updates. To approximately capture this intuition, we measured the number of queries to the malware download domain. As shown in Table 2, only 0.6% of the malware download domains in our drive-by paths are queried mulitple times within a small time window (two days, in our measurements). The higher percentage of social engineering malware paths with download domain recurrence is due to the fact that a significant fraction of the ones we observed used a free file sharing website for the malware download and that we count the domain query occurrences in aggregate, rather than per client.

(4) **Download Referrer:** In case of social engineering attacks, the HTTP transaction that delivers the malicious file download tends to carry a `Referer`, usually due to the direct user interaction that characterizes them. On the other hand, drive-by attack malware file delivery happens via a browser exploit. The request initiated from the shell code typically does not have a `Referer` header. Similarly, malware updates/drops initiated by malicious applications are already running on a compromised machine, and usually do not carry any referrer information. Table 2 shows that only 0.6% of all drive-by paths, in contrasts to 95% of social engineering paths, carried a `Referer` in the download node.

(5) **Download Path Length:** Drive-by and social engineering attacks typically generate download paths consisting of several nodes, mainly because a user has to first browse to a site that eventually leads to the actual attack. In addition, the malware distribution infrastructure is often built in such ways that enables malware downloads "as a service", which entails the use of a number of "redirection" steps. In contrast, download paths related to malware updates or drops tend to be very short. Table 2 reports the median number of nodes for drive-by and social engineering paths. In case of malware updates/drops, the median length for the path was only one node.

(6) **User-Agent Popularity:** The download paths for both drive-by and social engineering downloads typically include several nodes that report a popular browser user-agent string, as the victims use their browser to reach the attack. On the other hand, in most cases of a malware drop/update, it is not the browser, but the update software making the requests. In practice, we observed that the majority of malware update download paths did not report a popular user-agent string (only 36% of them did). Table 2 reports the percentage of paths that include a popular user-agent string.

3 WebWitness

Inspired by our study of real-world malware download paths, we develop a system called WebWitness that can automate the investigation of new malware download attacks. The primary goal of this system is to provide *context* around malicious executable downloads. To this end, given a traffic trace that includes all web traffic recorded during a time window preceding (and including) a malicious executable file download, WebWitness automatically traces back and categorizes the web paths that led the victim to the malicious download event.

In this section, we describe the components of our system, which are shown in Figure 2.

3.1 ATC - Download Path Traceback

Given a malicious file download trace from a given client, WebWitness aims to trace back the *download path* consisting of the sequence of web pages visited by the user that led her to a malware download attack (e.g., via social engineering or to a drive-by exploit). As detailed in Section 2.4, the trace may contain many HTTP transactions that are unrelated to the download. Furthermore, it is not always possible to correctly link two related consecutive HTTP transactions by simply leveraging their HTTP `Referer` or `Location` headers.

To mitigate the limitations of referrer-only approaches and more accurately trace back the download path, we devise an algorithm that leverages the features and heuristics we identified during our initial study of in-the-wild malware downloads presented in Section 2.4. In summary, we build a *transactions graph*, where nodes are HTTP transactions within the download trace, and edges connect transaction according to a "probable source of" relationship (explained in detail below). Then, starting from the node (i.e., the HTTP transaction) related to the malware file download, we walk back along the most probable edges until we find a node with no predecessor, which we label as the "origin" of the download path. In the following, we provide more details on our traceback algorithm.

Transactions Graph. Let D be the dataset of HTTP traffic generated by host A before (and including) the download event. We start by considering all HTTP transactions in D, and construct a weighted directed graph $G = (V,E)$. The vertices are A's HTTP transactions and the edges represent the relation "probable source of" for pairs of HTTP transactions. As an example, the edge $e = (v_1 \rightarrow v_2)$ implies that HTTP transaction v_1 likely produced HTTP transaction v_2, either automatically (e.g., via a server-imposed redirection, javascript, etc.) or through explicit user interaction (e.g., via a hyperlink click). Thus, we can consider v_1 as the "source of" v_2. Each edge has a weight that expresses the level of confidence we have on the "link" between two nodes (the weights are ordinal so their absolute values are not important). For example, the higher the weight assigned to $e = (v_1 \rightarrow v_2)$, the stronger the available evidence in support of the conclusion that v_1 is the "source of" v_2 (edge weights are further discussed below). Also, let t_1 and t_2 be the timestamp of v_1 and v_2, respectively. Regardless of any available evidence for a possible edge, the two nodes may be linked only if $t_1 \leq t_2$.

Heuristics and Edge Weights. To build the graph G and draw its edges, we leverage the seven features that we indentified in Section 2.4. Specifically, given two nodes (essentially, two URLs) in the directed graph G described earlier, an edge $e = (v_1 \rightarrow v_2)$ is created if any of the seven features is satisfied. For example, if v_1 and v_2 can be related via the "Domain-in-URL", we draw an edge between the two nodes. We associate a weight to each of the seven features; the "stronger" the feature, the higher its weight. For example, we assign a weight value $w_e = 7$ to the "Location" feature, $w_e = 6$ to the "Referrer" feature, and so on, with the "Ad-to-Ad" receiving a weight $w_e = 1$. The weight values are conveniently assigned simply to express relative importance and precedence among the edges to be considered by our greedy algorithm. If more than one feature happens to link two nodes, the edge will be assigned a weight equal to the maximum weight among the matching features.

Traceback Algorithm. Once G has been built, we use a *greedy algorithm* to construct an approximate "backtrace path". We start from the graph node related to the executable download event, and walk backwards on the graph by always choosing the next edge with the highest weight. Consider the example graph in Figure 3, in which thicker edges have a higher weight. We start from the download node d. At every step, we walk one node backwards following the highest weight edge. We proceed until we reach a node with no predecessor, which we mark as the *origin* of the download path. If a node has more than one predecessor whose edges have the same weight, we follow the edge related to the predecessor node with the smaller time gap to the current node (measured w.r.t. the corresponding HTTP transactions).

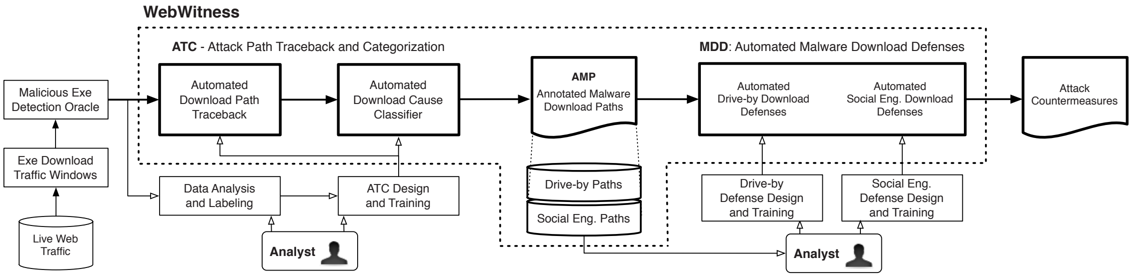

WebWitness
ATC - Attack Path Traceback and Categorization
MDD: Automated Malware Download Defenses
Malicious Exe Detection Oracle
Exe Download Traffic Windows
Live Web Traffic
Automated Download Path Traceback
Automated Download Cause Classifier
AMP
Annotated Malware Download Paths
Automated Drive-by Download Defenses
Automated Social Eng. Download Defenses
Attack Countermeasures
Data Analysis and Labeling
ATC Design and Training
Analyst
Drive-by Paths
Social Eng. Paths
Drive-by Defense Design and Training
Social Eng. Defense Design and Training
Analyst

Figure 2: WebWitness system details.

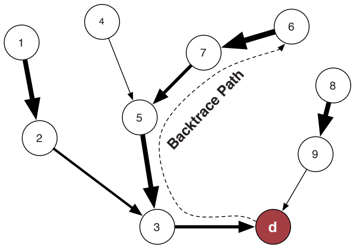


Figure 3: Example of download path traceback.

Possible False and Missing Edges: Naturally, the heuristics we use for tracing back the download path may in some cases add "false edges" to the graph or miss some edges. However, notice that these challenges are mitigated (though not always completely eliminated) by the following observations:

i) Our algorithm and heuristics aim to solve a much narrower problem than finding the correct "link" between all possible HTTP transactions in a network trace, because we are only concerned with tracing back a sequence of HTTP transactions that terminate into a malicious executable download.

ii) The "false edge" problem is mitigated by the fact that we always follow the strongest evidence. For example, consider Figure 3. Suppose the edge $(2 \rightarrow 3)$ was drawn due to rule (6), while edge $(5 \rightarrow 3)$ was drawn due to rule (2). In this case, even though edge $(2 \rightarrow 3)$ was mistakenly drawn (i.e., nodes 2 and 3 have no real "source of" relationship), the mistake is irrelevant, because our algorithm will choose $(5 \rightarrow 3)$ as part of the path, which is supported by stronger evidence.

iii) Our algorithm can output not only the sequence of HTTP transactions, but also the nature (and confidence) of every edge. Therefore, a threat analyst (or a downstream post processing system) can take the edge weights into account, before the reconstructed download path is used to make further decisions (e.g., remediation or takedown of certain domains in the download path).

3.2 ATC - Download Cause Classification

After we trace back the download path, we aim to label the reconstructed path as either *social engineering* or *drive-by* download. As shown in Figure 2, the output of this classification step allows us to obtain the annotated malware download paths (AMPs), which are then provided as input to the defense module (MDD).

While we are mainly interested in automatically identifying social engineering and drive-by download paths, we build a three-class classifier that can distinguish between three broad download causes, namely *social engineering*, *drive-by*, and *update/drop*. Essentially the *update/drop* class allows us to more easily identify and exclude malware downloads that are not caused by either social engineering or drive-by attacks.

To automatically classify the "cause" of an executable file download, WebWitness uses a supervised classification approach. First, we describe how we derive the features needed to translate malware download events into feature vectors that can be given as input to a statistical classifier. Then, we discuss how we derive the dataset used to train the classifier. To actually build the classifier, we used the random forest algorithm [7] (see Section 4).

Features: To discriminate between the three different classes, we engineered six statistical features that reflect, with a one-to-one mapping, the six characteristics of drive-by and social-engineering malware download paths that we discussed and measured in Section 2.5. For example, we measure binary feature (1) "Download Referrer" as true if the HTTP request that initiated the download has a `Referer` header; a numerical feature (2) representing the "age" of domains serving "commonly exploitable" content; etc.

Training dataset: To train the classifier, we use the dataset of in-the-wild malware download paths that we collected and manually labeled during our initial investigation of in-the-wild malware downloads discussed in Section 2.5. Our training dataset contained the following number of labeled download paths: 164 instances of *drive-by* download paths, 191 instances of *social engineering* paths, and 328 *update/drop* samples.

3.3 MDD - Drive-by Defense

The annotated download paths output by ATC provide a large and up-to-date dataset of real-world malware download incidents, including the web paths followed by the victims (see Figure 2). This information is very useful for studying new attack trends and developing more effective defenses. As new defenses are developed, they can be plugged into the MDD module, so that as new malware download paths are discovered we can automatically derive appropriate countermeasures.

As an example that demonstrates how WebWitness can enable the development of more effective malware download defenses, we develop a new defense against drive-by download attacks based on code injections. While code injection attacks are not new, current defenses rely mainly on blacklisting the URLs serving the actual drive-by exploit or malware download, rather than blocking the URLs from which malicious code is injected. Our results (Section 4) show that by automatically tracing back drive-by download paths and identifying the code injection URLs, we can enable better defenses against future malware attacks.

Identifying code injection URLs: Given a *drive-by* download path output by the ATC module, we aim to automatically identify the *landing*, *injection*, and *exploit* nodes within the download path.We tackle this problem using a supervised classification approach. Namely, we train a separate classifier for each of the three types of nodes on a drive-by download path. The final output is a labeled drive-by download path.

Exploit Page Classifier: The exploit classifier takes as input a drive-by download path and labels its nodes as *exploit* or *non-exploit*. We define an exploit node as a page that carries content that exploits a vulnerability on the victim's machine, causing it to eventually download a malicious executable. The search for exploit nodes proceed "backwards", starting from the node prior to the executable download and ending at the root. It is not uncommon to have more than one exploit node in one path (e.g., some exploit kits try several exploits before success). Thus, multiple nodes could be labeled as *exploit*.

To build the classifier, we use the following features:

(1) *Hops to the download page*. Number of nodes on the download path between the considered node and the final malware download node. *Intuition*: It is typical for the exploit node to only be a few hops away from the actual download. In many cases, the node prior to the download event is an exploit node, because once the exploit succeeds the executable is downloaded immediately.

(2) *"Commonly exploitable" content*. Boolean feature that indicates if a node contains content for Java, Silverlight, Flash or Adobe Reader. *Intuition*: Browser plug-ins are a popular exploitation vector. The exploit is typically delivered though their content.

(3) *Domain age*. The number of days since the first observation of the node's effective second level domain in a large historic passive DNS database. *Intuition*: Exploit domains tend to be short-lived and often only active for one day.

(4) *Same domain*. Boolean feature that is true if the node's domain is equal to the download domain. *Intuition*: It is common for the exploit and download to be served by the same domain, as also noted in [13].

Landing Page Classifier: Once the exploit node(s) is labeled, we attempt to locate the landing page URL. Essentially, the landing page is the web page where the drive-by attack path begins. Often, the landing page itself is a non-malicious page that was previously compromised (or "hacked"). The landing page classifier calculates the probability that a node preceding the exploit node (labeled by the exploit page classifier discussed earlier) is a landing page. Nodes with a probability higher than a tunable detection threshold (50% in our experiments) are classified as "candidate landing" nodes. If there are multiple candidates, the one with the highest probability is labeled as the landing node.

To label a node as either *landing* or *non-landing*, we engineered the following statistical features:

(1) *Hops to the exploit page*. This feature set consists of the number of non-redirect nodes and unique effective second level domains between the node and the exploit node. *Intuition*: Often, all the nodes between the landing and exploit node are redirects [36]. Also, most drive-by downloads use one to three types of malicious domains (injection, exploit, download). Therefore, in most cases there are zero or one domains (the one being the injection domain) on the download path between the landing and exploit nodes.

(2) *Domain age*. We use two features based on domain age. The first feature is the age of the node's effective second level domain as computed from a passive DNS database. *Intuition*: The domains associated to ("hacked") landing pages tend to be long-lived. Furthermore, "older" landing pages tend to offer more benefits to the attackers, as they often attract more visitors (i.e., potential victims), because it takes time for legitimate pages to become popular. The second feature is the age of the oldest domain between the node and the exploit node. *Intuition*: Nodes on the download path between the landing and exploit nodes tend to be less than a year in age. This is because they are typically malicious and recently registered.

(3) *Same domain*. Boolean feature that is true if the node's domain is equal to the exploit domain. *Intuition*: It is uncommon for an exploit to be served

from the same domain as the landing page. They are typically kept separate because installing an exploit kit on a compromised website may increase the likelihood of detection by the legitimate site's webmaster. In addition, it is much easier to manage a centralized exploit kit server than keep all the compromised websites up-to-date with the latest exploits.

Injection Page Classifier: We define the injection page to be the source of the code inserted into the "hacked" landing page. Typically, the injection and exploit nodes are separate and are served via different domain names. This provides a level of indirection that allows the exploit domain to change without requiring an update to the landing page. The injection node by definition is a successor to the landing page, but depending on the injection technique it may or may not be directly present in the download path traced back by the ATC module. Therefore, the classifier calculates the injection page probability for each direct successor of the landing node in the transactions graph, instead of only considering nodes in the reconstructed download path. The successor of the landing page node with the highest probability is labeled as the injection page node.

To identify the *injection* page, for each successor of the *landing* node we measure the following features:

(1) *On path*. Boolean feature indicating if the node is on the download path. *Intuition*: Being on the download path and a successor of the landing page, makes it a good candidate for the injection node. However, the injection node is not always on the download path due to the structure of some drive-by downloads.

(2) *Advertisement*. Boolean feature that is true if the node is an ad. *Intuition*: By definition, the injection page is not an ad, but code injected into the landing page. It is common for ads that are not related to the malicious download to be served on a landing page. This feature help us exclude those ad nodes.

(3) *Domain age*. The number of days since the first observation of the node's effective second level domain in passive DNS. *Intuition*: Injection pages typically have the sole purpose of injecting malicious code. They are rarely hosted directly on compromised pages, because this would expose the malicious code to cleanup by the legitimate site owners, ending the attacker's ability to exploit visitors. Consequently, injection pages are hosted on "young" domains that are typically active for the lifetime of a website compromise.

(4) *Successors*. There are two features that are derived from the node's successors. First is the number of direct successors. *Intuition*: Injection nodes tend to have only one direct successor. They typically perform an HTTP redirect or dynamically update the DOM to include the URL of the exploit domain. Benign pages often have more than one direct successor because they load content from many different files or sources. The second feature is boolean and it is true if one of the node's successors is on the download path. It indicates there is a possible "source of" relationship between it and a node on the download path. Even though the node itself may not be on the download path.

(5) *Same domain*. There are two boolean features that compare domain names. The first checks for equality between the node's domain and the landing domain. *Intuition*: It is uncommon for the landing domain to equal the injection domain for reasons similar to those described in the landing page classifier's "same domain" feature described earlier. The second feature compares the node's domain to the exploit domain. *Intuition*: In approximately 70% of the observations in our measurement study (Section 2), the exploit and injection domains were different.

4 Evaluation

In this section, we evaluate WebWitness' ATC and MDD modules. We also demonstrate the overall benefits of our new defense approach against drive-by downloads, by measuring the effectiveness of blacklisting the injection domains discovered by WebWitness. We show that while blacklisting the injection domains provides a better defense, compared to blacklisting only the exploit and download domains, injection domains appear very rarely in current blacklists, including Google Safe Browsing and a variety of large public blacklists.

4.1 ATC - Download Cause Classification

The download cause classifier uses a supervised learning approach to label each download path as either *social engineering*, *drive-by* or *update/drop* (Section 3.2). To evaluate its accuracy, we use WebWitness to traceback and classify all malicious downloads collected from the large academic network (Section 2) in the months *following* our initial study and development of the system. Specifically, all download events and samples used during evaluation have *no overlap* with the data we used for the study presented in Section 2, to design WebWitness' features and heuristics, or to train our classifiers. Each malicious download observed during the *testing period* was then classified as one of the following: *drive-by*, *social engineering* or *update*. From each of the three predicted classes we randomly sampled 50 downloads for manual verification. We limited the sample size to a total of 150 downloads because of the extensive manual analysis required to determine the ground truth, including reverse engineering web pages, heavy javascript deobfuscation, complex html and plugin content analysis, etc. This time consuming review process allowed us to identify the correct web path and the *true cause* of download, creating our *ground truth* for the evaluation. Table 3 reports the confusion matrix for the cause classifier.

Table 3: Cause Classifier - Confusion Matrix Results

		Predicted Class		
	Class	Drive-by	Social	Update/Drop
Ground Truth	Drive-by	47	1	0
	Social	2	46	3
	Update/Drop	1	3	47

The classifier correctly labeled over 93% of the downloads. Notice that these results represent the overall system performance of the ATC module, because the download paths used in the experiment (i.e., input to the cause classifier) were extracted using our download path traceback algorithm (Section 3.1). The two social engineering samples classified as drive-by downloads both had commonly exploitable content (CEC) on the download path. They were misclassified even though the CEC domain ages were greater than 200 days. The three update/drop samples classified as social engineering was caused by invalid download paths resulting from the false edges described in the next section. Finally the three social engineering downloads misclassified as update/drop was a result of small downloads paths (all were length 3) and high download domain recurrence (all greater than 20 of the 48 hourly buckets).

4.2 ATC - Download Path Traceback

To evaluate the accuracy of our download path traceback algorithm (Section 3.1), we use the 150 manually reviewed downloads; i.e., our ground truth, from Section 4.1. For path traceback, we consider two types of errors for review: (1) missing nodes: the traceback stops short, before reaching the origin of the download path (recall that the traceback algorithm works its way backwards from the download node to the path origin); (2) false node: a node that should not appear in the download path. Table 4 summarizes the results of our evaluation.

Table 4: Download Path Traceback Results.

	Paths	Correctly Traced Back	Missing	False
Drive-By	48	45	3	0
Social	51	46	2	3
Update/Drop	51	47	0	4

The results show that 92% of the download paths were correctly traced back by our system. The 5 with missing nodes all had a *referer* header in the origin node's request, but a matching URL was not contained in the trace. This was likely due to our system not observing all the packets related to those transactions. The 7 with the false nodes were all caused by the "same-domain" heuristic incorrectly connecting the paths of an update and a social engineering download. The heuristic failed because the updates were performed by a malicious executable seconds after the user was socially engineered into downloading it from the same domain as the update.

4.3 MDD - Detecting Injection Domains

As discussed in detail in Section 3.3, we aim to automatically identify the malicious code injection domains often employed in drive-by download attacks. To achieve this goal, we use a cascade of three classifiers: an *exploit*, a *landing*, and an *injection* classifier (Section 3.3). In the following, we evaluate the performance of each one.

To build the training dataset, we use 117 drive-by malware downloads collected and manually labeled during our six-month malware study described in Section 2. These 117 drive-by paths contained 246 exploit nodes (notice that it is not uncommon for a drive-by attack to serve more than one exploit, especially when the first exploit attempt fails). There is only one landing node and one injection node per download path.

Table 5: Node Labeling for Drive-By Download Paths

Experiment	Classifier	Correctly Labeled	Incorrectly Labeled
Cross-Validation	Exploit	99.19%	0%
	Landing	96.58%	0.17%
	Injection	94.87%	0.07%

We performed 10-fold cross-validation tests using the dataset described above. Table 5 summarizes the results. As can be seen, all classifiers are highly accurate. The results of the the injection page classifier represent the performance of the final injection domain detection task. This is due to fact that all tests were conducted using the three classifiers (exploit, landing, and injection) in cascade mode to mirror an actual deployment of WebWitness' MDD module. Thus, overall, we obtained a minimum of 94.87% detection rate at 0.07% false positives.

There were a total 7 domains mislabeled as injection by our system. The most common error was labeling the exploit domain as the injection domain; i.e., missing the fact that a separate injection domain existed. This was the case for 5 of the 7 mislabeled domains. Since these domains are malicious, blacklisting them will not cause false positives. The other two domains were benign. One of them had an Alexa rank over $260,000$ and the other above $1,600,000$. To mitigate such false positives, the newly discovered injection domains could be reviewed by analysts before blacklisting. As WebWitness provides the analyst with full details on the traffic collected before the download and the reconstructed download path, this information can make the analyst's verification process significantly less time-consuming.

4.4 MDD - Defense Efficacy & Advantages

Domain name and URL blacklisting are commonly practiced defenses [2]. However, blacklists are only effective if the blacklisted domains remain in use for some period of time after they are detected. The longer-lived a malicious domain, the more useful it is to blacklist it. As discussed in Sections 3.3 and 4.3, WebWitness is able not only to identify the domains from which malware files are downloaded, but also to identify the malicious code injection and exploit domains within drive-by malware download paths. Clearly, these domains are all candidates for blacklisting.

To evaluate the efficacy of blacklisting the code injection domains, we demonstrate the advantages this provides compared to the currently more common approach of blacklisting the exploit and download domains. To this end, we use a set of 88 "complete" injection-based drive-by download paths that we were able to collect from a large academic network. These samples were "complete" paths in the sense that they were manually verified to have an injection, exploit, and malware download node (and related domain).

We evaluate the effect of blocking the different types of drive-by path domains by counting the number of potential victims that would be saved by doing so. Specifically, we define a potential victim as a unique client host visiting a blacklisted domain. Notice that the actual number of hosts that get infected may be smaller than the number of potential victims, because only some of the hosts that visit a malicious domain involved in a drive-by download attack will "successfully" download and run the malware file (e.g., because an anti-virus blocked the malware file from running on the machine). However, we can use the potential victim count to provide a relative comparison on the effectiveness of blacklisting injection versus exploit and malware download domains.

To count the potential victims, we rely on a very large passive DNS (pDNS) database that spans multiple Internet Service Providers (ISPs) and corporate networks. This pDNS dataset stores the historic mappings between domains and IP addresses, and also provides a unique source identifier for each host that queries a given domain name. This allows us to identify all the unique hosts that queried a given domain in a given timeframe (e.g., a given day). For each injection-based drive-by download paths in our set, we compute the potential victims saved by counting the number of unique hosts that query the injection, exploit, and file download domains in the 30 days following the date when we observed and labeled the download event. Figure 4 shows our results, in which day-0 is the day when we detected a malicious download path (the victims counts are aggregated, per day, for all hosts contacting a malicious domain). We can immediately see that the number of potential victims that query the exploit or file download domains rapidly drops as the exploit domain ages. On the other hand, injection domains are longer lived, and blacklisting them would prevent a much larger number of potential victims from being redirected to new (unknown) and frequently churning exploit and file download domains. Blacklisting the injection domain saves almost 6 *times* more potential victims, compared to blacklisting the exploit domain.

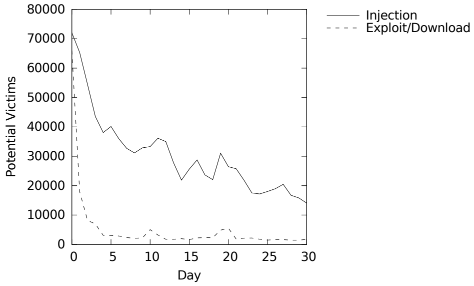


Figure 4: Potential victims saved by blocking the injection versus exploit/download domains on drive-by paths.

4.5 Blacklists & Google Safe Browsing

In this section, we aim to gain additional insights into the advantages that could be provided by our WebWitness' MDD module, compared to existing domain blacklists.

Public Domain Blacklists First, given the entire set of malicious domain names related to drive-by downloads discovered during our study and deployment of WebWitness, we counted how many of these domains appeared in popular public blacklists. We also measured the delay between when we first discovered the domain on a malware download path and when it appeared on a blacklist. This was possible because we repeatedly collected all domain names reported by the following set of public blacklists every day for more than a year: support.clean-mx.de, malwaredomains.com, zeustracker.abuse.ch, phishtank.com and malwaredomainlist.com. Table 6 summarizes our findings.

Table 6: Public Blacklisting Results.

	Uniq. Domains		Days: Detect to Blacklist		
	Observed	Blacklisted	Min.	Med.	Mean
Exploit/Download	152	9	1	20	29
Injection	52	6	20	31	36

As shown in Table 6, from all drive-by download paths that we were able to identify, reconstruct, and label, we collected a total of 52 unique drive-by code injection domains and 152 unique drive-by exploit and malware file download domains. Overall, less than 10% of these domains ever appeared on a public blacklist. As we can see, more exploit/download domains (a total of 9) were blacklisted, compared to the injection domains (only 6). Furthermore, we can see that the minimum time it took for an injection domain to appear in at least one blacklist was 20 days, whereas some exploit domains were blacklisted almost immediately (after only one day).

Because injection domains are typically longer lived than exploit domains, and because the same injection domain is often used throughout the course of a drive-by download campaign to redirect users to different (short-lived) exploit domains, identifying and blocking injection domains has a significant advantage. By helping to quickly identify and blacklist injection domains, WebWitness enables the creation of better defenses against drive-by downloads, thus helping to significantly reduce the number of potential malware victims, as we also demonstrated in the previous Section 4.4.

Google Safe Browsing For the last few weeks of our deployment of WebWitness, we checked the domain names related to the drive-by download paths reconstructed by our system against Google Safe Browsing (GSB) [2]. Specifically, given a malware download path and its malicious domains, we queried GSB on the next day, compared to the day the malware download was observed. Overall, during this final deployment period we observed 34 drive-by download paths. GSB detected a total of 6 malicious domains that were related to only 4 out of the 34 downloads. The domains GSB detected were used to serve drive-by exploits, the malware file themselves, or were related to ads used to lead the victims to a browser exploit. *None of the domains detected by GSB were injection domains*, even though our 34 download paths included 12 unique injection domains.

It is important to notice, however, that while GSB detected malicious domains related to only 4 out of our 34 drive-by download paths, there may be many more malware downloads that WebWitness cannot observe, simply because they are blocked "up front" by GSB. Because WebWitness passively collects malware download traces from the network whenever a malicious executable file download is identified in the traffic, it is very possible that in many cases GSB simply prevented users who were about to visit a drive-by-related domain from loading the malicious content, and therefore from downloading the malware file in the first place. Nonetheless, the fact that WebWitness automatically discovered 30 drive-by download paths that were not known to GSB demonstrates that our system can successfully complement existing defenses.

4.6 Case Studies

4.6.1 Social Engineering

Figure 5 shows the *download path* for an in-the-wild social engineering attack, including the "link" relationships between nodes in the path. The user first performs a search on `www.youtube.com` (A) for a "*facebook private profile viewer*", which is the *root* of the path. Next, the user clicks on the top search result leading to a "*trick*" page on `www.youtube.com` (B), which hosts a video demonstrating a program that supposedly allows the viewing of the private profiles of Facebook users. A textual description under the video provides a link to download a "profile viewer" application through a URL shortener `goo.gl` (C). This shortened URL link redirects the user to `uploading.com` (D), a free file sharing site that prompts the user with a link to start the download. This leads to another `uploading.com` (E) page that thanks the user for downloading the file and opens a new `uploading.com` (F) page that includes an `<iframe>` with source `fs689.uploading.com` (G), from which the executable file is downloaded. The file is labeled as "Trojan Downloader" by some anti-virus scanners.

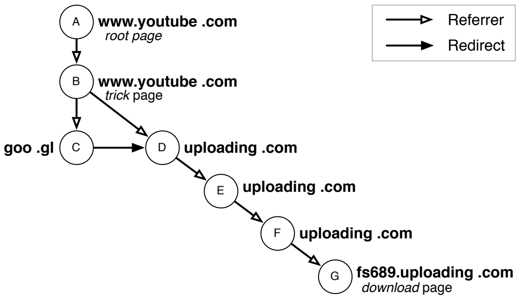


Figure 5: Social engineering download example.

Notice that no exploit appears to be involved in this attack, and that the user (highly likely) had to explicitly click on various links and on the downloaded malware file itself to execute it.

4.6.2 Drive-by

Figure 6 shows the *download path* related to an in-the-wild drive-by download.

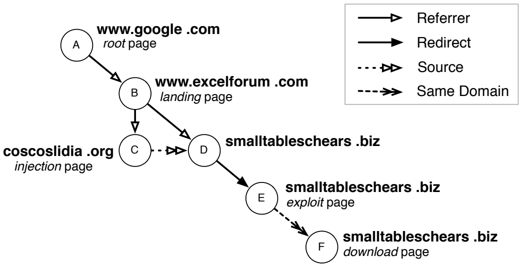


Figure 6: Drive-by download example.

The download path originates from (A) `www.google.com` (the *root* page), where the user entered the search terms "*add years and months together*." The first link in the search results, which the user clicked on, is for a webpage (B) on `www.excelforum[dot]com` (the *landing* page). Sadly, the page the user landed

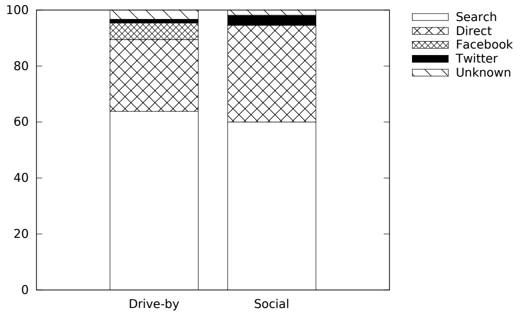


Figure 7: “Root” of malware download paths.

on was compromised several days earlier, resulting in the addition of a `<script>` tag with source at `coscoslidia[dot]org`, which is the *injection* page. The script is automatically retrieved from (C) and executed, forcing an `<iframe>` to be added and rendered. The source of the frame (D) is on the site `smalltableschears[dot]biz`, from which the content is immediately fetched and included in the page. The newly loaded javascript served by (D) then checks for the presence of vulnerable versions of several browser plugins. It quickly matches a version of the installed Adobe Flash Player to a known vulnerability and dynamically adds another `<iframe>` to the page, which pulls a malicious Flash exploit file from (E) on the same `smalltableschears[dot]biz` site (the *exploit* page). The Flash exploit succeeds and the shellcode fetches a malware binary (labeled as ZeroAccess by some AVs) from (F) on the same domain `smalltableschears[dot]biz` (the *download* page).

4.7 “Origin” of Malware Download Paths

Figure 7 shows a breakdown of the drive-by and social engineering “origins” behind the malware downloads. For drive-by downloads, 64% of the download paths started with a search. We noticed that the search query keywords were typically very “normal” (e.g., searching for a new car, social events, or simple tools, as shown in the example in Section 4.6.2), but unfortunately the search results linked to hacked websites that acted as the “entry point” to exploit distribution sites and malware downloads.

For social engineering downloads, about 60% of the web paths started with a search. Search engine queries that eventually led to social engineering attacks tended to be related to less legitimate content. For example, the search queries were often related to free streaming links, pirated movies, or pirated versions of popular expensive software. In these cases, the search results contained links offering content relevant to the search, but the related search result pages would also encourage the user to install malicious software disguised as some required application (e.g., a video codec or a software key generator).

The second most common origin is direct links, whereby a user arrives to a webpage directly (e.g., by clicking on a link within a spam email), rather than through a link from another site. Most of these direct links point to a benign website that is either hacked or displays malicious ads.

Facebook and Twitter represent a relatively infrequent origin for malware downloads (7% and 3% of the cases, respectively). While both Facebook and Twitter usually rely on encrypted (HTTPS) communications, we were able to determine if a download path originated from their sites by noticing that Facebook makes sure that all external requests carry a generic `www.facebook.com` referrer [14]. On the other hand, requests initiated by clicking on a link published on twitter carry a referrer containing a `t.co` shortening URL. During our entire deployment, we only observed one case in which a link from Facebook or Twitter led directly to a drive-by exploit kit. In all other cases, the links led first to a legitimate page that was hacked or that displayed a malicious ad.

For the remaining malicious downloads (less than 3%, overall) we were unable to trace them back to their origin (e.g., due to missing traffic).

During our deployment, we also found that malicious ads are responsible for a significant fraction of the malware downloads in our dataset. Specifically, malicious ads were included in the web path of about 25% of drive-by and 40% of social engineering malware downloads. The malicious ads we observed were typically displayed on relatively unpopular websites. We observed only one example of a malicious ad served on a website with a US Alexa ranking within the top 500.

5 Discussion and System Limitations

Our system only collected data during off-peak hours because it was sharing hardware resources with a production network monitoring system whose functionality could not be disrupted. Thus, our data is just a sample of the malicious downloads that occurred during this period. Also, due to the significant efforts required to analyze complex malware download traces, our evaluation ground truth is limited to a representative sample of the malicious downloads that occurred in the monitored network. However, based on our extensive manual analysis, we believe the samples to be very diverse because of the various exploit kits, exploits, social engineering tricks and malware observed, and therefore representative of the overall set of malware downloads that occurred during our deployment.

One may think that attackers could avoid detection by simply distributing malicious files over encrypted web traffic, using HTTPS. However, it is worth noting that in sensitive networks (e.g., enterprise and government networks) it is now common practice to deploy SSL Man-In-The-Middle (MITM) proxies, which allow for inspecting and recording the content both HTTP and HTTPS traffic (perhaps excluding the traffic towards some whitelisted sides, such as banking applications, etc.). WebWitness could simply work alongside such SSL MITM proxies.

Because the detection of malicious executable files is outside the scope of this paper, we have relied on a "detection oracle" to extract malicious download traces from the network traffic. For the sake of this study, we have chosen to rely on multiple AV scanners. It is well known, though, that AV scanners suffer from false positive and negatives. In addition, the labels assigned by the AV are often not completely meaningful. However, we should consider that using multiple AV scanners reduces the false negatives, and the set of filtering heuristics we discussed in Section 2.2 can mitigate the false positives. In addition, we used re-scanning over a period of a month for each of the downloaded executable files we collected, to further improve our ground truth. Finally, we used the AV labels to filter out adware downloads, because we are mainly interested in the potentially most damaging malware infections. We empirically found that the AV labels usually do a decent job at separating the broad adware and malware classes. Also, we manually reviewed all samples of malware downloads in our dataset, to further mitigate possible mislabeling problems.

Attackers with knowledge of our system may try to evade it by using a purposely crafted attack in attempt to alter some of the features we use in Section 3 to perform path traceback, categorization and for node labeling. Most likely, the attacker will have as a primary goal the evasion of our traceback algorithm. This, for example, could be done by forcing a "disconnect" between the final malware download node and its true predecessors. Such an attack theoretically may be possible, especially in case of drive-by attacks. In such events the browser is compromised and is (in theory) under the full control of the attacker. Now, if the malware download node is isolated in the reconstructed download path, the cause classifier may label the download event a malware update, thus preventing any further processing of the download path (i.e., any attempt to identify the exploit and injection domains).

However, we should also notice that most drive-by downloads are based on what we refer as "commonly exploitable" content in Section 3.3 (e.g., .jar, .swf, or .pdf files that carry an exploit). For such type of drive-by download attacks, the "commonly exploitable" content feature should connect the exploit and the download, if they occur in a small time window. If needed, the time window could be extended by requiring the domain severing the content to be young by checking its "age" before making a connection. Since the exploit must occur before the attacker has control of the browser, it is more difficult to evade.

6 Related Work

Client honeypots actively visit webpages and detect drive-by downloads though observing changes to the system [1, 21, 22, 28, 29] or by analyzing responses for malicious content [3, 10, 24]. These systems tend to have a low false positive rate, but only find malicious websites by visiting them with exploitable browser configurations; also, they have limited range in the quantity of pages they can crawl because they are much slower than static crawlers. Often candidate URLs are selected by filtering content from static crawlers [28, 29, 35], using heuristics to visit parts of the web that are likely more malicious [8] or using search engines to identify webpages that contain content similar to known malicious ones [12].

A number of techniques have been developed to detect drive-by downloads through examining content [10, 11, 15, 32, 34]. Signature based intrusion detection systems, such as Snort [34], passively search network traffic content for patterns of known attacks. Both static [11] and dynamic [10] analysis of JavaScript has been used to detect attacks. The disadvantages of using content is that it is complex and under the control of the attacker. Polymorphic malware and code obfuscation results in missed attacks for signature and static analysis systems, and dynamic analysis can be detected by malware and subverted by altering its execution path [15].

Other systems focus on the redirection chain that leads to drive-by downloads. Stringhini et al [36] create redirection graphs by aggregating redirection chains that end at the same webpage. Features from the redirection graph and visiting users are then used to classify the webpage as malicious or benign. Mekky et al [20] build browsing activity trees using the referrer and redirection headers as well as URLs embedded in the content. Features related to the redirection chain for each tree are extracted and used to classify the activity as malicious or benign. Li et al [17] apply page rank from the dark and bright side of the web to a partially labeled set of redirection chains to separate benign and malicious web paths. They find the majority of malicious paths are directed through traffic distribution systems. Using features from the redirection chain, Surf [18] detects malicious websites found in search engine results due to search poisoning and WarningBird [16] identifies malicious webpages posted on Twitter. These systems focus on the redirection chain and features extracted from it to classify a web activity as benign or malicious. Whereas, WebWitness provides context to malicious downloads by reconstructing the full download path (not just the redirection chain), classifying the cause of the download (drive-by, social, update) and identifying the roles of the domains involved in the attack.

Static blacklists [2] of domains/URLs and domain reputation systems [4, 5] identify malicious websites to prevent users from visiting them. Many of the domains on static blacklists are exploit and download domains that change frequently rendering them less effective. On the other hand, reputation systems only provide a malicious score for a domain and do not indicate their role or give context to an attack. By analyzing the structure of a malicious download, WebWitness can identify the type of attack and the domain roles; providing the highest value domains for blocking and reputation training data.

Recently researchers have proposed executable reputation systems [13,30,38] due the limitations of signature AV [27]. Instead of using content features from the executable content, they focus on properties of the malware distribution infrastructure. These systems can be very effective at identifying malicious downloads. However, they do not provide any context such as how and why the user came to download a malicious executable. Providing download context is the goal of WebWitness not malicious executable detection. We see these systems as complementary to WebWitness and as good candidates to replace our current oracle (signature AV) for malicious executable detection.

Web traffic reconstruction has been studied for example in [9, 25, 39]. WebPatrol [9] uses a client honeypot and a modified web proxy to collect and replay web-based malware scenarios. Unlike WebPatrol, WebWitness is not limited to drive-by downloads invoked through client honeypots and can provide context to drive-by and social engineering attacks on real users observed on live networks. ReSurf [39] uses the referrer header to build graphs of related HTTP transactions to reconstruct web-surfing activities. As discussed in this paper and evaluated in [25], this approach is very limited especially in reconstructing the entire download path of a malicious executable. Lastly, ClickMiner [25] reconstructs user-browser interactions by replaying recorded network traffic through an instrumented browser. Its focus is on the user's behavior that led to a webpage; whereas, WebWitness identifies the cause and structure of an attack that led to a malicious download.

7 Conclusion

We proposed a novel incident investigation system, named WebWitness. Our system targets two main goals: 1) automatically *trace back and label* the chain of events (e.g., visited web pages) preceding malware downloads, to highlight how users reach attack pages on the web; and 2) leverage these automatically labeled in-the-wild malware download paths to better understand current attack trends, and to *develop more effective defenses*.

We deployed WebWitness on a large academic network for a period of 10 months, where we collected and categorized thousands of *live* malware download paths. An analysis of this labeled data allowed us to design a new defense against drive-by downloads that rely on injecting malicious content into (hacked) legitimate web pages. For example, we show that on average by using the results of WebWitness we can decrease the infection rate of drive-by downloads based on malicious content injection by almost *6 times*, compared to existing URL blacklisting approaches.

Acknowledgments

This material is based in part upon work supported by the National Science Foundation (NSF) under grant No. CNS-1149051 and US Department of Commerce (Commerce) under grant no. 2106DEK. 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 NSF and Commerce.

References

[1] Capture-hpc client honeypot. `https://projects.honeynet.org/capture-hpc`.

[2] Google safe browsing api. `https://developers.google.com/safe-browsing/`.

[3] Honeyc. `https://projects.honeynet.org/honeyc`.

[4] Antonakakis, M., Perdisci, R., Dagon, D., Lee, W., and Feamster, N. Building a dynamic reputation system for DNS. In *the Proceedings of 19th USENIX Security Symposium (USENIX Security '10)* (2010).

[5] Antonakakis, M., Perdisci, R., Lee, W., Dagon, D., and Vasiloglou, N. Detecting Malware Domains at the Upper DNS Hierarchy. In *the Proceedings of 20th USENIX Security Symposium (USENIX Security '11)* (2011).

[6] Bakhshi, T., Papadaki, M., and Furnell, S. A practical assessment of social engineering vulnerabilities. In *2nd International Symposium on Human Aspects of Information Security & Assurance (HAISA 2008)* (2008), pp. 12–23.

[7] Breiman, L. Random forests. *Mach. Learn. 45*, 1 (Oct. 2001).

[8] Canali, D., Cova, M., Vigna, G., and Kruegel, C. Prophiler: A fast filter for the large-scale detection of malicious web pages. In *Proceedings of the 20th International Conference on World Wide Web* (New York, NY, USA, 2011), WWW '11, ACM.

[9] Chen, K. Z., Gu, G., Zhuge, J., Nazario, J., and Han, X. Webpatrol: Automated collection and replay of web-based malware scenarios. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2011), ASIACCS '11, ACM.

[10] Cova, M., Kruegel, C., and Vigna, G. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th International Conference on World Wide Web* (New York, NY, USA, 2010), WWW '10, ACM.

[11] Curtsinger, C., Livshits, B., Zorn, B., and Seifert, C. Zozzle: Fast and precise in-browser javascript malware detection. In *Proceedings of the 20th USENIX Conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association.

[12] INVERNIZZI, L., BENVENUTI, S., COVA, M., COMPARETTI, P. M., KRUEGEL, C., AND VIGNA, G. Evilseed: A guided approach to finding malicious web pages. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (2012), SP '12, IEEE Computer Society.

[13] INVERNIZZI, L., LEE, S.-J., MISKOVIC, S., MELLIA, M., TORRES, R., KRUEGEL, C., SAHA, S., AND VIGNA, G. Nazca: Detecting malware distribution in large-scale networks.

[14] JONES, M. Protecting privacy with referrers, 2010. `https://www.facebook.com/notes/facebook-engineering/protecting-privacy-with-referrers/392382738919`.

[15] KAPRAVELOS, A., SHOSHITAISHVILI, Y., COVA, M., KRUEGEL, C., AND VIGNA, G. Revolver: An automated approach to the detection of evasiveweb-based malware. In *Proceedings of the 22Nd USENIX Conference on Security* (Berkeley, CA, USA, 2013), SEC'13, USENIX Association.

[16] LEE, S., AND KIM, J. Warningbird: A near real-time detection system for suspicious urls in twitter stream. *IEEE Trans. Dependable Secur. Comput. 10*, 3 (May 2013).

[17] LI, Z., ALRWAIS, S., XIE, Y., YU, F., AND WANG, X. Finding the linchpins of the dark web: A study on topologically dedicated hosts on malicious web infrastructures. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (2013), SP '13.

[18] LU, L., PERDISCI, R., AND LEE, W. Surf: Detecting and measuring search poisoning. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2011), CCS '11, ACM.

[19] LU, L., YEGNESWARAN, V., PORRAS, P., AND LEE, W. Blade: An attack-agnostic approach for preventing drive-by malware infections. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2010), CCS '10, ACM.

[20] MEKKY, H., TORRES, R., ZHANG, Z.-L., SAHA, S., AND NUCCI, A. Detecting malicious http redirections using trees of user browsing activity. In *INFOCOM, 2014 Proceedings IEEE* (2014).

[21] MIN WANG, Y., BECK, D., JIANG, X., ROUSSEV, R., VERBOWSKI, C., CHEN, S., AND KING, S. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *In NDSS* (2006).

[22] MOSHCHUK, E., BRAGIN, T., GRIBBLE, S. D., AND LEVY, H. M. A crawler-based study of spyware on the web.

[23] NAPPA, A., RAFIQUE, M. Z., AND CABALLERO, J. Driving in the cloud: An analysis of drive-by download operations and abuse reporting. In *Proceedings of the 10th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (Berlin, Heidelberg, 2013), DIMVA'13, Springer-Verlag.

[24] NAZARIO, J. Phoneyc: A virtual client honeypot. In *Proceedings of the 2Nd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More* (Berkeley, CA, USA, 2009), LEET'09, USENIX Association.

[25] NEASBITT, C., PERDISCI, R., LI, K., AND NELMS, T. Clickminer: Towards forensic reconstruction of user-browser interactions from network traces. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer &#38; Communications Security* (New York, NY, USA, 2014), CCS '14, ACM.

[26] NELMS, T., PERDISCI, R., AND AHAMAD, M. Execscent: Mining for new c&#38;c domains in live networks with adaptive control protocol templates. In *Proceedings of the 22Nd USENIX Conference on Security* (Berkeley, CA, USA, 2013), SEC'13, USENIX Association.

[27] PERDISCI, R., LANZI, A., AND LEE, W. Classification of packed executables for accurate computer virus detection. *Pattern Recogn. Lett.*.

[28] PROVOS, N., MAVROMMATIS, P., RAJAB, M. A., AND MONROSE, F. All your iframes point to us. In *Proceedings of the 17th Conference on Security Symposium* (Berkeley, CA, USA, 2008), SS'08, USENIX Association.

[29] PROVOS, N., MCNAMEE, D., MAVROMMATIS, P., WANG, K., AND MODADUGU, N. The ghost in the browser analysis of web-based malware. In *Proceedings of the First Conference on First Workshop on Hot Topics in Understanding Botnets* (2007), HotBots'07.

[30] RAJAB, M. A., BALLARD, L., LUTZ, N., MAVROMMATIS, P., AND PROVOS, N. Camp: Content-agnostic malware protection. In *Proceedings of Annual Network and Distributed System Security Symposium, NDSS (February 2013)* (2013), Citeseer.

[31] RAJAB, M. A., BALLARD, L., MAVROMMATIS, P., PROVOS, N., AND ZHAO, X. The nocebo effect on the web: An analysis of fake anti-virus distribution. In *3rd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More* (Berkeley, CA, USA, 2010), LEET'10, USENIX Association.

[32] RATANAWORABHAN, P., LIVSHITS, B., AND ZORN, B. Nozzle: A defense against heap-spraying code injection attacks. In *Proceedings of the 18th Conference on USENIX Security Symposium* (Berkeley, CA, USA, 2009), SSYM'09, USENIX Association.

[33] RIECK, K., KRUEGER, T., AND DEWALD, A. Cujo: Efficient detection and prevention of drive-by-download attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference* (New York, NY, USA, 2010), ACSAC '10, ACM.

[34] ROESCH, M. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX Conference on System Administration* (1999).

[35] STOKES, J. W., ANDERSEN, R., SEIFERT, C., AND CHELLAPILLA, K. Webcop: Locating neighborhoods of malware on the web. In *Proceedings of the 3rd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More* (2010), LEET'10, USENIX Association.

[36] STRINGHINI, G., KRUEGEL, C., AND VIGNA, G. Shady paths: Leveraging surfing crowds to detect malicious web pages. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security* (2013), CCS '13, ACM.

[37] TOWNSEND, K. R&d: The art of social engineering. *Infosecurity 7*, 4 (2010), 32–35.

[38] VADREVU, P., RAHBARINIA, B., PERDISCI, R., LI, K., AND ANTONAKAKIS, M. Measuring and detecting malware downloads in live network traffic. In *ESORICS*. 2013.

[39] XIE, G., ILIOFOTOU, M., KARAGIANNIS, T., FALOUTSOS, M., AND JIN, Y. Resurf: Reconstructing web-surfing activity from network traffic. In *IFIP Networking Conference, 2013* (May 2013), pp. 1–9.

[40] ZHANG, J., SEIFERT, C., STOKES, J. W., AND LEE, W. Arrow: Generating signatures to detect drive-by downloads. In *Proceedings of the 20th International Conference on World Wide Web* (New York, NY, USA, 2011), WWW '11, ACM.

Vulnerability Disclosure in the Age of Social Media: Exploiting Twitter for Predicting Real-World Exploits

Carl Sabottke Octavian Suciu Tudor Dumitraș
University of Maryland

Abstract

In recent years, the number of software vulnerabilities discovered has grown significantly. This creates a need for prioritizing the response to new disclosures by assessing which vulnerabilities are likely to be exploited and by quickly ruling out the vulnerabilities that are not actually exploited in the real world. We conduct a quantitative and qualitative exploration of the vulnerability-related information disseminated on Twitter. We then describe the design of a Twitter-based exploit detector, and we introduce a threat model specific to our problem. In addition to response prioritization, our detection techniques have applications in risk modeling for cyber-insurance and they highlight the value of information provided by the victims of attacks.

1 Introduction

The number of software vulnerabilities discovered has grown significantly in recent years. For example, 2014 marked the first appearance of a 5 digit CVE, as the CVE database [46], which assigns unique identifiers to vulnerabilities, has adopted a new format that no longer caps the number of CVE IDs at 10,000 per year. Additionally, many vulnerabilities are made public through a *coordinated disclosure process* [18], which specifies a period when information about the vulnerability is kept confidential to allow vendors to create a patch. However, this process results in multi-vendor disclosure schedules that sometimes align, causing a flood of disclosures. For example, 254 vulnerabilities were disclosed on 14 October 2014 across a wide range of vendors including Microsoft, Adobe, and Oracle [16].

To cope with the growing rate of vulnerability discovery, the security community must prioritize the effort to respond to new disclosures by assessing the risk that the vulnerabilities will be exploited. The existing scoring systems that are recommended for this purpose, such as FIRST's Common Vulnerability Scoring System (CVSS) [54], Microsoft's exploitability index [21] and Adobe's priority ratings [19], err on the side of caution by marking many vulnerabilities as likely to be exploited [24]. The situation in the real world is more nuanced. While the disclosure process often produces *proof of concept* exploits, which are publicly available, recent empirical studies reported that only a small fraction of vulnerabilities are exploited *in the real world*, and this fraction has decreased over time [22,47]. At the same time, some vulnerabilities attract significant attention and are quickly exploited; for example, exploits for the Heartbleed bug in OpenSSL were detected 21 hours after the vulnerability's public disclosure [41]. To provide an adequate response on such a short time frame, the security community must quickly determine which vulnerabilities are exploited in the real world, while minimizing false positive detections.

The security vendors, system administrators, and hackers, who discuss vulnerabilities on social media sites like Twitter, constitute rich sources of information, as the participants in coordinated disclosures discuss technical details about exploits and the victims of attacks share their experiences. This paper explores the opportunities for *early exploit detection* using information available on Twitter. We characterize the exploit-related discourse on Twitter, the information posted before vulnerability disclosures, and the users who post this information. We also reexamine a prior experiment on predicting the development of proof-of-concept exploits [36] and find a considerable performance gap. This illuminates the threat landscape evolution over the past decade and the current challenges for early exploit detection.

Building on these insights, we describe techniques for detecting exploits that are active in the real world. Our techniques utilize *supervised machine learning* and ground truth about exploits from ExploitDB [3], OSVDB [9], Microsoft security advisories [21] and the descriptions of Symantec's anti-virus and intrusion-protection signatures [23]. We collect an unsampled corpus of tweets that contain the keyword "CVE," posted between February 2014 and January 2015, and we extract features for training and testing a support vector machine (SVM) classifier. We evaluate the false positive and false negative rates and we assess the detection lead time compared to existing data sets. Because Twitter is an open and free service, we introduce a *threat model*, considering realistic adversaries that can poison both the training and the testing data sets but that may be resource-bound, and we conduct simulations to evaluate the resilience of our detector to such attacks. Finally, we discuss the implications of our results for building security systems without secrets, the applications of early exploit detection and the value of sharing information about successful attacks.

In summary, we make three contributions:

- We characterize the landscape of threats related to information leaks about vulnerabilities before their public disclosure, and we identify features that can be extracted automatically from the Twitter discourse to detect exploits.
- To our knowledge, we describe the first technique for early detection of real-world exploits using social media.
- We introduce a threat model specific to our problem and we evaluate the robustness of our detector to adversarial interference.

Roadmap. In Sections 2 and 3 we formulate the problem of exploit detection and we describe the design of our detector, respectively. Section 4 provides an empirical analysis of the exploit-related information disseminated on Twitter, Section 5 presents our detection results, and Section 6 evaluates attacks against our exploit detectors. Section 7 reviews the related work, and Section 8 discusses the implications of our results.

2 The problem of exploit detection

We consider a *vulnerability* to be a software bug that has security implications and that has been assigned a unique identifier in the CVE database [46]. An *exploit* is a piece of code that can be used by an attacker to subvert the functionality of the vulnerable software. While many researchers have investigated the techniques for creating exploits, the utilization patterns of these exploits provide another interesting dimension to their security implications. We consider *real-world exploits* to be the exploits that are being used in real attacks against hosts and networks worldwide. In contrast, *proof-of-concept (PoC) exploits* are often developed as part of the vulnerability disclosure process and are included in penetration testing suites. We further distinguish between *public PoC exploits*, for which the exploit code is publicly available, and *private PoC exploits*, for which we can find reliable information that the exploit was developed, but it was not released to the public. A PoC exploit may also be a real-world exploit if it is used in attacks.

The existence of a real-world or PoC exploit gives urgency to fixing the corresponding vulnerability, and this knowledge can be utilized for prioritizing remediation actions. We investigate the opportunities for *early detection* of such exploits by using information that is available publicly, but is not included in existing vulnerability databases such as the National Vulnerability Database (NVD) [7] or the Open Sourced Vulnerability Database (OSVDB) [9]. Specifically, we analyze the Twitter stream, which exemplifies the information available from social media feeds. On Twitter, a community of hackers, security vendors and system administrators discuss security vulnerabilities. In some cases, the victims of attacks report new vulnerability exploits. In other cases, information leaks from the *coordinated disclosure* process [18] through which the security community prepares the response to the impending public disclosure of a vulnerability.

The vulnerability-related discourse on Twitter is influenced by trend-setting vulnerabilities, such as Heartbleed (CVE-2014-0160), Shellshock (CVE-2014-6271, CVE-2014-7169, and CVE-2014-6277) or Drupalgeddon (CVE-2014-3704) [41]. Such vulnerabilities are mentioned by many users who otherwise do not provide actionable information on exploits, which introduces a significant amount of noise in the information retrieved from the Twitter stream. Additionally, adversaries may inject fake information into the Twitter stream, in an attempt to poison our detector. Our *goals* in this paper are (i) to identify the good sources of information about exploits and (ii) to assess the opportunities for early detection of exploits in the presence of benign and adversarial noise. Specifically, we investigate techniques for minimizing *false-positive detections*—vulnerabilities that are not actually exploited—which is critical for prioritizing response actions.

Non-goals. We do not consider the detection of *zero-day attacks* [32], which exploit vulnerabilities before their public disclosure; instead, we focus on detecting the use of exploits against known vulnerabilities. Because our aim is to assess the value of publicly available information for exploit detection, we do not evaluate the benefits of incorporating commercial or private data feeds. The design of a complete system for early exploit detection, which likely requires mechanisms beyond the realm of Twitter analytics (e.g., for managing the reputation of data sources to prevent poisoning attacks), is also out of scope for this paper.

2.1 Challenges

To put our contributions in context, we review the three primary challenges for predicting exploits in the absence of adversarial interference: class imbalance, data scarcity, and ground truth biases.

Class imbalance. We aim to train a classifier that produces binary predictions: each vulnerability is classified as either exploited or not exploited. If there are significantly more vulnerabilities in one class than in the other class, this biases the output of supervised machine learning algorithms. Prior research on predicting the existence of proof-of-concept exploits suggests that this bias is not large, as over half of the vulnerabilities disclosed before 2007 had such exploits [36]. However, few vulnerabilities are exploited in the real world and the exploitation ratios tend to decrease over time [47]. In consequence, our data set exhibits a severe class imbalance: we were able to find evidence of real-world exploitation for only 1.3% of vulnerabilities disclosed during our observation period. This class imbalance represents a significant challenge for simultaneously reducing the false positive and false negative detections.

Data scarcity. Prior research efforts on Twitter analytics have been able to extract information from millions of tweets, by focusing on popular topics like movies [27], flu outbreaks [20, 26], or large-scale threats like spam [56]. In contrast, only a small subset of Twitter users discuss vulnerability exploits (approximately 32,000 users), and they do not always mention the CVE numbers in their tweets, which prevents us from identifying the vulnerability discussed. In consequence, 90% of the CVE numbers disclosed during our observation period appear in fewer than 50 tweets. Worse, when considering the known real-world exploits, close to half have fewer than 50 associated tweets. This data scarcity compounds the challenge of class imbalance for reducing false positives and false negatives.

Quality of ground truth. Prior work on Twitter analytics focused on predicting quantities for which good predictors are already available (modulo a time lag): the Hollywood Stock Exchange for movie box-office revenues [27], CDC reports for flu trends [45] and Twitter's internal detectors for highjacked accounts, which trigger account suspensions [56]. These predictors can be used as ground truth for training high-performance classifiers. In contrast, there is no comprehensive data set of vulnerabilities that are exploited in the real world. We employ as ground truth the set of vulnerabilities mentioned in the descriptions of Symantec's anti-virus and intrusion-protection signatures, which is, reportedly, the best available indicator for the exploits included in exploit kits [23, 47]. However, this dataset has coverage biases, since Symantec does not cover all platforms and products uniformly. For example, since Symantec does not provide a security product for Linux, Linux kernel vulnerabilities are less likely to appear in our ground truth dataset than exploits targeting software that runs on the Windows platform.

2.2 Threat model

Research in adversarial machine learning [28, 29], distinguishes between exploratory attacks, which poison the testing data, and causative attacks, which poison both the testing and the training data sets. Because Twitter is an open and free service, causative adversaries are a realistic threat to a system that accepts inputs from all Twitter users. We assume that these adversaries cannot prevent the victims of attacks from tweeting about their observations, but they can inject additional tweets in order to compromise the performance of our classifier. To test the ramifications of these causative attacks, we develop a threat model with three types of adversaries.

Blabbering adversary. Our weakest adversary is not aware of the statistical properties of the training features or labels. This adversary simply sends tweets with random CVEs and random security-related keywords.

Word copycat adversary. A stronger adversary is aware of the features we use for training and has access to our ground truth (which comes from public sources). This adversary uses fraudulent accounts to manipulate the word features and total tweet counts in the training data. However, this adversary is resource constrained and cannot manipulate any user statistics which would require either more expensive or time intensive account acquisition and setup (e.g., creation date, verification, follower and friend counts). The copycat adversary crafts tweets by randomly selecting pairs of non-exploited and exploited vulnerabilities and then sending tweets, so that the word feature distributions between these two classes become nearly identical.

Full copycat adversary. Our strongest adversary has full knowledge of our feature set. Additionally, this adversary has sufficient time and economic resources to purchase or create Twitter accounts with arbitrary user statistics, with the exception of verification and the account creation date. Therefore, the full copycat adversary can use a set of fraudulent Twitter accounts to fully manipulate almost all word and user-based features, which creates scenarios where relatively benign CVEs and real-world exploit CVEs appear to have nearly identical Twitter traffic at an abstracted statistical level.

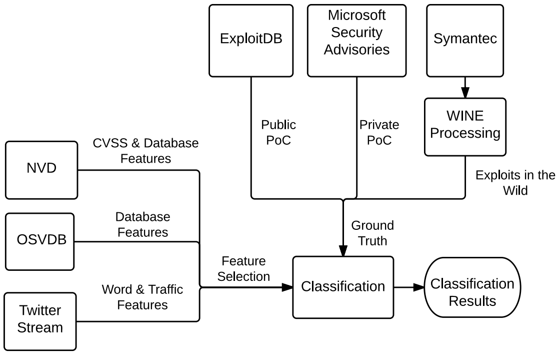


Figure 1: Overview of the system architecture.

3 A Twitter-based exploit detector

We present the design of a Twitter-based exploit detector, using supervised machine learning techniques. Our detector extracts vulnerability-related information from the Twitter stream, and augments it with additional sources of data about vulnerabilities and exploits.

3.1 Data collection

Figure 1 illustrates the architecture of our exploit detector. Twitter is an online social networking service that enables users to send and read short 140-character messages called "tweets", which then become publicly available. For collecting tweets mentioning vulnerabilities, the system monitors occurrences of the "CVE" keyword using Twitter's Streaming API [15]. The policy of the Streaming API implies that a client receives all the tweets matching a keyword as long as the result does not exceed 1% of the entire Twitter hose, when the tweets become samples of the entire matching volume. Because the CVE tweeting volume is not high enough to reach 1% of the hose (as the API signals rate limiting), we conclude that our collection contains all references to CVEs, except during the periods of downtime for our infrastructure.

We collect data over a period of one year, from February 2014 to January 2015. Out of the 1.1 billion tweets collected during this period, 287,717 contain explicit references to CVE IDs. We identify 7,560 distinct CVEs. After filtering out the vulnerabilities disclosed before the start of our observation period, for which we have missed many tweets, we are left with 5,865 CVEs.

To obtain context about the vulnerabilities discussed on Twitter, we query the National Vulnerability Database (NVD) [7] for the CVSS scores, the products affected and additional references about these vulnerabilities. Additionally, we crawl the Open Sourced Vulnerability Database (OSVDB) [9] for a few additional attributes, including the disclosure dates and categories of the vulnerabilities in our study.[1] Our data collection infrastructure consists of Python scripts, and the data is stored using Hadoop Distributed File System. From the raw data collected, we extract multiple features using Apache PIG and Spark, which run on top of a local Hadoop cluster.

Ground truth. We use three sources of ground truth. We identify the set of vulnerabilities *exploited in the real world* by extracting the CVE IDs mentioned in the descriptions of Symantec's anti-virus (AV) signatures [12] and intrusion-protection (IPS) signatures [13]. Prior work has suggested that this approach produces the best available indicator for the vulnerabilities targeted in exploits kits available on the black market [23, 47]. Considering only the vulnerabilities included in our study, this data set contains 77 vulnerabilities targeting products from 31 different vendors. We extract the creation date from the descriptions of AV signatures to estimate the date when the exploits were discovered. Unfortunately, the IPS signatures do not provide this information, so we query Symantec's Worldwide Intelligence Network Environment (WINE) [40] for the dates when these signatures were triggered in the wild. For each real-world exploit, we use the earliest date across these data sources as an estimate for the date when the exploit became known to the security community.

However, as mentioned in Section 2.1, this ground truth does not cover all platforms and products uniformly. Nevertheless, we expect that some software vendors, which have well established procedures for coordinated disclosure, systematically notify security companies of impending vulnerability disclosures to allow them to release detection signatures on the date of disclosure. For example, the members of Microsoft's MAPP program [5] receive vulnerability information in advance of the monthly publication of security advisories. This practice provides defense-in-depth, as system administrators can react to vulnerability disclosures either by deploying the software patches or by updating their AV or IPS signatures. To identify which products are well covered in this data set, we group the exploits by the vendor of the affected product. Out of the 77 real-world exploits, 41 (53%) target products from Microsoft and Adobe, while no other vendor accounts for more than 3% of exploits. This suggests that our ground truth provides the best coverage for vulnerabilities in Microsoft and Adobe products.

We identify the set of vulnerabilities with *public proof-of-concept exploits* by querying ExploitDB [3], a collaborative project that collects vulnerability exploits. We

[1] In the past, OSVDB was called the Open Source Vulnerability Database and released full dumps of their database. Since 2012, OSVDB no longer provides public dumps and actively blocks attempts to crawl the website for most of the information in the database.

identify exploits for 387 vulnerabilities disclosed during our observation period. We use the date when the exploits were added to ExploitDB as an indicator for when the vulnerabilities were exploited.

We also identify the set of vulnerabilities in Microsoft's products for which *private proof-of-concept* exploits have been developed by using the Exploitability Index [21] included in Microsoft security advisories. This index ranges from 0 to 3: 0 for vulnerabilities that are known to be exploited in the real world at the time of release for a security bulletin,[2] and 1 for vulnerabilities that allowed the development of exploits with consistent behavior. Vulnerabilities with scores of 2 and 3 are considered less likely and unlikely to be exploited, respectively. We therefore consider that the vulnerabilities with an exploitability index of 0 or 1 have an private PoC exploit, and we identify 218 such vulnerabilities. 22 of these 218 vulnerabilities are considered real-world exploits in our Symantec ground truth.

3.2 Vulnerability categories

To quantify how these vulnerabilities and exploits are discussed on Twitter, we group them into 7 categories, based on their utility for an attacker: Code Execution, Information Disclosure, Denial of Service, Protection Bypass, Script Injection, Session Hijacking and Spoofing. Although heterogeneous and unstructured, the summary field from NVD entries provides sufficient information for assigning a category to most of the vulnerabilities in the study, using regular expressions comprised of domain vocabulary.

Table 2 and Section 4 show how these categories intersect with POC and real-world exploits. Since vulnerabilities may belong to several categories (a code execution exploit could also be used in a denial of service), the regular expressions are applied in order. If a match is found for one category, the subsequent categories would not be matched.

Aditionally, the Unknown category contains vulnerabilities not matched by the regular expressions and those whose summaries explicitly state that the consequences are unknown or unspecified.

3.3 Classifier feature selection

The features considered in this study can be classified in 4 categories: Twitter Text, Twitter Statistics, CVSS Information and Database Information.

Keyword	MI Wild	MI PoC	Keyword	MI Wild	MI PoC
advisory	0.0007	0.0005	ok	0.0015	0.0002
beware	0.0007	0.0005	mcafee	0.0005	0.0002
sample	0.0007	0.0005	windows	0.0012	0.0011
exploit	0.0026	0.0016	w	0.0004	0.0002
go	0.0007	0.0005	microsoft	0.0007	0.0005
xp	0.0007	0.0005	info	0.0007	X
ie	0.0015	0.0005	rce	0.0007	X
poc	0.0004	0.0006	patch	0.0007	X
web	0.0015	0.0005	piyolog	0.0007	X
java	0.0007	0.0005	tested	0.0007	X
working	0.0007	0.0005	and	X	0.0005
fix	0.0012	0.0002	rt	X	0.0005
bug	0.0007	0.0005	eset	X	0.0005
blog	0.0007	0.0005	for	X	0.0005
pc	0.0007	0.0005	redhat	X	0.0002
reading	0.0007	0.0005	kali	X	0.0005
iis	0.0007	0.0005	0day	X	0.0009
ssl	0.0005	0.0003	vs	X	0.0005
post	0.0007	0.0005	linux	X	0.0009
day	0.0015	0.0005	new	X	0.0002
bash	0.0015	0.0009			

Table 1: Mutual information provided by the reduced set of keywords with respect to both sources of ground truth data. The "X" marks in the table indicate that the respective words were excluded from the final feature set due to MI below 0.0001 nats.

For the Twitter features, we started with a set of 1000 keywords and 12 additional features based on the distribution of tweets for the CVEs, e.g. the total number of tweets related to the CVE, the average age of the accounts posting about the vulnerability and the number of retweets associated to the vulnerability. For each of these initial features, we compute the mutual information (MI) of the set of feature values X and the class labels $Y \in \{\text{exploited}, \text{not exploited}\}$:

$$MI(Y,X) = \sum_{x \in X} \sum_{y \in Y} p(x,y) \ln \left(\frac{p(x,y)}{p(x)p(y)} \right)$$

Mutual information, expressed in nats, compares the frequencies of values from the joint distribution $p(x,y)$ (i.e. values from X and Y that occur together) with the product of the frequencies from the two distributions $p(x)$ and $p(y)$. MI measures how much knowing X reduces uncertainty about Y, and can single out useful features suggesting that the vulnerability is exploited as well as features suggesting it is not. We prune the initial feature set by excluding all features with mutual information below 0.0001 nats. For numerical features, we estimate probability distributions using a resolution of 50 bins per feature. After this feature selection process, we are left with 38 word features for real-world exploits.

Here, rather than use a wrapper method for feature selection, we use this mutual information-based filter method in order to facilitate the combination of automatic feature selection with intuition-driven manual pruning. For example, keywords that correspond to trend-setting vulnerabilities from 2014, like Heartbleed and Shellshock, exhibit a higher mutual information than many other potential keywords despite their relation to only a small subset of vulnerabilities. Yet, such highly

[2] We do not use this score as an indicator for the existence of real-world exploits because the 0 rating is available only since August 2014, toward the end of our observation period.

specific keywords are undesirable for classification due to their transitory utility. Therefore, in order to reduce susceptibility to concept drift, we manually prune these word features for our classifiers to generate a final set of 31 out of 38 word features (listed in Table 1 with additional keyword intuition described in Section 4.1).

In order to improve performance and increase classifier robustness to potential Twitter-based adversaries acting through account hijacking and Sybil attacks, we also derive features from NVD and OSVDB. We consider all 7 CVSS score components, as well as features that proved useful for predicting proof-of-concept exploits in prior work [36], such as the number of unique references, the presence of the token BUGTRAQ in the NVD references, the vulnerability category from OSVDB and our own vulnerability categories (see Section 3.2). This gives us 17 additional features. The inclusion of these non-Twitter features is useful for boosting the classifier's resilience to adversarial noise. Figure 2 illustrates the most useful features for detecting real-world or PoC exploits, along with the corresponding mutual information.

3.4 Classifier training and evaluation

We train linear support vector machine (SVM) classifiers [35, 38, 39, 43] in a feature space with 67 dimensions that results from our feature selection step (Section 3.3). SVMs seek to determine the maximum margin hyperplane to separate the classes of exploited and non-exploited vulnerabilities. When a hyperplane cannot perfectly separate the positive and negative class samples based on the feature vectors used in training, the basic SVM cost function is modified to include a regularization penalty, C, and non-negative slack variables, ξ_i. By varying C, we explore the trade-off between false negatives and false positives in our classifiers.

We train SVM classifiers using multiple rounds of stratified random sampling. We perform sampling because of the large imbalance in the class sizes between vulnerabilities exploited and vulnerabilities not exploited. Typically, our classifier training consists of 10 random training shuffles where 50% of the available data is used for training and the remaining 50% is used for testing. We use the `scikit-learn` Python package [49] to train our classifiers.

Category	# CVEs	Real-World	PoC	Both
	All Data / Good Coverage			
Code Execution	1249/322	66/39	192/14	28/8
Info Disclosure	1918/59	4/0	69/5	4/0
Denial Of Service	657/17	0/0	16/1	0/0
Protection Bypass	204/34	0/0	3/0	0/0
Script Injection	683/14	0/0	40/0	0/0
Session Hijacking	167/1	0/0	25/0	0/0
Spoofing	55/4	0/0	0/0	0/0
Unknown	981/51	7/0	42/6	5/0
Total	5914/502	77/39	387/26	37/8

Table 2: CVEs Categories and exploits summary. The first sub-column represents the whole dataset, while the second sub-column is restricted to Adobe and Microsoft vulnerabilities, for which our ground truth of real-world exploits provides good coverage.

An important caveat, though, is that our one year of data limits our ability to evaluate concept drift. In most cases, our cross-validation data is temporally intermixed with the training data, since restricting sets of training and testing CVEs to temporally adjacent blocks confounds performance losses due to concept drift with performance losses due to small sample sizes. Furthermore, performance differences between the vulnerability database features of our classifiers and those explored in [36] emphasize the benefit of periodically repeating previous experiments from the security literature in order to properly assess whether the results are subject to long-term concept drift.

Performance metrics. When evaluating our classifiers, we rely on two standard performance metrics: *precision* and *recall*.[3] Recall is equivalent to the true positive rate: Recall $= \frac{TP}{TP+FN}$, where TP is the number of true positive classifications and FN is the number of false negatives. The denominator is the total number of positive samples in the testing data. Precision is defined as: Precision $= \frac{TP}{TP+FP}$ where FP is the total number of false positives identified by the classifier. When optimizing classifier performance based on these criteria, the relative importance of these quantities is dependent on the intended applications of the classifier. If avoiding false negatives is priority, then recall must be high. However, if avoiding false positives is more critical, then precision is the more important metric. Because we envision utilizing our classifier as a tool for prioritizing the response to vulnerability disclosures, we focus on improving the precision rather than the recall.

4 Exploit-related information on Twitter

Table 2 breaks down the vulnerabilities in our study according to the categories described in Section 3.2. 1249 vulnerabilities allowing code execution were disclosed during our observation period. 66 have real-world exploits, and 192 have public proof-of-concept exploits; the intersection of these two sets includes 28 exploits. If we consider only Microsoft and Adobe vulnerabilities, for which we expect that our ground truth has good coverage (see Section 3.1), the table shows that 322 code-

[3] We choose precision and recall because Receiver Operating Characteristic (ROC) curves can present an overly optimistic view of a classifier's performance when dealing with skewed data sets [**?**].

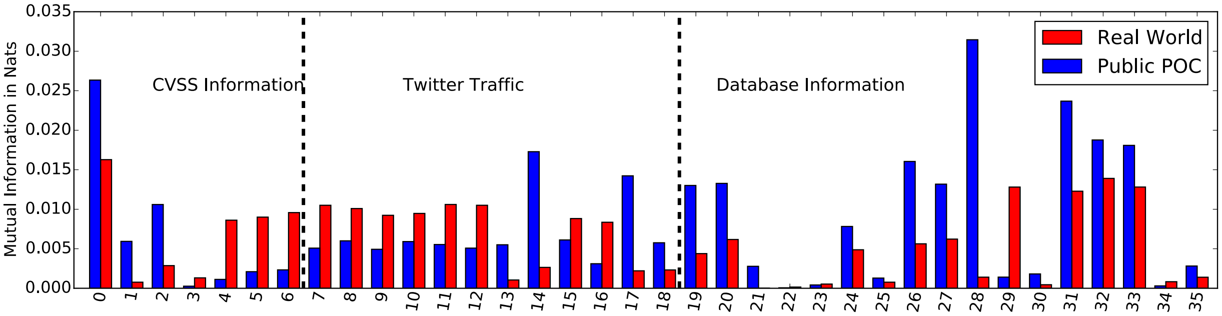


Figure 2: Mutual information between real world and public proof of concept exploits for CVSS, Twitter user statistics, NVD, and OSVDB features. **CVSS Information** - 0: CVSS Score, 1: Access Complexity, 2: Access Vector, 3: Authentication, 4: Availability Impact, 5: Confidentiality Impact, 6: Integrity Impact. **Twitter Traffic** - 7: Number of tweets, 8/9: # users with minimum T followers/friends, 10/11: # retweets/replies, 12: # tweets favorited, 13/14/15: Avg # hashtags/URLs/user mentions per tweet, 16: # verified accounts, 17: Avg age of accounts, 18: Avg # of tweets per account. **Database Information** - 19: # references in NVD, 20: # sources in NVD, 21/22: BUGTRAQ/SECUNIA in NVD sources, 23: allow in NVD summay, 24: NVD last modified date - NVD published date, 25: NVD last modified date - OSVDB disclosed date, 26: Number of tokens in OSVDB title, 27: Current date - NVD last modified date, 28: OSVDB in NVD sources, 29: code in NVD summay, 30: # OSVDB entries, 31: OSVDB Category, 32: Regex Category, 33: First vendor in NVD, 34: # vendors in NVD, 35: # affected products in NVD.

execution vulnerabilities were disclosed, 39 have real-world exploits, 14 have public PoC exploits and 8 have both real-world and public PoC exploits.

Information disclosure is the largest category of vulnerabilities from NVD and it has a large number of PoC exploits, but we find few of these vulnerabilities in our ground truth of real-world exploits (one exception is Heartbleed). Instead, most of the real-world exploits focus on *code execution* vulnerabilities. However, many proof-of-concept exploits for such vulnerabilities do not seem to be utilized in real-world attacks. To understand the factors that drive the differences between real-world and proof-of-concept exploits, we examine the CVSS base metrics, which describe the characteristics of each vulnerability. This analysis reveals that most of the real-world exploits allow *remote* code execution, while some PoC exploits require local host or local network access. Moreover, while some PoC vulnerabilities require authentication before a successful exploit, real-world exploits focus on vulnerabilities that do not require *bypassing authentication* mechanisms. In fact, this is the only type of exploit we found in the segment of our ground truth that has good coverage, suggesting that remote code-execution exploits with no authentication required are strongly favored by real-world attackers. Our ground truth does not provide good coverage of web exploits, which explains the lack of Script Injection, Session Highjacking and Spoofing exploits from our real-world data set.

Surprisingly, we find that, among the remote execution vulnerabilities for which our ground truth provides good coverage, there are *more real-world exploits than public PoC exploits*. This could be explained by the increasing prevalence of obfuscated disclosures, as reflected in NVD vulnerability summaries that mention the possibility of exploitation "via unspecified vectors" (for example, CVE-2014-8439). Such disclosures make it more difficult to create PoC exploits, as the technical information required is not readily available, but they may not thwart determined attackers who have gained experience in hacking the product in question.

4.1 Exploit-related discourse on Twitter

The Twitter discourse is dominated by a few vulnerabilities. Heartbleed (CVE-2014-0160) received the highest attention, with more than 25,000 tweets (8,000 posted in the first day after disclosure). 24 vulnerabilities received more than 1,000 tweets. 16 of these vulnerabilities were exploited: 11 in the real-world, 12 in public proofs of concept and 8 in private proofs of concept. The median number of tweets across all the vulnerabilities in our data set is 14.

The terms that Twitter users employ when discussing exploits also provide interesting insights. Surprisingly, the distribution of the keyword "0day" exhibits a high mutual information with public proof-of-concept exploits, but not with real-world exploits. This could be explained by confusion over the definition of the term *zero-day vulnerability*: many Twitter users understand this to mean simply a new vulnerability, rather than a vulnerability that was exploited in real-world attacks before its public disclosure [32]. Conversely, the distribution of the keyword "patch" has high mutual information only with the real-world exploits, because a common reason for posting tweets about vulnerabilities is to alert other users and point them to advisories for updating vulnerable software versions after exploits are detected in the wild. Certain words like “exploit” or “advisory” are useful for detecting both real-world and PoC exploits.

4.2 Information posted before disclosure

For the vulnerabilities in our data set, Figure 3 compares the dates of the earliest tweets mentioning the corresponding CVE numbers with the public disclosure dates for these vulnerabilities. Using the disclosure date recorded in OSVDB, we identify 47 vulnerabilities that were mentioned in the Twitter stream before their public disclosure. We investigate these cases manually to determine the sources of this information. 11 of these cases represent misspelled CVE IDs (e.g. users mentioning 6172 but talking about 6271 – Shellshock), and we are unable to determine the root cause for 5 additional cases owing to the lack of sufficient information. The remaining cases can be classified into 3 general categories of information leaks:

Disagreements about the planned disclosure date. The vendor of the vulnerable software sometimes posts links to a security bulletin ahead of the public disclosure date. These cases are typically benign, as the security advisories provide instructions for patching the vulnerability. A more dangerous situation occurs when the party who discovers the vulnerability and the vendor disagree about the disclosure schedule, resulting in the publication of vulnerability details a few days before a patch is made available [6, 14, 16, 17]. We have found 13 cases of disagreements about the disclosure date.

Coordination of the response to vulnerabilities discovered in open-source software. The developers of open-source software sometimes coordinate their response to new vulnerabilities through social media, e.g. mailing lists, blogs and Twitter. An example for this behavior is a tweet about a `wget` patch for CVE-2014-4877 posted by the patch developer, followed by retweets and advice to update the binaries. If the public discussion starts before a patch is completed, then this is potentially dangerous. However, in the 5 such cases we identified, the patching recommendations were first posted on Twitter and followed by an increased retweet volume.

Leaks from the coordinated disclosure process. In some cases, the participants in the coordinated disclosure process leak information before disclosure. For example, security researchers may tweet about having confirmed that a vulnerability is exploitable, along with the software affected. This is the most dangerous situation, as attackers may then contact the researcher with offers to purchase the exploit, before the vendor is able to release a patch. We have identified 13 such cases.

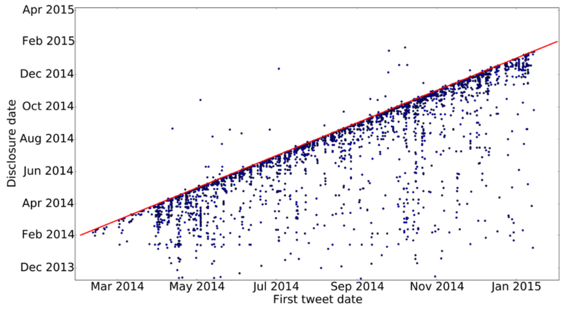


Figure 3: Comparison of the disclosure dates with the dates when the first tweets are posted for all the vulnerabilities in our dataset. Plotted in red is the identity line where the two dates coincide.

4.3 Users with information-rich tweets

The tweets we have collected were posted by approximately 32,000 unique users, but the messages posted by these users are not equally informative. Therefore, we quantify utility on a per user basis by computing the ratio of CVE tweets related to real-world exploits as well as the fraction of unique real-world exploits that a given user tweets about. We rank user utility based on the harmonic mean of these two quantities. This ranking penalizes users that tweet about many CVEs indiscriminately (e.g. a security news bot) as well as the thousands of users that only tweet about the most popular vulnerabilities (e.g. Shellshock and Heartbleed). We create a whitelist with the top 20% most informative users, and we use this whitelist in our experiments in the following sections as a means of isolating our classifier from potential adversarial attacks. Top ranked whitelist users include computer repair servicemen posting about the latest viruses discovered in their shops and security researchers and enthusiasts sharing information about the latest blog and news postings related to vulnerabilities and exploits.

Figure 4 provides an example of how the information about vulnerabilities and exploits propagates on Twitter amongst all users. The “Futex” bug, which enables unauthorized root access on Linux systems, was disclosed on June 6 as CVE-2014-3153. After identifying users who posted messages that had retweets counting for at least 1% of the total tweet counts for this vulnerability and applying a thresholding based on the number of retweets,

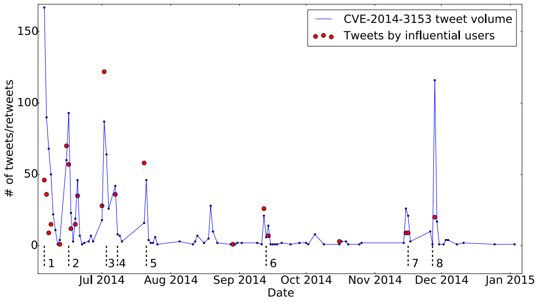


Figure 4: Tweet volume for CVE-2014-3153 and tweets from the most influential users. These influential tweets shape the volume of Twitter posts about the vulnerability. The 8 marks represent important events in the vulnerability's lifecycle: 1 - disclosure, 2 - Android exploit called Towelroot is reported and exploitation attempts are detected in the wild, 3,4,5 - new technical details emerge, including the Towelroot code, 6 - new mobile phones continue to be vulnerable to this exploit, 7 - advisory about the vulnerability is posted, 8 - exploit is included in ExploitDB.

we identify 20 influential tweets that shaped the volume of Twitter messages. These tweets correspond to 8 important milestones in the vulnerability's lifecycle, as marked in the figure.

While CVE-2014-3153 is known to be exploited in the wild, it is not included in our ground truth for real-world exploits, which does not cover the Linux platform. This example illustrates that monitoring a subset of users can yield most of the vulnerability- and exploit-related information available on Twitter. However, over reliance on a small number of user accounts, even with manual analysis, can increase susceptibility to data manipulation via adversarial account hijacking.

5 Detection of proof-of-concept and real-world exploits

To provide a baseline for our ability to classify exploits, we first examine the performance of a classifier that uses only the CVSS score, which is currently recommended as the reference assessment method for software security [50]. We use the total CVSS score and the exploitability subscore as a means of establishing baseline classifier performances. The exploitability subscore is calculated as a combination of the CVSS access vector, access complexity, and authentication components. Both the total score and exploitability subscore range from 0-10. By varying a threshold across the full range of values for each score, we can generate putative labels where vulnerabilities with scores above the threshold are marked as "real-world exploits" and vulnerabilities below the threshold are labeled as "not exploited". Unsurprisingly, since CVSS is designed as a high recall system which errs on the side of caution for vulnerability severity, the maximum possible precision for this baseline classifier is less than 9%. Figure 5 shows the recall and precision values for both total CVSS score thresholds and CVSS exploitability subscore thresholds.

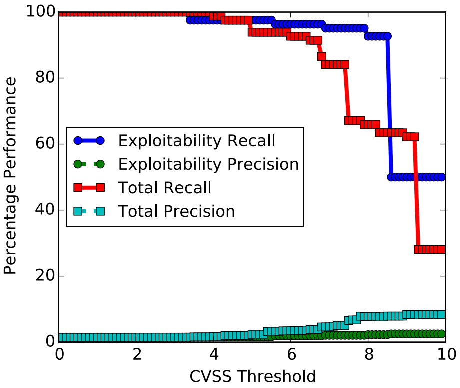
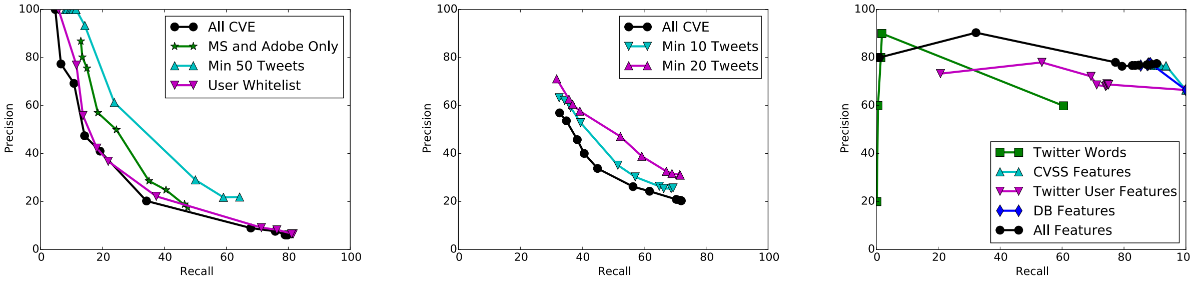


Figure 5: Precision and recall for classifying real world exploits with CVSS score thresholds.

Thus, this high recall, low precision vulnerability score is not useful by itself for real-world exploit identification, and boosting precision is a key area for improvement.

Classifiers for real-world exploits. Classifiers for real-world exploits have to deal with a severe class imbalance: we have found evidence of real-world exploitation for only 1.3% of the vulnerabilities disclosed during our observation period. To improve the classification precision, we train linear SVM classifiers on a combination of CVSS metadata features, features extracted from security-related tweets, and features extracted from NVD and OSVDB (see Figure 2). We tune these classifiers by varying the regularization parameter *C*, and we illustrate the precision and recall achieved. Values shown are for cross-validation testing averaged across 10 stratified random shuffles. Figure 6a shows the average cross-validated precision and recall that are simultaneously acheivable with our Twitter-enhanced feature set. These classifiers can achieve higher precision than a baseline classifier that uses only the CVSS score, but there is still a tradeoff between precision and recall. We can tune the classifier with regularization to decrease the number of false positives (increasing precision), but this comes at the cost of a larger number of false negatives (decreasing recall).



(a) Real-world exploits.

(b) Public proof-of-concept exploits.

(c) Private proof-of-concept exploits for vulnerabilities in Microsoft products.

Figure 6: Precision and recall of our classifiers.

This result partially reflects an additional challenge for our classifier: the fact that our ground truth is imperfect, as Symantec does not have products for all the platforms that are targeted in attacks. If our Twitter-based classifier predicts that a vulnerability is exploited and the exploit exists, but is absent from the Symantec signatures, we will count this instance as a false positive, penalizing the reported precision. To assess the magnitude of this problem, we restrict the training and evaluation of the classifier to the 477 vulnerabilities in Microsoft and Adobe products, which are likely to have a good coverage in our ground truth for real-world exploits (see Section 3.1). 41 of these 477 vulnerabilities are identified as real-world exploits in our ground truth. For comparison, we include the performance of this classifier in Figure 6a. Improving the quality of the ground truth allows us to bolster the values of precision and recall which are simultaneously achievable, while still enabling classification precision an order of magnitude larger than a baseline CVSS score-based classifier. Additionally, while restricting the training of our classifier to a whitelist made up of the top 20% most informative Twitter users (as described in Section 4.3) does not enhance classifier performance, it does allow us to achieve a precision comparable to the previous experiments (Figure 6a). This is helpful for preventing an adversary from poisoning our classifier, as discussed in Section 6.

These results illustrate the current potential and limitations for predicting real-world exploits using publicly-available information. Further improvements in the classification performance may be achieved through a broader effort for sharing information about exploits active in the wild, in order to assemble a high-coverage ground truth for training of classifiers.

Classifiers for proof-of-concept exploits. We explore two classification problems: predicting *public proof-of-concept exploits*, for which the exploit code is publicly available, and predicting *private proof-of-concept exploits*, for which we can find reliable information that the exploit was developed, but it was not released to the public. We consider these problems separately, as they have different security implications and the Twitter users are likely to discuss them in distinct ways.

First, we train a classifier to predict the availability of exploits in ExploitDB [3], the largest archive of *public* exploits. This is similar to the experiment reported by Bozorgi et al. in [36], except that our feature set is slightly different—in particular, we extract word features from Twitter messages, rather than from the textual descriptions of the vulnerabilities. However, we include the most useful features for predicting proof-of-concept exploits, as reported in [36]. Additionally, Bozorgi et al. determined the availability of proof-of-concept exploits using information from OSVDB [9], which is typically populated using references to ExploitDB but may also include vulnerabilities for which the exploit is rumored or private (approximately 17% of their exploit data set). After training a classifier with the information extracted about the vulnerabilities disclosed between 1991–2007, they achieved a precision of 87.5%.

Surprisingly, we are not able to reproduce their performance results, as seen in Figure 6b, when analyzing the vulnerabilities that appear in ExploitDB in 2014. The figure also illustrates the performance of a classifier trained with exclusion threshold for CVEs that lack sufficient quantities of tweets. These volume thresholds improve performance, but not dramatically.[4] In part, this is due to our smaller data set compared to [36], made up of vulnerabilities disclosed during one year rather than a 16-year period. Moreover, our ground truth for public proof-of-concept exploits also exhibits a high class im-

[4] In this case, we do not restrict the exploits to specific vendors, as ExploitDB will incorporate any exploit submitted.

balance: only 6.2% of the vulnerabilities disclosed during our observation period have exploits available in ExploitDB. This is in stark contrast to the prior work, where more than half of vulnerabilities had proof of concept exploits.

This result provides an interesting insight into the evolution of the threat landscape: today, proof-of-concept exploits are less centralized than in 2007, and ExploitDB does not have total coverage of public proof-of-concept exploits. We have found several tweets with links to PoC exploits, published on blogs or mailing lists, that were missing from ExploitDB (we further analyze these instances in Section 4). This suggests that information about public exploits is increasingly dispersed among social media sources, rather than being included in a centralized database like ExploitDB,[5] which represents a hurdle for prioritizing the response to vulnerability disclosures.

To address this concern, we explore the potential for predicting the existence of *private* proof-of-concept exploits by considering only the vulnerabilities disclosed in Microsoft products and by using Microsoft's Exploitability Index to derive our ground truth. Figure 6c illustrates the performance of a classifier trained with a conservative ground truth in which we treat vulnerabilities with scores of 1 or less as exploits. This classifier achieves precision and recall higher than 80%, even when only relying on database feature subsets. Unlike in our prior experiments, for the Microsoft Exploitability Index the classes are more balanced: 67% of the vulnerabilities (218 out of 327) are labeled as having a private proof-of-concept exploit. However, only 8% of the Microsoft vulnerabilities in our dataset are contained within our real-world exploit ground truth. Thus, by using a conservative ground truth that labels many vulnerabilities as exploits we can achieve high precision and recall, but this classifier performance does not readily translate to real-world exploit prediction.

Contribution of various feature groups to the classifier performance. To understand how our features contribute to the performance of our classifiers, in Figure 7 we compare the precision and recall of our real-world exploit classifier when using different subgroups of features. In particular, incorporating Twitter data into the classifiers allows for improving the precision beyond the levels of precision achievable with data that is currently available publicly in vulnerability databases. Both user features and word features generated based on tweets are capable of bolstering classifier precision in comparison to CVSS and features extracted from NVD and OSVDB. Consequently, the analysis of social media streams like Twitter is useful for boosting identification of exploits active in the real-world.

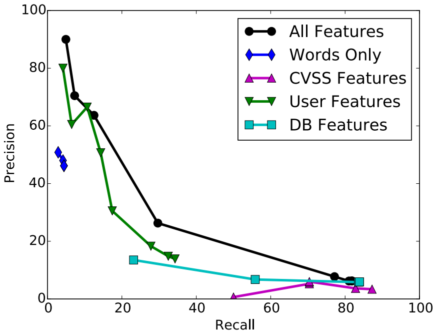


Figure 7: Precision and recall for classification of real world exploits with different feature subsets. Twitter features allow higher-precision classification of real-world exploits.

5.1 Early detection of exploits

In this section we ask the question: *How soon can we detect exploits active in the real world by monitoring the Twitter stream?* Without rapid detecton capabilities that leverage the real-time data availability inherent to social media platforms, a Twiter-based vulnerability classifier has little practical value. While the first tweets about a vulnerability precede the creation of IPS or AV signatures by a median time of 10 days, these first tweets are typically not informative enough to determine that the vulnerability is likely exploited. We therefore simulate a scenario where our classifier for real-world exploits is used in an online manner, in order to identify the dates when the output of the classifier changes from "not exploited" to "exploited" for each vulnerability in our ground truth.

We draw 10 stratified random samples, each with 50% coverage of "exploited" and "not exploited" vulnerabilities, and we train a separate linear SVM classifier with each one of these samples ($C = 0.0003$). We start testing each of our 10 classifiers with a feature set that does not include features extracted from tweets, to simulate the activation of the online classifier. We then continue to test the ensemble of classifiers incrementally, by adding one tweet at a time to the testing set. We update the aggregated prediction using a moving average with a window of 1000 tweets. Figure 8a highlights the tradeoff between precision and early detection for a range of aggregated SVM prediction thresholds. Notably, though, large precision sacrifices do not necessarily lead to large

[5] Indeed, OSVDB no longer seems to provide the exploitation availability flag.

gains for the speed of detection. Therefore, we choose an aggregated prediction threshold of 0.95, which achieves 45% precision and a median lead prediction time of two days before the first Symantec AV or WINE IPS signature dates. Figure 8b shows how our classifier detection lags behind first tweet appearances. The solid blue line shows the cumulative distribution function (CDF) for the number of days difference between the first tweet appearance for a CVE and the first Symantec AV or IPS attack signature. The green dashed line shows the CDF for the day difference between 45% precision classification and the signature creation. Negative day differences indicate that Twitter events occur before the creation of the attack signature. In approximately 20% of cases, early detection is impossible because the first tweet for a CVE occurs after an attack signature has been created. For the remaining vulnerabilities, Twitter data provides valuable insights into the likelihood of exploitation.

Figure 8c illustrates early detection for the case of Heartbleed (CVE-2014-1060). The green line indicates the first appearance of the vulnerability in ExploitDB, and the red line indicates the date when a Symantec attack signature was published. The dashed black line represents the earliest time when our online classifier is able to detect this vulnerability as a real-world exploit at 45% precision. Our Twitter-based classifier provides an "exploited" output 3 hours after the first tweet appears related to this CVE on April 7, 2014. Heartbleed exploit traffic was detected 21 hours after the vulnerability's public disclosure [41]. Heartbleed appeared in ExploitDB on the day after disclosure (April 8, 2014), and Symantec published the creation of an attack signature on April 9, 2014. Additionally, by accepting lower levels of precision, our Twitter-based classifiers can achieve even faster exploit detection. For example, with classifier precision set to approximately 25%, Heartbleed can be detected as an exploit within 10 minutes of its first appearance on Twitter.

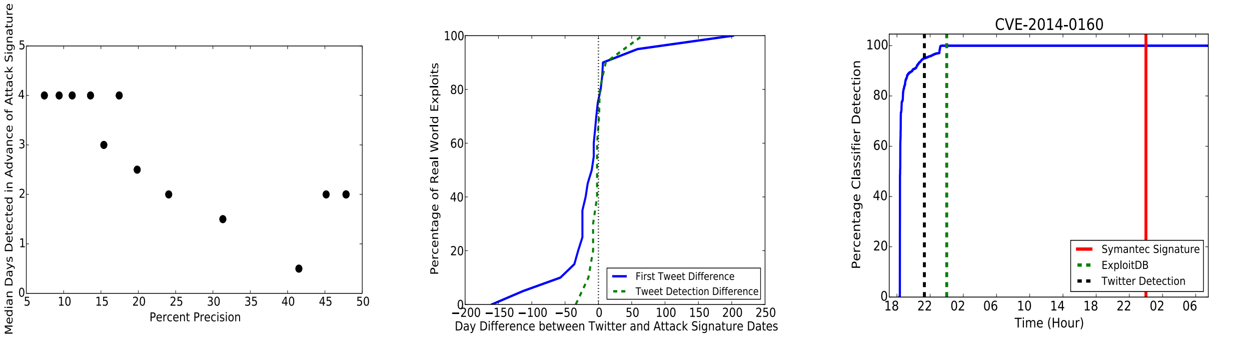
6 Attacks against the exploit detectors

The public nature of Twitter data necessitates considering classification problems not only in an ideal environment, but also in an environment where adversaries may seek to poison the classifiers. In causative adversarial machine learning (AML) attacks, the adversaries make efforts to have a direct influence by corrupting and altering the training data [28, 29, 31].

With Twitter data, learning the statistics of the training data is as simple as collecting tweets with either the REST or Streaming APIs. Features that are likely to be used in classification can then be extracted and evaluated using criteria such as correlation, entropy, or mutual information, when ground truth data is publicly available. In this regard, the most conservative assumption for security is that an adversary has complete knowledge of a Twitter-based classifier's training data as well as knowledge of the feature set.

When we assume an adversary works to create both false negatives and false positives (an availability AML security violation), practical implementation of a basic causative AML attack on Twitter data is relatively straightforward. Because of the popularity of spam on Twitter, websites such as `buyaccs.com` cheaply sell large volumes of fraudulent Twitter accounts. For example, on February 16, 2015 on buyaccs.com, the baseline price for 1000 AOL email-based Twitter accounts was $17 with approximately 15,000 accounts available for purchase. This makes it relatively cheap (less than $300 as a base cost) to conduct an attack in which a large number of users tweet fraudulent messages containing CVEs and keywords, which are likely to be used in a Twitter-based classifier as features. Such an attacker has two main limitations. The first limitation is that, while the attacker can add an extremely large number of tweets to the Twitter stream via a large number of different accounts, the attacker has no straightforward mechanism for removing legitimate, potentially informative tweets from the dataset. The second limitation is that additional costs must be incurred if an attacker's fraudulent accounts are to avoid identification. Cheap Twitter accounts purchased in bulk have low friend counts and low follower counts. A user profile-based preprocessing stage of analysis could easily eliminate such accounts from the dataset if an adversary attempts to attack a Twitter classification scheme in such a rudimentary manner. Therefore, to help make fraudulent accounts seem more legitimate and less readily detectable, an adversary must also establish realistic user statistics for these accounts.

Here, we analyze the robustness of our Twitter-based classifiers when facing three distinct causative attack strategies. The first attack strategy is to launch a causative attack without any knowledge of the training data or ground truth. This *blabbering adversary* essentially amounts to injecting noise into the system. The second attack strategy corresponds to the *word-copycat adversary*, who does not create a sophisticated network between the fraudulent accounts and only manipulates word features and the total tweet count for each CVE. This attacker sends malicious tweets, so that the word statistics for tweets about non-exploited and exploited CVEs appear identical at a user-naive level of abstraction. The third, most powerful adversary we consider is the *full-copycat adversary*. This adversary manipulates the user statistics (friend, follower, and status counts) of a large number of fraudulent accounts as well as the text content of these CVE-related tweets to launch a more sophisticated Sybil attack. The only user statis-



(a) Tradeoff between classification speed and precision.

(b) CDFs for day differences between first tweet for a CVE, first classification, and attack signature creation date.

(c) Heartbleed detection timeline between April 7th and April 10th, 2014.

Figure 8: Early detection of real-world vulnerability exploits.

tics which we assume this full copycat adversary cannot arbitrarily manipulate are account verification status and account creation date, since modifying these features would require account hijacking. The goal of this adversary is for non-exploited and exploited CVEs to appear as statistically identical as possible on Twitter at a user-anonymized level of abstraction.

For all strategies, we assume that the attacker has purchased a large number of fraudulent accounts. Fewer than 1,000 out of the more than 32,000 Twitter users in our CVE tweet dataset send more than 20 CVE-related tweets in a year, and only 75 accounts send 200 or more CVE-related tweets. Therefore, if an attacker wishes to avoid tweet volume-based blacklisting, then each account cannot send a high number of CVE-related tweets. Consequently, if the attacker sets a volume threshold of 20-50 CVE tweets per account, then 15,000 purchased accounts would enable the attacker to send 300,000-750,000 adversarial tweets.

The blabbering adversary, even when sending 1 million fraudulent tweets, is not able to force the precision of our exploit detector below 50%. This suggests that Twitter-based classifiers can be relatively robust to this type of random noise-based attack (black circles in Fig. 9). When dealing with the word-copycat adversary (green squares in Fig. 9), performance asymptotically degrades to 30% precision. The full-copycat adversary can cause the precision to drop to approximately 20% by sending over 300,000 tweets from fraudulent accounts. The full-copycat adversary represents a practical upper bound for the precision loss that a realistic attacker can inflict on our system. Here, performance remains above baseline levels even for our strongest Sybil attacker due to our use of non-Twitter features to increase classifier robustness. Nevertheless, in order to recover performance, implementing a Twitter-based vulnerability classifier in a realistic setting is likely to require curation of whitelists and blacklists for informative and adversarial users. As shown in figure 6a, restricting the classifier to only consider the top 20% of users with the most relevant tweets about real-world exploits causes no performance degradation and fortifies the classifier against low tier adversarial threats.

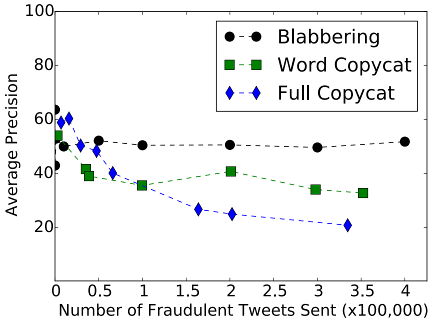


Figure 9: Average linear SVM precision when training and testing data are poisoned by the three types of adversaries from our threat model.

7 Related work

Previous work by Allodi et al. has highlighted multiple deficiencies in CVSS version 2 as a metric for predicting whether or not a vulnerability will be exploited in the wild [24], specifically because predicting the small fraction of vulnerabilities exploited in the wild is not one of the design goals of CVSS. By analyzing vulnerabilities in exploit kits, work by Allodi et al. has also established that Symantec threat signatures are an effective source of information for determining which vulnerabilities are exploited in the real world, even if the coverage is not complete for all systems [23].

The closest work to our own is Bozorgi et al., who applied linear support vector machines to vulnerability and exploit metadata in order predict the development of proof-of-concept exploits [36]. However, the existence of a POC exploit does not necessarily mean that an exploit will be leveraged for attacks in the wild, and real world attacks occur in only of a small fraction of the cases for which a vulnerability has a proof of concept exploit. [25,47]. In contrast, we aim to detect exploits that are active in the real world. Hence, our analysis expands on this prior work [36] by focusing classifier training on real world exploit data rather than POC exploit data and by targeting social media data as a key source of features for distinguishing real world exploits from vulnerabilities that are not exploited in the real world.

In prior analysis of Twitter data, success has been found in a wide variety of applications including earthquake detection [53], epidemiology [26, 37], and the stock market [34, 58]. In the security domain, much attention has been focused on detecting Twitter spam accounts [57] and detecting malicious uses of Twitter aimed at gaining political influence [30, 52, 55]. The goals of these works is distinct from our task of predicting whether or not vulnerabilities are exploited in the wild. Nevertheless, a practical implementation of our vulnerability classification methodology would require the detection of fraudulent tweets and spam accounts to prevent poisoning attacks.

8 Discussion

Security in Twitter analytics. Twitter data is publicly available, and new users are free to join and start sending messages. In consequence, we cannot obfuscate or hide the features we use in our machine learning system. Even if we had not disclosed the features we found most useful for our problem, an adversary can collect Twitter data, as well as the data sets we use for ground truth (which are also public), and determine the most information rich features within the training data in the the same way we do. Our exploit detector is an example of a *security system without secrets*, where the integrity of the system does not depend on the secrecy of its design or of the features it uses for learning. Instead, the security properties of our system derive from the fact that the adversary can inject new messages in the Twitter stream, but cannot remove any messages sent by the other users. Our threat model and our experimental results provide practical bounds for the damage the adversary can inflict on such a system. This damage can be reduced further by incorporating techniques for identifying adversarial Twitter accounts, for example by assigning a reputation score to each account [44,48,51].

Applications of early exploit detection. Our results suggest that, the information contained in security-related tweets is an important source for timely security-related information. Twitter-based classifiers can be employed to guide the prioritization of response actions after vulnerability disclosures, especially for organizations with strict policies for testing patches prior to enterprise-wide deployment, which makes patching a resource-intensive effort. Another potential application is modeling the risk associated with vulnerabilities, for example by combining the likelihood of real-world exploitation, produced by our system, with additional metrics for vulnerability assessment, such as the CVSS severity scores or the odds that the vulnerable software is exploitable given its deployment context (e.g. whether it is attached to a publicly-accessible network). Such models are key for the emerging area of cyber-insurance [33], and they would benefit from an evidence-based approach for estimating the likelihood of real-world exploitation.

Implications for information sharing efforts. Our results highlight the current challenges for the early detection of exploits, in particular the fact that the existing sources of information for exploits active in the wild do not cover all the platforms that are targeted by attackers. The discussions on security-related mailing lists, such as Bugtraq [10], Full Disclosure [4] and oss-security [8], focus on disclosing vulnerabilities and publishing exploits, rather than on reporting attacks in the wild. This makes it difficult for security researchers to assemble a high-quality ground truth for training supervised machine learning algorithms. At the same time, we illustrate the potential of this approach. In particular,our whitelist identifies 4,335 users who post information-rich messages about exploits. We also show that the classification performance can be improved significantly by utilizing a ground truth with better coverage. We therefore encourage the victims of attacks to share relevant technical information, perhaps through recent information-sharing platforms such as Facebook's ThreatExchange [42] or the Defense Industrial Base voluntary information sharing program [1].

9 Conclusions

We conduct a quantitative and qualitative exploration of information available on Twitter that provides early warnings for the existence of real-world exploits. Among the products for which we have reliable ground truth, we identify more vulnerabilities that are *exploited in* *the real-world* than vulnerabilities for which *proof-of-concept exploits* are available publicly. We also identify a group of 4,335 users who post information-rich messages about real-world exploits. We review several unique challenges for the exploit detection problem, including the skewed nature of vulnerability datasets, the frequent scarcity of data available at initial disclosure times and the low coverage of real world exploits in the ground truth data sets that are publicly available. We characterize the threat of information leaks from the coordinated disclosure process, and we identify features that are useful for detecting exploits.

Based on these insights, we design and evaluate a detector for real-world exploits utilizing features extracted from Twitter data (e.g., specific words, number of retweets and replies, information about the users posting these messages). Our system has fewer false positives than a CVSS-based detector, boosting the detection precision by *one order of magnitude*, and can detect exploits a median of *2 days ahead of existing data sets*. We also introduce a threat model with three types of adversaries seeking to poison our exploit detector, and, through simulation, we present practical bounds for the damage they can inflict on a Twitter-based exploit detector.

Acknowledgments

We thank Tanvir Arafin for assistance with our early data collection efforts. We also thank the anonymous reviewers, Ben Ransford (our shepherd), Jimmy Lin, Jonathan Katz and Michelle Mazurek for feedback on this research. This research was supported by the Maryland Procurement Office, under contract H98230-14-C-0127 and by the National Science Foundation, which provided cluster resources under grant CNS-1405688 and graduate student support with GRF 2014161339.

References

[1] Cyber security / information assurance (CS/IA) program. `http://dibnet.dod.mil/`.

[2] Drupalgeddon vulnerability - cve-2014-3704. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3704`.

[3] Exploits database by offensive security. `https:http://exploit-db.com/`.

[4] Full disclosure mailing list. `http://seclists.org/fulldisclosure/`.

[5] Microsoft active protections program. `https://technet.microsoft.com/en-us/security/dn467918`.

[6] Microsoft's latest plea for cvd is as much propaganda as sincere. http://blog.osvdb.org/2015/01/12/microsofts-latest-plea-for-vcd-is-as-much-propaganda-as-sincere/.

[7] National vulnerability database (NVD). `https://nvd.nist.gov/`.

[8] Open source security. `http://oss-security.openwall.org/wiki/mailing-lists/oss-security`.

[9] Open sourced vulnerability database (OSVDB). `http://www.osvdb.org/`.

[10] Securityfocus bugtraq. `http://www.securityfocus.com/archive/1`.

[11] Shellshock vulnerability - cve-2014-6271. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271`.

[12] Symantec a-z listing of threats & risks. http://www.symantec.com/security_response/landing/azlisting.jsp.

[13] Symantec attack signatures. `http://www.symantec.com/security_response/attacksignatures/`.

[14] Technet blogs - a call for better coordinated vulnerability disclosure. http://blogs.technet.com/b/msrc/archive/2015/01/11/a-call-for-better-coordinated-vulnerability-disclosure.aspx/.

[15] The twitter streaming api. `https://dev.twitter.com/streaming/overview/`.

[16] Vendors sure like to wave the ”coordination” flag. http://blog.osvdb.org/2015/02/02/vendors-sure-like-to-wave-the-coordination-flag-revisiting-the-perfect-storm/.

[17] Windows elevation of privilege in user profile service - google security research. https://code.google.com/p/google-security-research/issues/detail?id=123.

[18] Guidelines for security vulnerability reporting and response. Tech. rep., Organization for Internet Safety, September 2004. `http://www.symantec.com/security/OIS_Guidelines%20for%20responsible%20disclosure.pdf`.

[19] Adding priority ratings to adobe security bulletins. http://blogs.adobe.com/security/2012/02/when-do-i-need-to-apply-this-update-adding-priority-ratings-to-adobe-security-bulletins-2.html, 2012.

[20] Achrekar, H., Gandhe, A., Lazarus, R., Yu, S.-H., and Liu, B. Predicting flu trends using twitter data. In *Computer Communications Workshops (INFOCOM WKSHPS), 2011 IEEE Conference on* (2011), IEEE, pp. 702–707.

[21] Alberts, B., and Researcher, S. A Bounds Check on the Microsoft Exploitability Index The Value of an Exploitability Index Exploitability. 1–7.

[22] Allodi, L., and Massacci, F. A Preliminary Analysis of Vulnerability Scores for Attacks in Wild. In *CCS BADGERS Workshop* (Raleigh, NC, Oct. 2012).

[23] Allodi, L., and Massacci, F. A preliminary analysis of vulnerability scores for attacks in wild. *Proceedings of the 2012 ACM Workshop on Building analysis datasets and gathering experience returns for security - BADGERS '12* (2012), 17.

[24] Allodi, L., and Massacci, F. Comparing vulnerability severity and exploits using case-control studies. *(Rank B)ACM Transactions on Embedded Computing Systems 9*, 4 (2013).

[25] Allodi, L., Shim, W., and Massacci, F. Quantitative Assessment of Risk Reduction with Cybercrime Black Market Monitoring. *2013 IEEE Security and Privacy Workshops* (2013), 165–172.

[26] Aramaki, E., Maskawa, S., and Morita, M. Twitter Catches The Flu : Detecting Influenza Epidemics using Twitter The University of Tokyo. In *Proceedings of the 2011 Conference on Emperical Methods in Natural Language Processing* (2011), pp. 1568–1576.

[27] Asur, S., and Huberman, B. A. Predicting the future with social media. In *Web Intelligence and Intelligent Agent Technology (WI-IAT), 2010 IEEE/WIC/ACM International Conference on* (2010), vol. 1, IEEE, pp. 492–499.

[28] Barreno, M., Bartlett, P. L., Chi, F. J., Joseph, A. D., Nelson, B., Rubinstein, B. I., Saini, U., and Tygar, J. Open problems in the security of learning. In *Proceedings of the 1st . . .* (2008), p. 19.

[29] Barreno, M., Nelson, B., Joseph, A. D., and Tygar, J. D. The security of machine learning. *Machine Learning 81* (2010), 121–148.

[30] Benevenuto, F., Magno, G., Rodrigues, T., and Almeida, V. Detecting spammers on twitter. *Collaboration, electronic messaging, anti-abuse and spam conference (CEAS) 6* (2010), 12.

[31] Biggio, B., Nelson, B., and Laskov, P. Support Vector Machines Under Adversarial Label Noise. *ACML 20* (2011), 97–112.

[32] Bilge, L., and Dumitraș, T. Before we knew it: An empirical study of zero-day attacks in the real world. In *ACM Conference on Computer and Communications Security* (2012), T. Yu, G. Danezis, and V. D. Gligor, Eds., ACM, pp. 833–844.

[33] Böhme, R., and Schwartz, G. Modeling Cyber-Insurance: Towards a Unifying Framework. In *WEIS* (2010).

[34] Bollen, J., Mao, H., and Zeng, X. Twitter mood predicts the stock market. *Journal of Computational Science 2* (2011), 1–8.

[35] Boser, B. E., Guyon, I. M., and Vapnik, V. N. A Training Algorithm for Optimal Margin Classifiers. In *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory* (1992), pp. 144–152.

[36] Bozorgi, M., Saul, L. K., Savage, S., and Voelker, G. M. Beyond Heuristics : Learning to Classify Vulnerabilities and Predict Exploits. In *KDD '10 Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining* (2010), p. 9.

[37] Broniatowski, D. A., Paul, M. J., and Dredze, M. National and local influenza surveillance through twitter: An analysis of the 2012-2013 influenza epidemic. *PLoS ONE 8* (2013).

[38] Chang, C.-C., and Lin, C.-J. LIBSVM: A Library for Support Vector Machines. *ACM Transactions on Intelligent Systems and Technology 2* (2011), 27:1—-27:27.

[39] Cortes, C., Cortes, C., Vapnik, V., and Vapnik, V. Support Vector Networks. *Machine Learning 20* (1995), 273˜–˜297.

[40] Dumitraș, T., and Shou, D. Toward a Standard Benchmark for Computer Security Research: The {Worldwide Intelligence Network Environment (WINE)}. In *EuroSys BADGERS Workshop* (Salzburg, Austria, Apr. 2011).

[41] Durumeric, Z., Kasten, J., Adrian, D., Halderman, J. A., Bailey, M., Li, F., Weaver, N., Amann, J., Beekman, J., Payer, M., and Paxson, V. The matter of Heartbleed. In *Proceedings of the Internet Measurement Conference* (Vancouver, Canada, Nov. 2014).

[42] Facebook. ThreatExchange. `https://threatexchange.fb.com/`.

[43] Guyon, I., Guyon, I., Boser, B., Boser, B., Vapnik, V., and Vapnik, V. Automatic Capacity Tuning of Very Large VC-Dimension Classifiers. In *Advances in Neural Information Processing Systems* (1993), vol. 5, pp. 147–155.

[44] Horng, D., Chau, P., Nachenberg, C., Wilhelm, J., Wright, A., and Faloutsos, C. Polonium : Tera-Scale Graph Mining and Inference for Malware Detection. *Siam International Conference on Data Mining (Sdm)* (2011), 131–142.

[45] Lazer, D. M., Kennedy, R., King, G., and Vespignani, A. The parable of Google Flu: traps in big data analysis.

[46] Mitre. Common vulnerabilities and exposures (CVE). urlhttp://cve.mitre.org/.

[47] Nayak, K., Marino, D., Efstathopoulos, P., and Dumitraș, T. Some Vulnerabilities Are Different Than Others: Studying Vulnerabilities and Attack Surfaces in the Wild. In *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses* (Gothenburg, Sweeden, Sept. 2014).

[48] Pandit, S., Chau, D., Wang, S., and Faloutsos, C. Netprobe: a fast and scalable system for fraud detection in online auction networks. *Proceedings of the 16th . . . 42* (2007), 210, 201.

[49] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine Learning in {P}ython. *Journal of Machine Learning Research 12* (2011), 2825–2830.

[50] Quinn, S., Scarfone, K., Barrett, M., and Johnson, C. Guide to Adopting and Using the Security Content Automation Protocol {(SCAP)} Version 1.0. NIST Special Publication 800-117, July 2010.

[51] Rajab, M. A., Ballard, L., Lutz, N., Mavrommatis, P., and Provos, N. CAMP: Content-agnostic malware protection. In *Network and Distributed System Security (NDSS) Symposium* (San Diego, CA, Feb 2013).

[52] Ratkiewicz, J., Conover, M. D., Meiss, M., Gonc, B., Flammini, A., and Menczer, F. Detecting and Tracking Political Abuse in Social Media. *Artificial Intelligence* (2011), 297–304.

[53] Sakaki, T., Okazaki, M., and Matsuo, Y. Earthquake shakes Twitter users: real-time event detection by social sensors. In *WWW '10: Proceedings of the 19th international conference on World wide web* (2010), p. 851.

[54] Scarfone, K., and Mell, P. An analysis of CVSS version 2 vulnerability scoring. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement, ESEM 2009* (2009), pp. 516–525.

[55] Thomas, K., Grier, C., and Paxson, V. Adapting Social Spam Infrastructure for Political Censorship. In *LEET* (2011).

[56] Thomas, K., Li, F., Grier, C., and Paxson, V. Consequences of Connectivity: Characterizing Account Hijacking on Twitter. ACM Press, pp. 489–500.

[57] Wang, A. H. Don't follow me: Spam detection in Twitter. *2010 International Conference on Security and Cryptography (SECRYPT)* (2010), 1–10.

[58] Wolfram, M. S. A. Modelling the Stock Market using Twitter. *iccsinformaticsedacuk Master of* (2010), 74.

Needles in a Haystack: Mining Information from Public Dynamic Analysis Sandboxes for Malware Intelligence

Mariano Graziano
Eurecom

Davide Canali
Eurecom

Leyla Bilge
Symantec Research Labs

Andrea Lanzi
Universita' degli Studi di Milano

Davide Balzarotti
Eurecom

Abstract

Malware sandboxes are automated dynamic analysis systems that execute programs in a controlled environment. Within the large volumes of samples submitted every day to these services, some submissions appear to be different from others, and show interesting characteristics. For example, we observed that malware samples involved in famous targeted attacks – like the Regin APT framework or the recently disclosed malwares from the Equation Group – were submitted to our sandbox months or even years before they were detected in the wild. In other cases, the malware developers themselves interact with public sandboxes to test their creations or to develop a new evasion technique. We refer to similar cases as *malware developments*.

In this paper, we propose a novel methodology to automatically identify *malware development* cases from the samples submitted to a malware analysis sandbox. The results of our experiments show that, by combining dynamic and static analysis with features based on the file submission, it is possible to achieve a good accuracy in automatically identifying cases of *malware development*. Our goal is to raise awareness on this problem and on the importance of looking at these samples from an intelligence and threat prevention point of view.

1 Introduction

Malware sandboxes are automated dynamic analysis tools that execute samples in an isolated and instrumented environment. Security researchers use them to quickly collect information about the behavior of suspicious samples, typically in terms of their execution traces and API calls. While customized sandboxes are often installed in the premises of security companies, some sandboxes are available as public online services, as it is the case for Malwr [13], Anubis [10], ThreatExpert [14], VirusTotal [16], and many others [5, 18, 4, 6, 15, 1, 3]

The main advantage of these systems is the fact that the analysis is completely automated and easily parallelizable, thus providing a way to cope with the overwhelming number of new samples that are collected every day. However, due to this extreme parallelization, an incredible amount of reports are generated every day. This makes the task of distinguishing new and important malware from the background noise of polymorphic and uninteresting samples very challenging.

In particular, two important and distinct observations motivate our work. First, it is relatively common that malware samples used to carry out famous targeted attacks were collected by antivirus companies or public sandboxes long before the attacks were publicly discovered [25]. For instance, the binaries responsible for operation Aurora, Red October, Regin, and even some of the new one part of the Equation Group were submitted to the sandbox we used in our experiments several months before the respective attacks appeared in the news [11, 40, 17, 50, 45, 35]. The reasons behind this phenomenon are not always clear. It is possible that the files were automatically collected as part of an automated network or host-based protection system. Or maybe a security analyst noticed something anomalous on a computer and wanted to double-check if a suspicious file exhibited a potentially malicious behavior. It is even possible that the malware developers themselves submitted an early copy of their work to verify whether it triggered any alert on the sandbox system. Whatever the reason, the important point is that no one paid attention to those files until it was too late.

The second observation motivating our study is the constant arm race between the researchers that put continuous effort to randomize their analysis environments, and the criminals that try to fingerprint those systems to avoid being detected. As a consequence of this hidden battle, malware and packers often include evasion techniques for popular sandboxes [19] and updated information about the internal sandbox details are regularly posted on public websites [2]. These examples prove that there must be a constant interaction between malware developers and popular public malware analysis services. This interaction is driven by the need to collect updated information as well as to make sure that new malware creation would go undetected. Even though detecting this interaction might be very difficult, we believe it would provide valuable information for malware triage.

Up to the present, malware analysis services have collected large volumes of data. This data has been used both to enhance analysis techniques [23, 46] and to extrapolate trends and statistics about the evolution of malware families [24]. Unfortunately, to the best of our knowledge, these datasets have never been used to systematically study malware development and support malware intelligence on a large scale. The only public exception is a research recently conducted by looking at VirusTotal to track the activity of specific high-profile hacking groups involved in APT campaigns [52, 27].

In this paper, we approach this objective by applying data-mining and machine learning techniques to study the data collected by Anubis Sandbox [10], a popular malware dynamic analysis service. At the time we performed our analysis, the dataset contained the analysis reports for over 30 millions unique samples. Our main goal is to automatically detect if miscreants submit their samples during the malware development phase and, if this is the case, to acquire more insights about the dynamics of malware development. By analyzing the metadata associated to the sample submissions, it might be possible to determine the software provenance and implement an early-warning system to flag suspicious submission behaviors.

It is important to understand that our objective is not to develop a full-fledged system, but instead to explore a new direction and to show that by combining metadata with static and dynamic features it is possible to successfully detect many examples of malware development submitted to public sandboxes. In fact, our simple prototype was able to automatically identify thousands of development cases, including botnets, keyloggers, backdoors, and over a thousand unique trojan applications.

2 Overview and Terminology

There are several reasons why criminals may want to interact with an online malware sandbox. It could be just for curiosity, in order to better understand the analysis environment and estimate its capabilities. Another reason could be to try to escape from the sandbox isolation to perform some malicious activity, such as scanning a network or attacking another machine. Finally, criminals may also want to submit samples for testing purposes, to make sure that a certain evasion technique works as expected in the sandbox environment, or that a certain malware prototype does not raise any alarm.

In this paper, we focus on the detection of what we call *malware development*. We use the term *"development"* in a broad sense, to include anything that is submitted by the author of the file itself. In many cases the author has access to the source code of the program – either because she wrote it herself or because she acquired it from someone else. However, this is not always the case, e.g., when the author of a sample uses a builder tool to automatically generate a binary according to a number of optional configurations (see Section 6 for a practical example of this scenario). Moreover, to keep things simple, we also use the word *"malware"* as a generic term to model any suspicious program. This definition includes traditional malicious samples, but also attack tools, packers, and small probes written with the only goal of exfiltrating information about the sandbox internals.

Our main goal is to automatically detect suspicious submissions that are likely related to malware development or to a misuse of the public sandbox. We also want to use the collected information for malware intelligence. In this context, *intelligence* means a process, supported by data analysis, that helps an analyst to infer the motivation, intent, and possibly the identity of the attacker.

Our analysis consists of five different phases. In the first phase, we filter out the samples that are not interesting for our analysis. Since the rest of the analysis is quite time-consuming, any sample that cannot be related to malware development or that we cannot process with our current prototype is discarded at this phase. In the second phase, we cluster the remaining samples based on their binary similarity. Samples in each cluster are then compared using a more fine-grained static analysis technique. Afterwards, we collect six sets of features, based respectively on static characteristics of the submitted files, on the results of the dynamic execution of the samples in the cluster, and on the metadata associated to the samples submissions. This features are finally provided to a classifier that we previously trained to identify the *malware development* clusters.

3 Data reduction

The first phase of our study has the objective of reducing the amount of data by filtering out all the samples that are not relevant for our analysis. We assume that a certain file could be a candidate for malware development only if two conditions are met. First, the sample must have been submitted to the public sandbox *before* it was observed in the wild. Second, it has to be part of a *manual* submission done by an individual user – and not, for example, originating from a batch submission of a security company or from an automated malware collection or protection system.

We started by filtering out the large number of batch submissions Anubis Sandbox receives from several researchers, security labs, companies, universities and registered users that regularly submit large bulks of binaries. As summarized in Table 1, with this step we managed to reduce the data from 32 million to around 6.6 million binaries. These samples have been collected by Anubis Sandbox from 2006 to 2013.

Then, to isolate the new files that were never observed in the wild, we applied a two-step approach. First, we removed those submissions that, while performed by single users, were already part of a previous batch submission. This reduced the size of the dataset to half a million samples. In the second step, we removed the files that were uploaded to the sandbox *after* they were observed by two very large external data sources: Symantec's Worldwide Intelligence Network (WINE), and VirusTotal.

After removing corrupted or not executable files (e.g, Linux binaries submitted to the Microsoft Windows sandbox), we remained with 184,548 files that match our initial definition of candidates for malware development. Before sending them to the following stages of our analysis, we applied one more filter to remove the packed applications. The rationale behind this choice is very simple. As explained in Section 4, the majority of our features work also on packed binaries, and, therefore, some potential malware development can be identified also in this category. However, it would be very hard for us to verify our results without having access to the decompiled code of the application. Therefore, in this paper we decided to focus on unpacked binaries, for which it is possible to double-check the findings of our system. The packed executables were identified by leveraging the SigBuster [37] signatures.

Table 1 summarizes the number of binaries that are filtered out after each step. The filtering phase reduced the data to be analyzed from over 32 millions to just above 121,000 candidate files, submitted by a total of 68,250 distinct IP addresses. In the rest of this section we describe in more details the nature and role of the Symantec and VirusTotal external sources.

Symantec Filter

Symantec Worldwide Intelligence Network Environment (WINE) is a platform that allows researchers to perform data intensive analysis on a wide range of cyber security relevant datasets, collected from over a hundred million hosts [28]. The data provided by WINE is very valuable for the research community, because these hosts are computers that are actively used by real users which are potential victims of various cyber threats. WINE adopts a 1:16 sampling on this large-scale data such that all types of complex experiments can be held at scale.

Dataset	Submissions
Initial Dataset	32,294,094
Submitted by regular users	6,660,022
Not already part of large submissions	522,699
Previously unknown by Symantec	420,750
Previously unknown by VirusTotal	214,321
Proper executable files	184,548
Final (not packed binaries)	121,856

Table 1: Number of submissions present in our dataset at each data reduction step.

To filter out from our analysis the binaries that are not good candidates to belong to malware development, we used two WINE datasets: the binary reputation and the AntiVirus telemetry datasets. The binary reputation dataset contains information about all of the executables (both malicious and benign) downloaded by Symantec customers over a period of approximately 5 years. To preserve the user privacy, this data is collected only from the users that gave explicit consent for it. At the time we performed our study, the binary reputation dataset included reports for over 400 millions of distinct binaries. On the other hand, the AntiVirus telemetry dataset records only the detections of known files that triggered the Norton Antivirus Engine on the users' machines.

The use of binary reputation helps us locating the exact point in time in which a binary was first disseminated in the wild. The AntiVirus telemetry data provided instead the first time the security company deployed a signature to detect the malware. We combined these datasets to remove those files that had already been observed by Symantec either before the submission to Anubis Sandbox, or within 24 hours from the time they were first submitted to the sandbox.

VirusTotal Filter

VirusTotal is a public service that provides virus scan results and additional information about hundreds of millions of analyzed files. In particular, it incorporates the detection results of over 50 different AntiVirus engines – thus providing a reliable estimation of whether a file is benign or malicious. Please note that we fetched the VirusTotal results for each file in our dataset several months (and in some cases even years) after the file was first submitted. This ensures that the AV signatures were up to date, and files were not misclassified just because they belonged to a new or emerging malware family.

Among all the information VirusTotal provides about binaries, the most important piece of information we incorporate in our study is the first submission time of a certain file to the service. We believe that by combining the timestamps obtained from the VirusTotal and Symantec datasets, we achieved an acceptable approximation of the first time a certain malicious file was observed in the wild.

4 Sample Analysis

If a sample survived the data reduction phase, it means that (with a certain approximation due to the coverage of Symantec and Virustotal datasets) it had never been observed in the wild before it was submitted to the online malware analysis sandbox. Although this might be a good indicator, it is still not sufficient to flag the submission as part of a potential malware development. In fact, there could be other possible explanations for this phenomenon, such as the fact that the binary was just a new metamorphic variation of an already known malware family.

Therefore, to reduce the risk of mis-classification, in this paper we consider a candidate for possible development only when we can observe at least two samples that clearly show the changes introduced by the author in the software. In the rest of this section we describe how we find these groups of samples by clustering similar submissions together based on the sample similarity.

4.1 Sample Clustering

In the last decade, the problem of malware clustering has been widely studied and various solutions have been proposed [31, 33, 51, 32]. Existing approaches typically use behavioral features to group together samples that likely belong to the same family, even when the binaries are quite different. Our work does not aim at proposing a new clustering method for malware. In fact, our goal is quite different and requires to group files together only when they are very similar (we are looking for small changes between two versions of the same sample) and not when they just belong to the same family. Therefore, we leverage a clustering algorithm that simply groups samples together based on their binary similarity (as computed by *ssdeep* [38]) and on a set of features we extract from the submission metadata.

Moreover, we decided to put together similar binaries into the same cluster only if they were submitted to our sandbox in a well defined time window. Again, the assumption is that when a malware author is working on a new program, the different samples would be submitted to the online sandbox in a short timeframe. Therefore, to cluster similar binaries we compute the binary similarities among all the samples submitted in a sliding window of seven days. We then shift the sliding window ahead of one day and repeat this step. We employ this sliding window approach in order (1) to limit the complexity of the computation and the total number of binary comparisons, and (2) to ensure that only the binaries that are similar and have been submitted within one week from each other are clustered together. We also experimented with other window sizes (between 2 and 15 days) but while we noticed a significant reduction of clusters for shorter thresholds, we did not observed any advantage in increasing it over one week.

Similarities among binaries are computed using the *ssdeep* [38] tool which is designed to detect similarities on binary data. *ssdeep* provides a light-weight solution for comparing a large-number of files by relying solely on similarity digests that can be easily stored in a database. As we already discarded packed binaries in the data reduction phase, we are confident that the similarity score computed by *ssdeep* is a very reliable way to group together binaries that share similar code snippets. After computing the similarity metrics, we executed a simple agglomerative clustering algorithm to group the binaries for which the similarity score is greater than 70%. Note that this step is executed separately for each time window, but it preserves transitivity between binaries in different sliding windows. For example, if file *A* is similar to *B* inside *window1*, and *B* is similar to file *C* inside the next sliding window, at the end of the process *A*, *B* and *C* will be grouped into the same cluster. As a result, a single cluster can model a malware development spanning also several months.

Starting from the initial number of binaries, we identified 5972 clusters containing an average of 4.5 elements each.

Inter-Cluster Relationships

The *ssdeep* algorithm summarizes the similarity using an index between 0 (completely different) and 100 (perfect match). Our clustering algorithm groups together samples for which the difference between the fuzzy hashes is greater than the 70% threshold. This threshold was chosen according to previous experiments [38], which concluded that 70% similarity is enough to guarantee a probability of misclassification close to zero.

However, if the malware author makes very large changes on a new version of his program, our approach may not be able to find the association between the two versions. Moreover, the final version of a malware development could be compiled with different options, making a byte-level similarity too imprecise. To mitigate these side effects, after the initial clustering step, we perform a refinement on its output by adding inter-clusters edges whenever two samples in the same time window share the same submission origin (i.e., either from the same IP address or using the same email address for the registration). These are "weak" connections that do not model a real similarity between samples, and therefore they are more prone to false positives. As a consequence, our system does not use them when performing its automated analysis to report suspicious clusters. However, as explained in Section 6, these extra connections can be very useful during the analysis of a suspicious cluster to gain a more complete picture of a malware development.

After executing this refinement step, we were able to link to our clusters an additional 10,811 previously isolated binaries. This procedure also connected several clusters together, to form 225 macro groups of clusters.

4.2 Intra-cluster Analysis

Once our system had clustered the binaries that likely belong to the same malware development, we investigate each cluster to extract more information about its characteristics. In particular, we perform a number of code-based analysis routines to understand if the samples in the same cluster share similar code-based features.

Code Normalization

Code normalization is a technique that is widely used to transform binary code to a canonical form [26]. In our study, we normalize the assembly code such that the differences between two binaries can be determined more accurately. Under the assumption that two consecutive variations of the same program are likely compiled with the same tool chain and the same options, code normalization can be very useful to remove the noise introduced by small variations between two binaries.

There are several approaches that have been proposed to normalize assembly code [36, 49, 34]. Some of them normalize just the operands, some the mnemonics, and some normalize both. In this paper, we chose to normalize only the operands so that we can preserve the semantics of the instructions. In particular, we implemented a set of IDA Pro plugins to identify all the functions in the code and then replace, for each instruction, each operand with a corresponding placeholder tag: `reg` for registers, `mem` for memory locations , `val` for constant values, `near` for near call offsets, and `ref` for references to memory locations. These IDA scripts were run in batch mode to pre-process all the samples in our clusters.

Programming Languages

The second step in our intra-cluster analysis phase consists in trying to identify the programming language used to develop the samples. The programming language can provide some hints about the type of development. For example, scripting languages are often used to develop tools or probes designed to exfiltrate information from the sandbox. Moreover, it is likely that a malware author would use the same programming language for all the intermediate versions of the same malware. Therefore, if a cluster includes samples of a malware development, all samples should typically share the same programming language. Exceptions, as the one explained in Section 6, may point to interesting cases.

To detect the programming language of a binary we implemented a simple set of heuristics that incorporate the information extracted by three tools: `PEiD`, the `pefile` python library, and the Linux `strings` command. First, we use `pefile` to parse the Import Address Table (IAT) and obtain the list of libraries that are linked to the binary. Then, we search for programming language specific keywords on the extracted list. For example, the "VB" keyword in the library name is a good indicator of using Visual Basic, and including `mscoree.dll` in the code can be linked to the usage of Microsoft `.NET`. In the second step of our analysis, we analyze the strings and the output of `PEiD` to detect compiler specific keywords (e.g., `type_info` and RTTI produced by C++ compilers, or "Delphi" strings generated by the homonymous language).

With these simple heuristics, we identified the programming language of 14,022 samples. The most represented languages are Visual Basic (49%), C (21%), Delphi (18%), Visual Basic .Net (7%), and C++ (3%). The large number of Visual Basic binaries could be a consequence of the fact that a large number of available tools that automatically create generic malware programs adopt this language.

Fine-grained Sample Similarity

In this last phase, we look in more detail at the similarity among the samples in the same cluster. In particular, we are interested to know why two binaries show a certain similarity: Did the author add a new function to the code? Did she modify a branch condition, or remove a basic block? Or maybe the code is exactly the same, and the difference is limited to some data items (such as a domain name, or a file path).

To answer these questions, we first extract the timeline of each cluster, i.e., the sequence in which each sample was submitted to the sandbox in chronological order. Moving along the timeline, we compare each couple of samples using a number of static analysis plugins we developed for IDA Pro.

The analysis starts by computing and comparing the *call graph* of the two samples. In this phase we compare the normalized code of each function, to check which functions of the second binary were present unchanged in the first binary. The output is a list of additional function that were not present in the original file, plus a list of functions that were likely modified by the author – i.e., those function that share the same position in the call graph but whose code does not perfectly match. However, at this level of granularity it is hard to say if something was modified in the function or if the author just removed the function and added another with the same callee.

Therefore, in these cases, we "zoom" into the function and repeat our analysis, this time comparing their *control flow graphs* (CFGs). Using a similar graph-based approach, this time we look for differences at the basic block level. If the two CFGs are too different, we conclude that the two functions are not one the evolution of the other. Otherwise, we automatically locate the different basic blocks and we generate a similarity measure that summarize the percentage of basic blocks that are shared by the two functions.

4.3 Feature Extraction

Based on the analysis described in the previous sections, our system automatically extracts a set of 48 attributes that we believe are relevant to study the dynamics of malware development.

This was done in two phases. First, we enriched each sample with 25 individual features, divided in six categories (see the Appendix for a complete list of individual features). The first class includes self-explanatory file features (such as its name and size). The Timestamps features identify when the sample was likely created, when it was submitted to Anubis Sandbox, and when it was later observed in the wild. While the creation time of the binary (extracted from the PE headers) could be manually faked by the author, we observed that this is seldom the case in practice, in particular when the author submits a probe or an intermediate version of a program. In fact, in these cases we often observed samples in which the compilation time precedes the submission time by only few minutes.

The third category of features contain the output of the VirusTotal analysis on the sample, including the set of labels associated by all AntiVirus software and the number of AVs that flag the sample as malicious. We then collect a number of features related to the user who submitted the sample. Since the samples are submitted using a web browser, we were able to extract information regarding the browser name and version, the language accepted by the system (sometime useful to identify the nationality of the user) and the IP from which the client was connecting from. Two features in this set require more explanation. The email address is an optional field that can be specified when submitting a sample to the sandbox web interface. The proxy flag is instead an attempt to identify if the submitter is using an anonymization service. We created a list of IP addresses related to these services and we flagged the submissions in which the IP address of the submitter appears in the blacklist. In the Binary features set we record the output of the fine-grained binary analysis scripts, including the number of sections and functions, the function coverage, and the metadata extracted by the PE files. Finally, in the last feature category we summarize the results of the sandbox behavioral report, such as the execution time, potential runtime errors, use of evasion techniques, and a number of boolean flags that represent which behavior was observed at runtime (e.g., HTTP traffic, TCP scans, etc.)

In the second phase of our analysis we extended the previous features from a single sample to the cluster that contains it. Table 2 shows the final list of aggregated attributes, most of which are obvious extensions of the values of each sample in the cluster. Some deserve instead a better explanation. For instance, the cluster `shape` (A3) describes how the samples are connected in the cluster: in a tightly connected group, in a chain in which each node is only similar to the next one, or in a mixed shape including a core group and a small tail. The `Functions diff` (B13) summarized how many functions have been modified in average between one sample and the next one. `Dev time` (B25) tells us how far apart in time each samples were submitted to the sandbox, and `Connect Back` (B24) counts how many samples in the cluster open a TCP connection toward the same /24 subnetwork from which the sample was submitted. This is a very common behavior for probes, as well as for testing the data exfiltration component of a malicious program.

Finally, some features such as the number of crashes (C8) and the average VT detection (D4) are not very interesting per se, but they become more relevant when compared with the number of samples in the cluster. For example, imagine a cluster containing three very similar files. Two of them run without errors, while the third one crashes. Or two of them are not detected by AV signatures, but one is flagged as malware by most of the existing antivirus software.

While we are aware of the fact that each feature could be easily evaded by a motivated attacker, as described in Section 6 the combinations of all them is usually sufficient to identify a large number of development clusters. Again, our goal is to show the feasibility of this approach and draw attention to a new problem, and not to propose its definitive solution.

A: Cluster Features	
A.1 Cluster_id	The ID of the cluster
A.2 Num Elements	The number of samples in the cluster
A.3 Shape	An approximation of the cluster shape (GROUP—MIX—CHAIN)
B: Samples Features	
B.1-4 Filesize stats	Min, Max, Avg, and Variance of the samples filesize
B.5-8 Sections stats	Min, Max, Avg, and Variance of the number of sections
B.9-12 Functions stats	Min, Max, Avg, and Variance of the number of functions
B.13 Functions diff	Average number of different functions
B.14 Sections diff	Average number of different sections
B.15 Changes location	One of: Data, Code, Both, None
B.16 Prog Languages	List of programming languages used during the development
B.17 Filename Edit Distance	The Average edit distance of the samples's filenames
B.18 Avg Text Coverage	Avg text coverage of the .text sections
B.19-22 CTS Time	Min, Max, Avg, and Variance of the difference between compile and the submission time
B.23 Compile time Flags	Booleans to flag `NULL` or constant compile times
B.24 Connect back	True if any file in the cluster contacts back the submitter's /24 network
B.25 Dev time	Average time between each submission (in seconds)
C: Sandbox Features	
C.1 Sandbox Only	Numer of samples seen only by the sandbox (and not from external sources)
C.2 Short Exec	Number of samples terminating the analysis in less than 60s
C.4-6 Exec Time	Min, Max, and Avg execution time of the samples within the sandbox
C.7 Net Activity	The number of samples with network activity
C.7 Time Window	Time difference between first and last sample in the cluster (in days)
C.8 Num Crashes	Number of samples crashing during their execution inside the sandbox
D: Antivirus Features	
D.1-3 Malicious Events	Min, Max, Avg numbers of behavioral flags exibited by the samples
D.4-5 VT detection	Average and Variance of VirusTotal detection of the samples in the cluster
D.6 VT Confidence	Confidence of the VirusTotal score
D.7 Min VT detection	The score for the sample with the minimum VirusTotal Detection
D.8 Max VT detection	The score for the sample with the maximum VirusTotal Detection
D.9 AV Labels	All the AV labels for the identified pieces of malware in the cluster
E: Submitter Features	
E.1 Num IPs	Number of unique IP addresses used by the submitter
E.2 Num E-Mails	Number of e-mail addresses used by the submitter
E.3 Accept Languages	Accepted Languages from the submitter's browser

Table 2: List of Features associated to each cluster

	AUC	Det. Rate	False Pos.
Full data	0.999	98.7%	0%
10-folds Cross-Validation	0.988	97.4%	3.7%
70% Percentage Split	0.998	100%	11.1%

Table 3: Classification accuracy, including detection and false positive rates, and the Area Under the ROC Curve (AUC)

5 Machine Learning

Machine learning provides a very powerful set of techniques to conduct automated data analysis. As the goal of this paper is to automatically distinguishing malware developments from other submissions, we tested with a number of machine learning techniques applied to the set of features we presented in detail in the previous section.

Among the large number of machine learning algorithms we have tested our training data with, we have obtained the best results by using the logistic model tree (LMT). LMT combines the logistic regression and decision tree classifiers by building a decision tree whose leaves have linear regression models [41].

Training Set

The most essential phase of machine learning is the training phase where the algorithm learns how to distinguish the characteristics of different classes. The success of the training phase strictly depends on a carefully prepared labeled data. If the labeled data is not prepared carefully, the outcome of machine learning can be misleading. To avoid this problem, we manually labeled a number of clusters that were randomly chosen between the ones created at the end of our analysis phase. Manual labeling was carried out by an expert that performed a manual static analysis of the binaries to identify the type and objective of each modification. With this manual effort, we flagged 91 clusters as non-development and 66 as development. To estimate the accuracy of the LMT classifier, we conducted a 10-fold cross validation and a 70% percentage split evaluation on the training data.

Feature Selection

In the previous section, we have presented a comprehensive set of features that we believe can be related to the evolution of samples and to distinguish malware developments from ordinary malware samples. However, not all the features contribute in the same way to the final classification, and some works well only when used in combination with other classes.

To find the subset of features that achieves the optimal classification accuracy while helping us to obtain the list of features that contribute the most to it, we leveraged a number of features selection algorithms that are widely used in machine learning literature: Chi-Square, Gain Ratio and Relief-F attribute evaluation. Chi-square attribute evaluation computes the chi-square statistics of each feature with respect to the class, which in our case is the fact of being a malware development or not. The Gain Ratio evaluation, on the other hand, evaluates the effect of the feature by measuring its gain ratio. Finally, the Relief-F attribute evaluation methodology assigns particular weights to each feature according to how much they are successful to distinguish the classes from each other. This weight computation is based on the comparison of the probabilities of two nearest neighbors having the same class and the same feature value.

While the order slightly differs, the ten most effective features for the accuracy of the classifier for all three feature selection algorithms are the same. As also the common sense suggests, the features we extract from the binary similarity and the analysis of the samples are the most successful. For example, the connect back feature that checks if the sample connects back to the same IP address of the submitter, the average edit distance of the filenames of the samples, the binary function similarity, and the sample compile time features are constantly ranked on the top of the list. The submitter features and the sandbox features are following the sample features in the list. All of the features except the number of sandbox evasions, the VirusTotal results, and the features we extracted from the differences on the file sizes in the clusters had a contribution to the accuracy. After removing those features, we performed a number of experiments on the training set to visualize the contribution of the different feature sub-sets to the classification accuracy. Figure 1 shows (in log-scale) the impact of each class and combination of classes. Among all the classes the samples-based features produced the best combination of detection and false positive rates (i.e. 88.2% detection rate with 7.4% false positives). In particular, the ones based on the static and dynamic analysis of the binaries seem to be the core of the detection ability of the system. Interestingly, the cluster-based features alone are the worst between all sets, but they increase the accuracy of the final results when combined with other features.

The results of the final classifier are reported in Table 3: 97.4% detection with of 3.7% false positives, according to 10-folds cross validation experiment. Note that we decided to tune the classifier to favor detection over false positives, since the goal of our system is only to tag suspicious submissions that would still need to be manually verified by a malware analyst.

6 Results

Our prototype implementation was able to collect substantial evidences related to a large number of malware developments.

In total, our system flagged as potential development 3038 clusters over a six years period. While this number was too large for us to perform a manual verification of each case, if such a system would be deployed we estimate between two and three alerts generated per day. Therefore, we believe our tool could be used as part of an early warning mechanism to automatically collect information about suspicious submissions and report them to human experts for further investigation.

In addition to the 157 clusters already manually labeled to prepare the training set for the machine learning component, we also manually verified 20 random clusters automatically flagged as suspicious by our system. Although according to the 10-fold cross validation experiments the false positive rate is 3.7%, we have not found any false positives on the clusters we randomly selected for our manual validation.

Our system automatically detected the development of a diversified group of real-world malware, ranging from generic trojans to advanced rootkits. To better understand the distribution of the different malware families, we verified the AV labels assigned to each reported cluster. According to them, 1474 clusters were classified as malicious, out of which our system detected the development of 45 botnets, 1082 trojans, 83 backdoors, 4 keyloggers, 65 worms, and 21 malware development tools (note that each development contained several different samples modeling intermediate steps). A large fraction

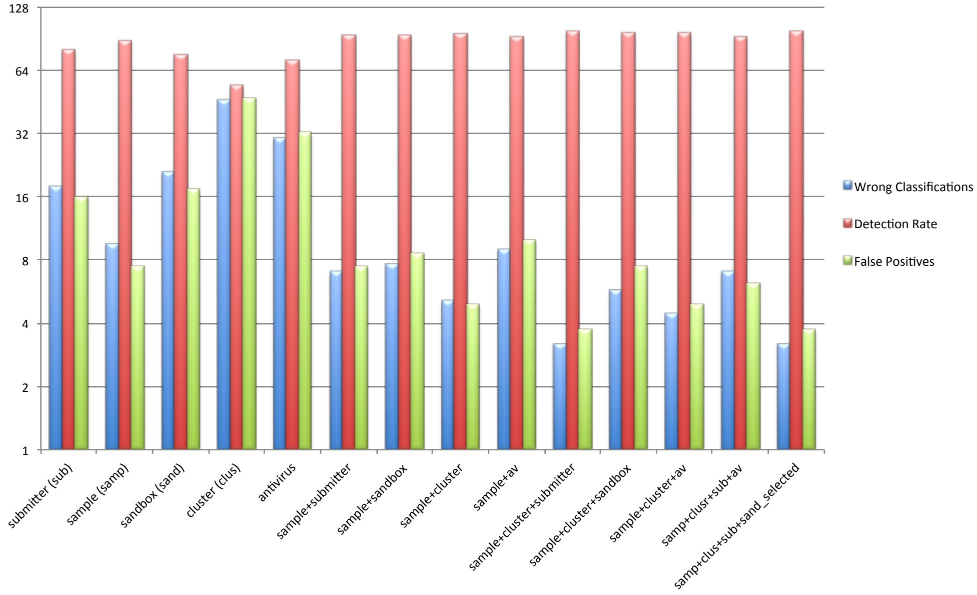


Figure 1: Classification success of different feature combinations.

Campaign	Early Submission	Time Before Public Disclosure	Submitted by
Operation Aurora	✓	4 months	US
Red October	✓	8 months	Romania
APT1	✓	43 months	US
Stuxnet	✓	1 months	US
Beebus	✓	22 months	Germany
LuckyCat	✓	3 months	US
BrutePOS	✓	5 months	France
NetTraveller	✓	14 months	US
Pacific PlugX	✓	12 months	US
Pitty Tiger	✓	42 months	US
Regin	✓	44 months	UK
Equation	✓	23 months	US

Table 4: Popular campaigns of targeted attacks in the sandbox database

of the clusters that were not identified by the AV signatures contained the development of probes, i.e., small programs whose goal is only to collect and transmit information about the system where they run. Finally, some clusters also contained the development or testing of offensive tools, such as packers and binders.

6.1 Targeted Attacks Campaigns

Before looking at some of the malware development cases detected by our system, we wanted to verify our initial hypothesis that even very sophisticated malware used in targeted attacks are often submitted to public sandboxes months before the real attacks are discovered. For this reason, we created a list of hashes of known and famous APT campaigns, such as the ones used in operation Aurora and Red October. In total, we collected 1271 MD5s belonging to twelve different campaigns. As summarized in Table 4, in all cases we found at least one sample in our database before the campaign was publicly discovered (*Early Submission* column). For example, for Red October the first sample was submitted in February 2012, while the campaign was later detected in October

2012. The sample of Regin was collected a record 44 months before the public discovery.

Finally, we checked from whom those samples were submitted to the system. Interestingly, several samples were first submitted by large US universities. A possible explanation is that those samples were automatically collected as part of a network-based monitoring infrastructure maintained by security researchers. Other were instead first submitted by individual users (for whom we do not have much information) from several different countries, including US, France, Germany, UK, and Romania. Even more interesting, some were first submitted from DSL home Internet connections. However, we cannot claim that we observed the development phase of these large and popular targeted attacks campaigns as in all cases the samples were already observed in the wild (even though undetected and no one was publicly aware of their existence) before they were submitted to our sandbox. It is important to note that for this experiment we considered the entire dataset, without applying any filtering and clustering strategy. In fact, in this case we did not want to spot the *development* of the APT samples, but simply the fact that those samples were submitted and available to researchers long before they were publicly discovered.

We believe the sad message to take away from this experiment is that all those samples went unnoticed. As a community, there is a need for some kind of early warning system to report suspicious samples to security researches. This could prevent these threats from flying under the radar and could save months (or even years) of damage to the companies targeted by these attacks.

6.2 Case studies

In the rest of this section we describe in more details three development scenarios. While our system identified many more interesting cases, due to space limitation we believe the following brief overview provides a valuable insight on the different ways in which attackers use (and misuse) public sandboxes. Moreover, it also shows how a security analyst can use the information collected by our system to investigate each case, and reconstruct both the author behavior and his final goal.

In the first example, the malware author introduced an anti-sandbox functionality to a Trojan application. In this case the analyst gathers intelligence information about the modus operandi of the attacker and about all the development phases.

In the second scenario, we describe a step by step development in which the attacker tries to collect information from the sandbox. This information is later used to detect the environment and prevent the execution of a future malware in the sandbox. In the last example, we show how an attacker uses the sandbox as a testbed to verify the behavior of the malware. In this case, the author generated the binary using one of the many dedicated builder applications that can be downloaded from the Internet or bought on the black market.

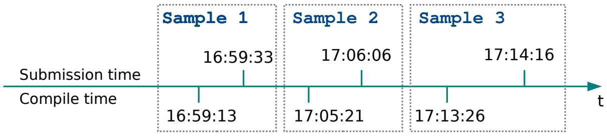


Figure 2: Anti-sandbox check - Timeline

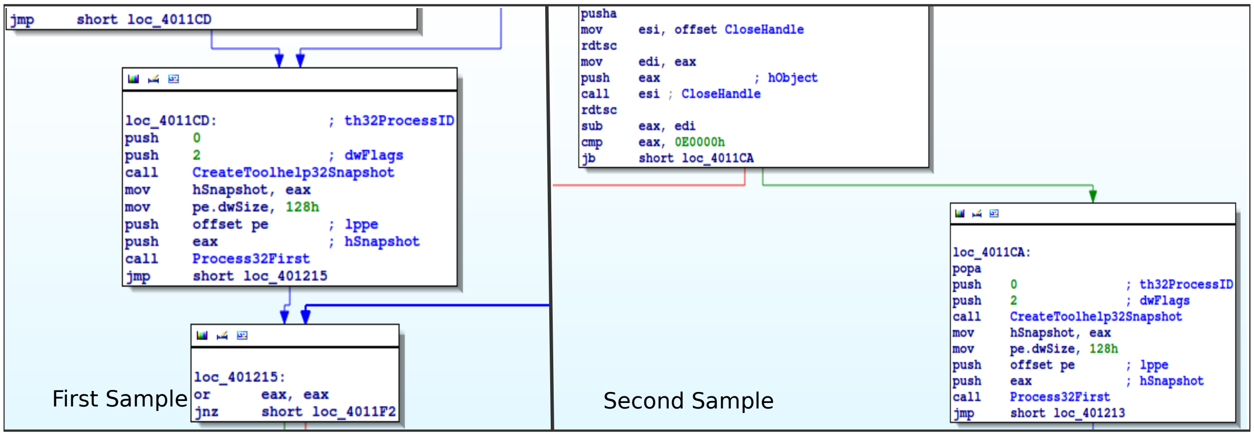
Example I: Anti-sandbox Malware

The cluster related to this example contains three samples. The timeline (summarized in Figure 2) already suggests a possible development. In fact, the difference between the `submission` time and the `compile` time is very small.

A quick look at the static features of the cluster shows that the three samples are very similar, and share the same strings as well as the same imphash (the *import hash* [20, 21] recently introduced also by VirusTotal). However, the first sample is composed of 21 functions, while the last two samples have 22 functions. Our report also shows how the first and the second samples differ for two functions: the author modified the function `start`, and introduced a new function `CloseHandle`. This information (so far extracted completely automatically by our system) is a good starting point for a closer analysis.

We opened the two executables in IDA Pro, and quickly identified the two aforementioned functions (snippet in Figure 3). It was immediately clear that the `start` function was modified to add an additional basic block and a call to the new `CloseHandle` function. The new basic block uses the `rdtsc` x86 instruction to read the value of the Timestamp Counter Register (TSC), which contains the number of CPU cycles since the last reset. The same snippet of assembly is called two times to check the time difference. After the first `rdtsc` instruction there is a call to `CloseHandle`, using the timestamp as handler (probably an invalid handler). These two well known tricks are here combined to detect the Anubis Sandbox environment – due to the delay introduced by its checks during program execution. The Anubis Sandbox's core is slower in looking up the handlers table, and this time discrepancy is the key to detect the analysis environment. In this case the difference has to be less than `0E0000h`, or the program would immediately terminate by calling the `ExitProcess` function.

The last sample in the cluster was submitted only to tune the threshold and for this reason there were no important differences with the second sample. The *control*



```
jmp     short loc_4011CD

loc_4011CD:                  ; th32ProcessID
push    0
push    2                    ; dwFlags
call    CreateToolhelp32Snapshot
mov     hSnapshot, eax
mov     pe.dwSize, 128h
push    offset pe            ; lppe
push    eax                  ; hSnapshot
call    Process32First
jmp     short loc_401215

loc_401215:
or      eax, eax
jnz     short loc_4011F2
```

```
pusha
mov     esi, offset CloseHandle
rdtsc
mov     edi, eax
push    eax             ; hObject
call    esi ; CloseHandle
rdtsc
sub     eax, edi
cmp     eax, 0E0000h
jb      short loc_4011CA

loc_4011CA:
popa
push    0                    ; th32ProcessID
push    2                    ; dwFlags
call    CreateToolhelp32Snapshot
mov     hSnapshot, eax
mov     pe.dwSize, 128h
push    offset pe            ; lppe
push    eax                  ; hSnapshot
call    Process32First
jmp     short loc_401213
```

Figure 3: Anti-sandbox check - Start function comparison

flow graph analysis performed automatically by our system report a very high similarity between the first two samples, in line with the little modifications we found in the disassembled code. Finally, the behavioral features extracted by our system confirm our hypothesis: the first sample was executed until the analysis timeout, but the execution of the second one terminated after only five seconds.

The behavior described so far suggest malicious intents. This is also confirmed by other cluster metadata. For instance, while the first sample in the cluster was unknown to VirusTotal, the last one was clearly identified as a common `Trojan` application. This suggests that the original sample, without the timing check, has never been used in the wild. Once more, the fact that all three samples have been submitted days before the trojan was first observed in the wild strongly supports the fact that the person who submitted them was indeed the malware author.

Example II: Testing a Trojan Dropper

The second cluster we want to describe is composed of five samples. Our report indicates that the first four are written in Delphi and the last one is written in Visual Basic. This is already a strange fact, since the two programming languages are quite different and it is unlikely that they could generate similar binaries.

In this case the cluster timeline does not provide useful information as all the Delphi samples share exactly the same compilation time: 20th of June, 1992. Only the Visual Basic sample had a compilation time consistent with the submission. On the contrary, the submission times provide an interesting perspective. All the samples have been submitted in few hours and this might indicate a possible development. In addition, there are two IP addresses involved: one for the four Delphi samples and one for the final Visual Basic version. The static features of the first four samples show very little differences, suggesting that these are likely just small variations of the same program. In average, they share 169 out of 172 functions and 7 out of 8 PE sections. By inspecting the changes, we notice that the attacker was adding some threads synchronization code to a function responsible for injecting code into a different process. The *control flow graph* similarity reported by our tool was over 98%, confirming the small differences we observed between each versions. Once the author was happy with the result, she submitted one more sample, this time completely different from the previous ones. Despite the obvious differences in most of the static analysis features, the fuzzyhash similarity with sample 4 was 100%. A rapid analysis showed that this perfect match was due to the fact that the Visual Basic application literally embedded the entire binary of the fourth Delphi program. In addition, the behavior report confirmed that, once executed, the Visual Basic Trojan dropped the embedded executable that was later injected inside a target process. None of the Antivirus software used by VirusTotal recognized the first four samples as malicious. However, the last one was flagged by 37 out of 50 AVs as a *trojan dropper* malware.

It is important to stress that a clear advantage of our system is that it was able to automatically reconstruct the entire picture despite the fact that not all samples were submitted from the same IP address (even though all located in the same geographical area). Moreover, we were able to propagate certain metadata extracted by our system (for example the username of the author extracted from the binary compiled with Visual Studio) from one sample to the others in which that information was missing. This ability to retrieve and propagate metadata between different samples can be very useful during an investigation.

Another very interesting aspect of this malware development is the fact that after the process injection, the program used a well known dynamic DNS service (`no-ip`) to resolve a domain name. The IP address returned by the DNS query pointed exactly to the same machine that was used by the author to submit the sample. This suggests that the attacker was indeed testing his attack before releasing it, and this information could be used to locate the attacker machine.

We identified a similar *connect-back* behavior in other 1817 clusters. We also noticed how most of these clusters contain samples generated by known trojan *builders*, like Bifrost [8] or PoisonIvy [9]. While this may seem to prove that these are mostly unsophisticated attacks, FireEye [22] recently observed how the Xtremerat builder [7] (which appeared in 28 of our clusters) was used to prepare samples used in several targeted attacks.

Example III: Probe Development

In this last example we show an attacker fingerprinting the analysis environment and how, at the end, she manages to create her own successful antisandbox check. The cluster consists of two samples, both submitted from France in a time span of 23 hours by the same IP address. The two samples have the same size, the same number of functions (164), and of sections (4). There is only one function (`_start`) and two sections (`.text` and `.rdata`) presenting some differences. The two programs perform the same actions, they create an empty text file and then they retrieve the file attributes through the API `GetFileAttributes`. The only differences are on the API version they use (`GetFileAttributesA` or `GetFileAttributesW`) and on the file name to open.

At a first look, this cluster did not seem very interesting. However the inter-cluster connections pointed to other six loosely correlated samples submitted by the same author in the same week. As explained in Section 4, these files have not been included in the core cluster because the binary similarity was below our threshold. In this case, these samples were all designed either to collect information or to test anti-virtualization/emulation tricks. For instance, one binary implemented all the known techniques based on `idt`, `gdt` and `ldt` to detect a virtual machine monitor [48, 47, 42]. Another one simply retrieved the computer name, and another one was designed to detect the presence of inline hooking. Putting all the pieces together, it is clear that the author was preparing a number of probes to assess various aspects of the sandbox environment.

This example shows how valuable the inter-clusters edges can be to better understand and link together different submissions that, while different between each other at a binary level, are likely part of the same organized "campaign".

6.3 Malware Samples in the Wild

As we already mentioned at the beginning of the section, out of 3038 clusters reported as malware development candidates by our machine learning classifier, 1474 (48%) contained binaries that were detected by the antivirus signatures as malicious (according to VirusTotal).

A total of 228 of the files contained in these clusters were later detected in the wild by the Symantec's antivirus engine. The average time between the submission to our sandbox and the time the malware was observed in the wild was 135 days – i.e., it took between four and five months for the antivirus company to develop a signature and for the file to appear on the end-users machines. Interestingly, some of these binaries were later detected on more than 1000 different computers in 13 different countries all around the world (obviously a lower bound, based on the alerts triggered on a subset of the Symantec's customers). This proves that, while these may not be very sophisticated malware, they certainly have a negative impact on thousands of normal users.

7 Limitations

We are aware of the fact that once this research is published, malware authors can react and take countermeasures to sidestep this type of analysis systems. For instance, they may decide to use "private" malware checkers, and avoid interacting with public sandboxes altogether. First of all, this is a problem that applies to many analysis techniques ranging from botnet detection, to intrusion prevention, to malware analysis. Despite that, we believe that it is important to describe our findings so that other researchers can work in this area and propose more robust methodologies in the future.

Moreover, as we mentioned in the introduction, after we completed our study someone noticed that some known malware development groups were testing their creation on VirusTotal [52, 27]. This confirms that what we have found is not an isolated case but a widespread phenomenon that also affects other online analysis systems. Second, now that the interaction between malware developers and public sandboxes is not a secret anymore, there is no reason that prevents us from publishing our findings as well.

We are aware of the fact that our methodology is not perfect, that it can be evaded, and that cannot catch all development cases. However, we believe the key message of the paper is that malware authors are abusing public sandboxes to test their code, and at the moment we do not need a very sophisticated analysis to find them. Since this is the first paper that tries to identify these cases, we found that our approach was already sufficient to detect thousands of them. Certainly more research is needed in this area to develop more precise monitoring and early warning system to analyze the large amounts of data automatically collected by public services on a daily basis.

8 Related Work

While there has been an extensive amount of research on malware analysis and detection, very few works in the literature have studied the datasets collected by public malware dynamic analysis sandboxes. The most comprehensive study in this direction was conducted by Bayer et al. [24]. The authors looked at two years of Anubis [10] reports and they provided several statistics about malware evolution and about the prevalent types of malicious behaviors observed in their dataset.

Lindorfer et al. [43] conducted the first study in the area of malware development by studying the evolution over time of eleven known malware families. In particular, the authors documented the malware updating process and the changes in the code for a number of different versions of each family. In our study we look at the malware development process from a different angle. Instead of studying different versions of the same well known malware, we try to detect, on a large scale, the authors of the malware at the moment in which they interact with the sandbox itself. In a different paper, Lindorfer et al. [44] proposed a technique to detect environment sensitive malware. The idea is to execute each malware sample multiple times on several sandboxes equipped with different monitoring implementations and then compare the normalized reports to detect behavior discrepancies.

A similar research area studies the phylogeny [30] of malware by using approaches taken from the biology field. Even if partially related to our work, in our study we were not interested in understanding the relationship between different species of malware, but only to detect suspicious submissions that may be part of a malware development activity.

In a paper closer to our work, Jang et al. [34] studied how to infer the software evolution looking at program binaries. In particular, the authors used both static and dynamic analysis features to recover the software lineage. While Jang's paper focused mostly on benign programs, some experiments were also conducted on 114 malicious software with known lineage extracted from the Cyber Genome Project [12]. Compared to our work, the authors used a smaller set of static and dynamic features especially designed to infer the software lineage (e.g., the fact that a linear development is characterized by a monotonically increasing file size). Instead, we use a richer set of features to be able to distinguish malware developments from variations of the same samples collected on the wild and not submitted by the author. While our approaches share some similarities, the goals are clearly different.

Other approaches have been proposed in the literature to detect similarities among binaries. Flake [29] proposed a technique to analyze binaries as graphs of graphs, and we have been inspired by his work for the *control flow analysis* described in Section 4. Kruegel et al. [39] proposed a similar technique in which they analyzed the control flow graphs of a number of worms and they used a graph coloring technique to cope with the graph-isomorphism problem.

Finally, one step of our technique required to cluster together similar malware samples. There are several papers in the area of malware clustering [31, 33, 51, 32]. However, their goal is to cluster together samples belonging to the same malware family as fastest as possible and with the highest accuracy. This is a crucial task for all the Antivirus companies. However, our goal is different as we are interested in clustering samples based only on binary similarity and we do not have any interest in clustering together members of the same family based on their behavior.

9 Conclusion

Public dynamic analysis sandboxes collect thousands of new malware samples every day. Most of these submissions belong to well known malware families, or are benign files that do not pose any relevant security threat. However, hidden in this large amount of collected data, few samples have something special that distinguishes them from the rest. In this paper, we discussed the importance of looking at these samples from an intelligence and threat prevention point of view.

We show that several binaries used in the most famous targeted attack campaigns had been submitted to our sandbox months before the attack was first reported. Moreover, we propose a first attempt to mine the database of a popular sandbox, looking for signs of malware development. Our experiments show promising results. We were able to automatically identify thousands of developments, and to show how the authors modify their programs to test their functionalities or to evade detections from known sandboxes. Around 1,500 of them were real malware developments – some of which have been later observed on thousands of infected machines around the world.

Acknowledgment

We would like to thank Claudio Guarnieri for the fruitful discussions and insights.

References

[1] Amnpardaz SandBox Jevereg. http://jevereg.amnpardaz.com/.

[2] AV Tracker. http://avtracker.info/.

[3] Comodo Instant Malware Analysis. http://camas.comodo.com/.

[4] ThreatTrack Security Public Sandbox. http://www.threattracksecurity.com/resources/sandbox-malware-analysis.aspx.

[5] ViCheck. https://www.vicheck.ca.

[6] Xandora - Suspicious File Analyzer. http://www.xandora.net/xangui/.

[7] Xtreme RAT. https://sites.google.com/site/xxtremerat/.

[8] Bifrost Builder. http://www.megasecurity.org/trojans/b/bifrost/Bifrost2.0special.html, 2008.

[9] Poison Ivy RAT. http://www.poisonivy-rat.com, 2008.

[10] Anubis. http://anubis.iseclab.org, 2009.

[11] A new approach to China. http://googleblog.blogspot.fr/2010/01/new-approach-to-china.html, 2010.

[12] Darpa Cyber Genome Project. https://www.fbo.gov/index?s=opportunity&mode=form&id=c34caee99a41eb14d4ca81949d4f2fde, 2010.

[13] Malwr. https://malwr.com, 2010.

[14] ThreatExpert. http://www.threatexpert.com/, 2010.

[15] Malbox. http://malbox.xjtu.edu.cn/, 2011.

[16] Virustotal += Behavioural Information. http://blog.virustotal.com/2012/07/virustotal-behavioural-information.html, 2012.

[17] The Red October Campaign - An Advanced Cyber Espionage Network Targeting Diplomatic and Government Agencies. https://www.securelist.com/en/blog/785/, 2013.

[18] TotalHash. http://totalhash.com/, 2013.

[19] RDG Tejon Crypter. http://blackshop.freeforums.org/rdg-tejon-crypter-2014-t743.html, 2014.

[20] Tracking Malware with Import Hashing. https://www.mandiant.com/blog/tracking-malware-import-hashing/, 2014.

[21] VirusTotal += imphash. http://blog.virustotal.com/2014/02/virustotal-imphash.html, 2014.

[22] XtremeRAT: Nuisance or Threat? http://www.fireeye.com/blog/technical/2014/02/xtremerat-nuisance-or-threat.html, 2014.

[23] Balzarotti, D., Cova, M., Karlberger, C., Kruegel, C., Kirda, E., and Vigna, G. Efficient Detection of Split Personalities in Malware. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, February 2010), NDSS 10.

[24] Bayer, U., Habibi, I., Balzarotti, D., Kirda, E., and Kruegel, C. A view on current malware behaviors. In *USENIX workshop on large-scale exploits and emergent threats (LEET)* (April 2009), LEET 09.

[25] Bilge, L., and Dumitras, T. Before we knew it: An empirical study of zero-day attacks in the real world. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 833–844.

[26] Bruschi, D., Martignoni, L., and Monga, M. Using Code Normalization for Fighting Self-Mutating Malware. In *Proceedings of the International Symposium of Secure Software Engineering (ISSSE)* (Mar. 2006), IEEE Computer Society. Arlington, VA, USA.

[27] Dixon, B. Watching attackers through virustotal. http://blog.9bplus.com/watching-attackers-through-virustotal/, 2014.

[28] Dumitras, T., and Shou, D. Toward a standard benchmark for computer security research: The worldwide intelligence network environment (wine). In *Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security* (2011), BADGERS '11.

[29] Flake, H. Structural comparison of executable objects. In *In Proceedings of the IEEE Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA* (2004), pp. 161–173.

[30] Hayes, M., Walenstein, A., and Lakhotia, A. Evaluation of malware phylogeny modelling systems using automated variant generation, 2009.

[31] Hu, X., Bhatkar, S., Griffin, K., and Shin, K. G. Mutantx-s: Scalable malware clustering based on static features. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (Berkeley, CA, USA, 2013), USENIX ATC'13, USENIX Association, pp. 187–198.

[32] Jacob, G., Comparetti, P. M., Neugschwandtner, M., Kruegel, C., and Vigna, G. A static, packer-agnostic filter to detect similar malware samples. In *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (Berlin, Heidelberg, 2013), DIMVA'12, Springer-Verlag, pp. 102–122.

[33] Jang, J., Brumley, D., and Venkataraman, S. Bitshred: Feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2011), CCS '11, pp. 309–320.

[34] Jang, J., Woo, M., and Brumley, D. Towards automatic software lineage inference. In *Proceedings of the 22Nd USENIX Conference on Security* (Berkeley, CA, USA, 2013), SEC'13, USENIX Association, pp. 81–96.

[35] Kaspersky GReAT Team. Equation: The death star of malware galaxy. http://securelist.com/blog/research/68750/equation-the-death-star-of-malware-galaxy/, 2015.

[36] Khoo, W. M., and Lio, P. Unity in diversity: Phylogenetic-inspired techniques for reverse engineering and detection of malware families. *SysSec Workshop* (2011), 3–10.

[37] Koivunen, T. Sigbuster. http://www.teamfurry.com", 2009.

[38] Kornblum, J. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation 3, Supplement*, 0 (2006), 91 – 97.

[39] Kruegel, C., Kirda, E., Mutz, D., Robertson, W., and Vigna, G. Polymorphic worm detection using structural information of executables. In *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection* (Berlin, Heidelberg, 2006), RAID'05, Springer-Verlag, pp. 207–226.

[40] Kurtz, G. Operation Aurora hit Google, Others. http://web.archive.org/web/20100327181927/http://siblog.mcafee.com/cto/operation-%E2%80%9Caurora%E2%80%9D-hit-google-others, 2010.

[41] LANDWEHR, N., HALL, M., AND FRANK, E. Logistic model trees. In *Machine Learning: ECML 2003* (2003), Springer Berlin Heidelberg, pp. 241–252.

[42] LIGH, M. Using IDT for VMM Detection. `http://www.mnin.org/?page=vmmdetect`.

[43] LINDORFER, M., DI FEDERICO, A., MAGGI, F., MILANI COMPARETTI, P., AND ZANERO, S. Lines of Malicious Code: Insights Into the Malicious Software Industry. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)* (2012).

[44] LINDORFER, M., KOLBITSCH, C., AND MILANI COMPARETTI, P. Detecting Environment-Sensitive Malware. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID)* (2011).

[45] MORGAN MARQUIS-BOIRE, CLAUDIO GUARNIERI, AND RYAN GALLAGHER. Secret malware in european union attack linked to u.s. and british intelligence. `https://firstlook.org/theintercept/2014/11/24/secret-regin-malware-belgacom-nsa-gchq/`, 2014.

[46] MOSER, A., KRUEGEL, C., AND KIRDA, E. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2007), SP '07, IEEE Computer Society, pp. 231–245.

[47] QUIST, D., AND SMITH, V. Detecting the Presence of Virtualmachines Using the Local Data Table. `http://www.offensivecomputing.net/files/active/0/vm.pdf`.

[48] RUTKOWSKA, J. Red Pill... or how to detect VMM using (almost) one CPU instruction. `http://web.archive.org/web/20070911024318/http://invisiblethings.org/papers/redpill.html`, 2004.

[49] SÆBJØRNSEN, A., WILLCOCK, J., PANAS, T., QUINLAN, D., AND SU, Z. Detecting code clones in binary executables. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* (2009), ISSTA '09.

[50] SYMANTEC SECURITY RESPONSE. Regin: Top-tier espionage tool enables stealthy surveillance. `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/regin-analysis.pdf`, 2014.

[51] WICHERSKI, G. pehash: A novel approach to fast malware clustering. In *Proceedings of the 2Nd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More* (Berkeley, CA, USA, 2009), LEET'09, USENIX Association, pp. 1–1.

[52] ZETTER, K. A google site meant to protect you is helping hackers attack you. `http://www.wired.com/2014/09/how-hackers-use-virustotal/`, 2014.

A Individual Sample Features

A: File Features	
A.1 Filename	The original name of the file submitted by the user
A.2 File size	The size of the file
A.3 MD5	Simple hash used for lookup in other data sources
A.4 Fuzzy Hashes	Using SSDeep algorithm
B: Timestamps	
B.1 Submission time	Time in which the sample was submitted to Anubis Sandbox
B.2 Compile time	Time in which the binary was compiled
B.3 Symantec first	Time the sample was first observed in the wild by Symantec
B.4 VirusTotal first	Time in which the binary was first submitted to VirusTotal
C: AV Features	
C.1 AV-Detection	Number of AV that flag the samples as malicious (according to VirusTotal)
C.2 AV-Labels	List of AV labels associated to the sample (according to VirusTotal)
D: User-based Features	
D.1 User Agent	User agent of the browser used to submit the sample
D.2 Languages	Languages accepted by the user browser (according to the `accept-language` HTTP header)
D.3 IP	IP address of the user who submitted the file
D.4 IP Geolocation	Geolocation of the user IP address
D.5 Email address	Optional email address specified when the sample was submitted
D.6 Proxy	Boolean value used to identify submission through popular anonymization proxies
E: Binary Features	
E.1 N.Sections	Number of sections in the PE file
E.2 N.Fuctions	Number of functions identified by the disassembly
E.3 Code Coverage	Fraction of `.text` segment covered by the identified functions
E.4 Programming Language	Programming language used to develop the binary
E.5 Metadata	Filenames and username extracted from the PE file
F: Behavioral Features	
F.1 Duration	Duration in seconds of the analysis
F.2 Errors	Error raised during the analysis
F.3 Evasion	Known anti-sandbox techniques detected by the sandbox itself
F.4 Behavior Bitstring	Sequence of 24 boolean flags that characterize the behavior of the sample. (`has_popups`, `has_udp_traffic`, `has_http`, `has_tcp_address_scan`, `modified_registry_keys`, ...)

Table 5: List of Individual Features associated to each sample

USENIX Association

Supplement to the Proceedings of the
22nd USENIX Security Symposium

August 14–16, 2013
Washington, D.C.

Message from the 22nd USENIX Security Symposium Program Chair and USENIX Executive Director

In this supplement to the Proceedings of the 22nd USENIX Security Symposium, we are pleased to announce the publication of the paper, "Dismantling Megamos Crypto: Wirelessly Lockpicking a Vehicle Immobilizer," by Roel Verdult, Flavio D. Garcia, and Baris Ege. This paper, which was accepted by the USENIX Security '13 Program Committee, was withdrawn from publication by its authors in response to the imposition of an injunction by the High Court of Justice in the United Kingdom prohibiting the authors, their institutions, and anyone who assists them, from publishing key sections of the paper. We now join the authors in their delight that USENIX may now publish their paper in this supplement to the original Proceedings. Verdult and Garcia will present the paper in a special evening session during the 24th USENIX Security Symposium. Although two years have passed, this work remains important and relevant to our community.

Sam King, *USENIX Security '13 Program Chair*
Casey Henderson, *USENIX Executive Director*

Dismantling Megamos Crypto: Wirelessly Lockpicking a Vehicle Immobilizer

Roel Verdult
Institute for Computing and Information Sciences,
Radboud University Nijmegen, The Netherlands.
rverdult@cs.ru.nl

Flavio D. Garcia
School of Computer Science,
University of Birmingham, UK.
f.garcia@cs.bham.ac.uk

Barış Ege
Institute for Computing and Information Sciences,
Radboud University Nijmegen, The Netherlands.
b.ege@cs.ru.nl

Abstract

The Megamos Crypto transponder is used in one of the most widely deployed electronic vehicle immobilizers. It is used among others in most Audi, Fiat, Honda, Volkswagen and Volvo cars. Such an immobilizer is an anti-theft device which prevents the engine of the vehicle from starting when the corresponding transponder is not present. This transponder is a passive RFID tag which is embedded in the key of the vehicle.

In this paper we have reverse-engineered all proprietary security mechanisms of the transponder, including the cipher and the authentication protocol which we publish here in full detail. This article reveals several weaknesses in the design of the cipher, the authentication protocol and also in their implementation. We exploit these weaknesses in three practical attacks that recover the 96-bit transponder secret key. These three attacks only require wireless communication with the system. Our first attack exploits weaknesses in the cipher design and in the authentication protocol. We show that having access to only two eavesdropped authentication traces is enough to recover the 96-bit secret key with a computational complexity of 2^{56} cipher ticks (equivalent to 2^{49} encryptions). Our second attack exploits a weakness in the key-update mechanism of the transponder. This attack recovers the secret key after 3×2^{16} authentication attempts with the transponder and negligible computational complexity. We have executed this attack in practice on several vehicles. We were able to recover the key and start the engine with a transponder emulating device. Executing this attack from beginning to end takes only 30 minutes. Our third attack exploits the fact that some car manufacturers set weak cryptographic keys in their vehicles. We propose a time-memory trade-off which recovers such a weak key after a few minutes of computation on a standard laptop.

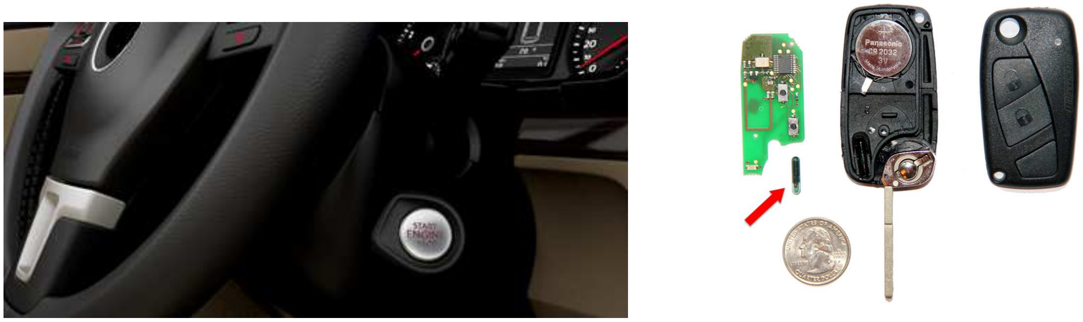
1 Introduction

Electronic vehicle immobilizers have been very effective at reducing car theft. Such an immobilizer is an electronic device that prevents the engine of the vehicle from starting when the corresponding transponder is not present. This transponder is a low-frequency RFID chip which is typically embedded in the vehicle's key. When the driver starts the vehicle, the car authenticates the transponder before starting the engine, thus preventing hot-wiring. In newer vehicles the mechanical ignition key has often been removed and replaced by a start button, see Figure 1(a). In such vehicles the immobilizer transponder is the only anti-theft mechanism that prevents a hijacker from driving away.

A distinction needs to be made between the vehicle immobilizer and the remotely operated central locking system. The latter is battery powered, operates at an ultra-high frequency (UHF), and only activates when the user pushes a button on the remote to (un)lock the doors of the vehicle. Figure 1(b) shows a disassembled car key where it is possible to see the passive Megamos Crypto transponder and also the battery powered remote of the central locking system.

The Megamos Crypto transponder is the first cryptographic immobilizer transponder manufactured by [19] and is currently one of the most widely used. The manufacturer claims to have sold more than 100 million immobilizer chips including Megamos Crypto transponders [22]. Figure 2 shows a list of vehicles that use or have used Megamos Crypto at least for some version/year. As it can be seen from this list, many Audi, Fiat, Honda, Volkswagen and Volvo cars used Megamos Crypto transponders at the time of this research (fall 2012).

The transponder uses a 96-bit secret key and a proprietary cipher in order to authenticate to the vehicle. Furthermore, a 32-bit PIN code is needed in order to be able to write on the memory of the transponder. The con-



(a) Keyless ignition with start button

(b) Megamos Crypto transponder in a car key

Figure 1: Megamos Crypto integration in vehicular systems

crete details regarding the cipher design and authentication protocol are kept secret by the manufacturer and little is currently known about them.

From our collaboration with the local police it was made clear to us that sometimes cars are being stolen and nobody can explain how. They strongly suspect the use of so-called ‘car diagnostic’ devices. Such a device uses all kind of custom and proprietary techniques to bypass the immobilizer and start a car without a genuine key. This motivated us to evaluate the security of vehicle immobilizer transponders. There are known attacks for three of the four widely used immobilizer transponders, namely DST40, Keeloq and Hitag2. Although, at the time of this research, little was known about the security of the Megamos Crypto transponder.

1.1 Our contribution

In this paper we have fully reverse-engineered all cryptographic mechanisms of Megamos Crypto which we publish here in full detail. For this we used IDA Pro[1] to decompile the software package that comes with the Tango Programmer[2].

Furthermore, we have identified several weaknesses in Megamos Crypto which we exploit in three attacks. Our first attack consists of a cryptanalysis of the cipher and the authentication protocol. Our second and third attack not only look at the cipher but also at the way in which it is implemented and poorly configured by the automotive industry.

Our first attack, which comprises *all* vehicles using Megamos Crypto, exploits the following weaknesses.

- The transponder lacks a pseudo-random number generator, which makes the authentication protocol vulnerable to replay attacks.

Make	Models
Alfa Romeo	147, 156, GT
Audi	A1, A2, A3, **A4 (2000)**, A6, A8, Allroad, Cabrio, Coupé, Q7, S2, S3, S4, S6, S8, **TT (2000)**
Buick	Regal
Cadillac	CTS-V, SRX
Chevrolet	Aveo, Kalos, Matiz, Nubira, Spark, Evanda, Tacuma
Citroën	**Jumper (2008)**, Relay
Daewoo	Kalos, Lanos, Leganza, Matiz, Nubira, Tacuma
DAF	CF, LF, XF
Ferrari	California, 612 Schaglietti
Fiat	Albea, Doblò, Idea, Mille, Multipla, Palio, **Punto (2002)**, Seicento, Siena, Stilo, **Ducato (2004)**
Holden	Barina, Frontera
Honda	Accord, Civic, CR-V, FR-V, HR-V, Insight, **Jazz (2002)**, Legend, Logo, S2000, Shuttle, Stream
Isuzu	Rodeo
Iveco	Eurocargo, Daily
Kia	Carnival, Clarus, Pride, Shuma, Sportage
Lancia	Lybra, Musa, Thesis, Y
Maserati	Quattroporte
Opel	Frontera
Pontiac	G3
Porsche	911, 968, Boxster
Seat	Altea, Córdoba, Ibiza, Leon, Toledo
Skoda	**Fabia (2011)**, Felicia, Octavia, Roomster, Super, Yeti
Ssangyong	Korando, Musso, Rexton
Tagaz	Road Partner
Volkswagen	Amarok, Beetle, Bora, Caddy, Crafter, Cross Golf, Dasher, Eos, Fox, Gol, **Golf (2006, 2008)**, Individual, Jetta, Multivan, New Beetle, Parati, Polo, Quantum, Rabbit, Saveiro, Santana, **Scirocco (2011)**, Touran, Tiguan, Voyage, **Passat (1998, 2005)**, Transporter
Volvo	C30, **S40 (2005)**, S60, S80, V50, V70, XC70, XC90, XC94

Figure 2: Vehicles that used Megamos Crypto for some version/year [39]. Boldface and year indicate specific vehicles we experimented with.

- The internal state of the cipher consists of only 56 bits, which is much smaller than the 96-bit secret key.
- The cipher state successor function can be inverted, given an internal state and the corresponding bit of cipher-text it is possible to compute the predecessor state.
- The last steps of the authentication protocol

[1] https://www.hex-rays.com/products/ida/

[2] http://www.scorpio-lk.com

provides and adversary with 15-bits of known-plaintext.

We present two versions of this attack. First we introduce a simple (but more computationally intensive) attack that recovers the secret key of the transponder with a computational complexity of 2^{56} encryptions. Then we optimize this attack, reducing its computational complexity to 2^{49} by using a time-memory trade-off. For this trade-off, a 12 terabyte lookup table needs to be pre-computed. This optimized version of the attack takes advantage of the fact that some of the cipher components can be run quite autonomously.

Our second attack exploits the following weaknesses.

- Currently, the memory of many Megamos Crypto transponders in the field is either unlocked or locked with a publicly known default PIN code [17]. This means that anybody has write access to the memory of the transponder. This also holds for the secret key bits.
- The 96-bit secret key is written to the transponder in blocks of 16 bits instead of being an atomic operation.

This attack recovers the 96-bit secret key of such a transponder within 30 minutes. This time is necessary to perform 3×2^{16} authentication attempts to the transponder and then recover the key with negligible computational complexity. We have executed this attack in practice and recovered the secret key of several cars from various makes and models. Having recovered the key we were able to emulate the transponder and start the vehicles.

Our third attack is based on the following observation. Many of the keys that we recovered using the previous attack had very low entropy and exhibit a well defined pattern, i.e., the first 32 bits of the key are all zeros. This attack consists of a time-memory trade-off that exploits this weakness to recover the secret key, within a few minutes, from two authentication traces. This attack requires storage of a 1.5 terabyte rainbow table.

We propose a simple but effective mitigating measure against our second attack. This only involves setting a few bits on the memory of the transponder and can be done by anyone (even the car owners themselves) with a compatible RFID reader.

Finally, we have developed an open source library for custom and proprietary RFID communication schemes that operate at an frequency of 125 kHz. We used this library to provide eavesdropping, emulation and reader support for Megamos Crypto transponders with the Proxmark III device[3]. The reader functionality allows the user to send simple commands like read and write to the transponder. In particular, this library can be used to set the memory lock bit and a random PIN code as a mitigation for our second attack, as described in Section 8.

1.2 Related work

In the last decades, semiconductor companies introduced several proprietary algorithms specifically for immobilizer security. Their security often depends on the secrecy of the algorithm. When their inner-workings are uncovered, it is often only a matter of weeks before the first attack is published. There are several examples in the literature that address the insecurity of proprietary algorithms. The most prominent ones are those breaking A5/1 [31], DECT [45, 47], GMR [18], WEP [24] and also many RFID systems like the MIFARE Classic [16, 26, 29, 46], CryptoRF [30] and iClass [27, 28].

Besides Megamos Crypto, there are only three other major immobilizer products being used. The DST transponder which was reverse-engineered and attacked by Bono et.al. in [9]; KeeLoq was first attacked by Bogdanov in [6] and later this attack was improved in [12, 36, 38]; Hitag2 was anonymously published in [60] and later attacked in [8, 13, 35, 52, 53, 57, 58].

With respect to vehicle security, Koscher et. al. attracted a lot of attention from the scientific community when they demonstrated how to compromise the board computer of a modern car [11, 40]. They were able to remotely exploit and control many car features such as tracking the car via GPS and adjust the speeding of the car. In 2011, Francillon et. al. [25] showed that with fairly standard equipment it is possible to mount a relay-attack on all keyless-entry systems that are currently deployed in modern cars.

The scientific community proposed several alternatives [43, 44, 59, 61, 62] to replace the weak proprietary ciphers and protocols. There are several commercial vehicle immobilizer transponders that makes use of standard cryptography, like AES [14]. Examples include the Hitag Pro transponder from NXP Semiconductors and ATA5795 transponder from Atmel. To the best of our knowledge, only Atmel made an open protocol design [1] and published it for scientific scrutiny. The security of their design was analyzed by Tillich et. al. in [54].

2 Technical background

This section briefly describes what a vehicle immobilizer is and how it is used by the automotive industry. Then we describe the hardware setup we use for our experiments. Finally we introduce the notation used throughout the paper.

2.1 Immobilizer

To prevent a hijacker from hot-wiring a vehicle, car manufacturers incorporated an electronic car immobilizer as

[3] http://www.proxmark.org/

an extra security mechanism. In some countries, having such an immobilizer is enforced by law. For example, according to European Commission directive (95/56/EC) it is mandatory that all cars sold in the EU from 1995 are fitted with an electronic immobilizer. Similar regulations apply to other countries like Australia, New Zealand (AS/NZS 4601:1999) and Canada (CAN/ULC S338-98). Although in the US it is not required by law, according to the independent organization Insurance Institute for Highway Safety (IIHS), 86 percent of all new passenger cars sold in the US had an engine immobilizer installed [55].

An electronic car immobilizer consists of three main components: a small transponder chip which is embedded in (the plastic part of) the car key, see Figure 1(b); an antenna coil which is located in the dashboard of the vehicle, typically around the ignition barrel; and the immobilizer unit that prevents the vehicle from starting the engine when the transponder is absent.

The immobilizer unit communicates through the antenna coil and enumerates all transponders that are in proximity of field. The transponder identifies itself and waits for further instructions. The immobilizer challenges the transponder and authenticates itself first. On a successful authentication of the immobilizer unit, the transponder sends back its own cryptographic response. Only when this response is correct, the immobilizer unit enables the engine to start.

The immobilizer unit is directly connected to the internal board computer of the car, also referred to as Electrical Control Unit (ECU). To prevent hot-wiring a car, the ECU blocks fuel-injection, disables spark-plugs and deactivates the ignition circuit if the transponder fails to authenticate.

2.2 Hardware setup

We used the Proxmark III to eavesdrop and communicate with the car and transponder. This is a generic RFID protocol analysis tool [56] that supports raw data sampling at a frequency of 125 kHz. We implemented a custom firmware and FPGA design that supports the modulation and encoding schemes of Megamos Crypto transponders. The design samples generic analog-digital converter (ADC) values and interpret them in real-time in the micro-controller. We have implemented commands to eavesdrop, read and emulate a transponder. Our library is able to decode field and transponder modulation simultaneously and is very precise in timing.

Figure 3: Proxmark 3

2.3 Notation

Throughout this paper we use the following mathematical notation. Let $\mathbb{F}_2 = \{\mathtt{0},\mathtt{1}\}$ be the set of Booleans. The symbol $\oplus$ denotes exclusive-or (XOR), $\mathtt{0}^n$ denotes a bitstring of n zero-bits. ε denotes the empty bitstring. Given two bitstrings x and y, xy denotes their concatenation. Sometimes we write this concatenation explicitly with $x \cdot y$ to improve readability. $\overline{x}$ denotes the bitwise complement of x. Given a bitstring $x \in \mathbb{F}_2^k$, we write x_i to denote the i-th bit of x. For example, given the bitstring $x = \mathtt{0x03} = \mathtt{00000011} \in \mathbb{F}_2^8$, $x_0 = 0$ and $x_6 = x_7 = 1$.

3 Megamos Crypto

This section describes Megamos Crypto in detail. We first describe the Megamos Crypto functionality, memory structure, and communication protocols, this comes from the product datasheet [21] and the application note [23]. Then we briefly describe how we reverse-engineered the cryptographic algorithms and protocols used in Megamos Crypto. Finally, we describe these algorithms and protocols in detail.

3.1 Memory

There are two types of Megamos Crypto transponders, in automotive industry often referred to as Magic I (V4070) [20] and Magic II (EM4170) [21]. The EM4170 transponder is the newer version and it has 16 memory blocks of 16-bit words. The contents of these memory blocks are depicted in Figure 4. The older version (V4070) supports exactly the same read and write operations and cryptographic algorithms, but it only has 10 memory blocks. The blocks 10 to 15, which store 64 bits of additional user memory and a 32-bit PIN code are simply not readable. The EM4170 transponder uses the same communication and is therefore backwards compatible with the V4070 transponder. Note that in some cars the new revision is deployed as replacement for the V4070 without making use of, or even initializing the additional user memory blocks and PIN code. The whole memory is divided in three sections with different access rights, see Figure 4.

The transponder identifier *id* is always read-only. The write access over the other memory blocks is determined by the value of the lock-bit l_0. Just as specified, the value of lock-bit l_1 does not have any influence the memory access conditions. Similarly, a successful or failed authentication has no effect on the access conditions.

- When $l_0 = 0$, all memory blocks (except *id*) of a Megamos Crypto transponder are still writable. The key k, PIN code *pin* are write-only and the user memory *um* blocks (which includes the lock-bits l) are read-write. However, after a successful write in block 1, the new value of l_0 determines the access condition for future write operations.

- When $l_0 = 1$, all writing is disabled. However, it does not affect the read access conditions. This means that the key k, PIN code pin can not be read out and the user memory um becomes read-only. Because the lock-bits l are stored in a user memory block they can always be read out.

The EM4170 allows to set the lock-bit l_0 back to zero using a PIN code pin. A valid PIN code resets the access conditions and enables again writing of k, pin, um and l. The PIN code has to be known or overwritten to the transponder before it is locked, otherwise an exhaustive search of the PIN code is required.

Block	Content	Denoted by
0	user memory	$um_0 \dots um_{15}$
1	user memory, lock bits	$um_{16} \dots um_{29} l_0 l_1$
2	device identification	$id_0 \dots id_{15}$
3	device identification	$id_{16} \dots id_{31}$
4	crypto key	$k_0 \dots k_{15}$
5	crypto key	$k_{16} \dots k_{31}$
6	crypto key	$k_{32} \dots k_{47}$
7	crypto key	$k_{48} \dots k_{63}$
8	crypto key	$k_{64} \dots k_{79}$
9	crypto key	$k_{80} \dots k_{95}$
10	pin code	$pin_0 \dots pin_{15}$
11	pin code	$pin_{16} \dots pin_{31}$
12	user memory	$um_{30} \dots um_{45}$
13	user memory	$um_{46} \dots um_{61}$
14	user memory	$um_{62} \dots um_{77}$
15	user memory	$um_{78} \dots um_{93}$

░ read-only
▒ write-only
□ read-write

Figure 4: Megamos Crypto transponder memory layout

3.2 Functionality and communication

The Megamos Crypto transponder supports four different operations: `read`, `write`, `reset` and `authenticate`.

- `read` operations are performed by three different commands, each returns multiple blocks. The transponder returns the concatenation of these blocks in one bitstring. The three available bitstrings are $id_{31} \dots id_0$, $l_1 l_0 um_{29} \dots um_0$ and $um_{93} \dots um_{30}$.
- `write` stores a 16-bit memory block in the memory of the transponder. The arguments for this command are the block number and the data. After receiving the command, the transponder stores the data in memory if the access conditions allow the requested write operation.
- `reset` takes the id and 32-bit PIN code as an argument. If the PIN code matches the value that is stored in pin, then the lock-bit l_0 is reset, see Section 3.1 for more details about l_0.
- `authenticate` takes three arguments. The first one is a 56-bit car nonce n_C. The second argument is a bitstring of 7 zero bits. The datasheet [21] refers to them as "divergency bits". It seems that these bit-periods are used to initialize the cipher. In Section 3.6 we show that the authentication protocol exactly skips 7 cipher steps before it starts generating output. The third argument is a 28-bit authenticator from the car a_C. If successful, the transponder responds with its 20-bit authenticator a_T.

When the driver turns on the ignition, several back-and-forward messages between the car and transponder are exchanged. It starts with the car reading out the transponder memory blocks that contains the identity, user memory and lock-bits. Next, the car tries to authenticate using the shared secret key k. If the authentication fails, the car retries around 20 times before it reports on the dashboard that the immobilizer failed to authenticate the transponder. Figure 5 shows an eavesdropped trace of a German car that initializes and authenticates a Megamos Crypto transponder.

To the best of our knowledge, there is no publicly available document that describes the structure of Megamos Crypto cipher. However, a simplified representation of the authentication protocol is presented in the EM4170 application note [23] as shown in Figure 6. It does not specify any details beyond the transmitted messages and the checks which the car and transponder must perform. The car authenticates by sending a nonce $n_C = Random$ and the corresponding authenticator $a_C = f(Rnd, K)$. When the car successfully authenticated itself, the Megamos Crypto transponder sends the transponder authenticator $a_T = g(Rnd, f, K)$ back to car.

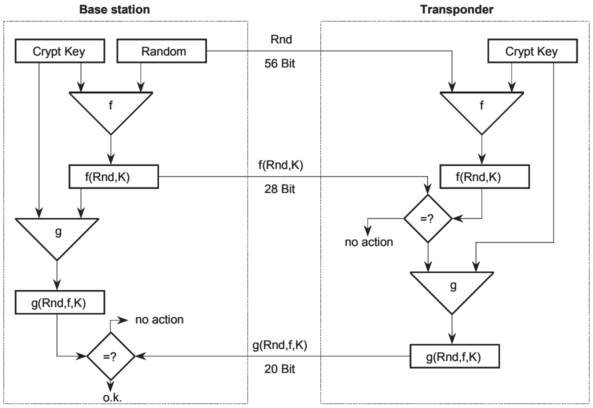


Figure 6: Authentication procedure excerpt from [23]

For communication the Megamos Crypto transponder uses a low frequency wave of 125 kHz and applies amplitude shift keying (ASK) modulation by putting a small resistance on the electro magnetic field. It utilizes a cus-

Origin	Message	Description
Car	`3`	Read identifier
Transponder	`A9 08 4D EC`	Identfier $id_{31} \dots id_0$
Car	`5`	Read user memory and lock-bits
Transponder	`80 00 95 13`	First user memory $l_1 l_0 um_{29} \dots um_0$
Car	`F`	Read large user memory (EM4170)
Transponder	`AA AA AA AA AA AA AA AA`	Second user memory $um_{93} \dots um_{30}$
Car	`6 | 3F FE 1F B6 CC 51 3F |` 0^7 `| F3 55 F1 A`	Authentication, $n_{C_{55}} \dots n_{C_0}$, 0^7, a_C
Transponder	`60 9D 6`	Car authenticated successful, send back a_T

Figure 5: Eavesdropped Megamos Crypto authentication using the 96-bit key `000000000000010405050905`. The structure of the secret key of the car suggests that it has an entropy of only 24 bits.

tom encoding scheme for status bits and a Manchester encoding scheme for transmitting data bits. The Megamos Crypto immobilizer unit signals the transponder to receive a command by dropping the field two consecutive times in a small time interval. Then it drops the field a few microseconds to modulate a zero and leaves the field on to modulate a one.

This way of modulation introduces the side-effect that the immobilizer unit and the transponder could get out-of-sync. When the immobilizer unit sends a bitstring of contiguous ones, there are no field drops for almost 15 milliseconds. The manufacturer realized this was a problem, but instead of proposing an alternative communication scheme they suggest to choose random numbers with more zeros's than ones and especially avoid sequential ones [23]. From a security perspective it sounds like a bad idea to suggest to system integrators that they should effectively drop entropy from the used random numbers.

To get a fair estimate of communication timings we did some experiments. With our hardware setup we were able to reach the highest communication speed with the transponder that is possible according to the datasheet. It allows us to read out the identifier *id* in less than 14 milliseconds and successfully authenticate within 34 milliseconds. These timings confirm that an adversary can wirelessly pickpocket the identifier and all its user memory in less than a second from a distance of one inch. Standing close to a victim for only a fraction of a second enables the adversary to gather the transponder identifier. When this identifier is emulated to the corresponding car, it is possible to gather partial authentication traces. Because the transponder lacks a random generator, this partial traces can later be used to retrieve the responses from the transponder which extends them to successful authentication traces. With a number of successful authentication traces it is possible to recover the secret key as described in Section 5.

3.3 Reverse-engineering the cipher

Recent articles point out the lack of security [11, 40, 41] in modern cars. The software in existing cars is designed with safety in mind, but is still immature in terms of security. Legacy protocols and technologies are often vulnerable to a number of remote and local exploits.

Most car keys need to be preprogrammed, which is also referred to as pre-coded, before they can be associated to a car. During this initialization phase the user memory blocks are filled with manufacturer specific data to prevent mixing of keys. This step adds no security, it just restricts the usage of keys that were meant a specific car make or model.

There are several car locksmith tools[4][5][6] in the after market that can initialize or change such transponder data. Such tools fully support the modulation/encoding schemes and communication protocol of the Megamos Crypto transponder. They implement some publicly available functionality like the `read`, `write` and `reset` commands. However, they do not implement the authentication protocol. To perform a successful authentication, knowledge of the Megamos Crypto cipher is necessary to compute the authentication messages a_C and a_T.

More advanced car diagnostic tools like AVDI[7] and Tango Programmer[8] offer functionality that goes beyond "legitimate" usage. These devices are able to dump the board-computer memory, recover the dealer code, and add a new blank transponder to the car. For this the tools do not require a genuine key to be present but they do need physical access to the can bus.

These diagnostic tools use the Megamos Crypto authentication functionality to speed up the process of adding new transponders to the car. For this, the tool needs the Megamos Crypto algorithm to compute valid

[4] http://www.istanbulanahtar.com
[5] http://www.advanced-diagnostics.co.uk
[6] http://www.jmausa.com
[7] http://www.abritus72.com
[8] http://www.scorpio-lk.com

authentication attempts. We would like to emphasize that non of these tools is able to recover the secret key of a transponder or perform any kind of cryptanalysis. In fact, within the legitimate automotive industry Megamos Crypto is believed to be unclonable.

The software package that comes with the Tango Programmer implements all cryptographic operations of the transponder including the Megamos Crypto cipher. We have analyzed the software thoroughly and extracted the algorithm from it.

Since the application implements several counter measures against reverse-engineering, this task was not trivial at all. It is highly protected with an executable obfuscator that runs a custom virtual machine, as described in [51], and a number of advanced anti-debugging tricks to avoid exposure of its inner workings. To perform our security evaluation of the Megamos Crypto cipher we bypassed all these measures and reverse engineered the cipher in a semi-automatic way by observing the memory state changes and guessing the intermediate cryptographic calculations.

Furthermore, we observed every Megamos Crypto related function call from the program instructions memory segment. When the program counter entered a suspicious memory segment, we invoked our clean-up routine that automatically grouped and dropped all unnecessary instructions (unconditional re-routings, sequential operations on the same variables, random non-influential calculations). After analysing this at run-time, the actual working of the algorithm was quickly deduced from the optimized and simplified persistent instruction set.

3.4 Cipher

This section describes the Megamos Crypto cipher in detail. The cipher consists of five main components: a Galois Linear Feedback Shift Register, a non-linear Feedback Shift Register, and three 7-bit registers. A schematic representation of the cipher is depicted in Figure 7.

Definition 3.1 (Cipher state)**.** *A Megamos Crypto* cipher state $s = \langle g,h,l,m,r \rangle$ *is an element of* $\mathbb{F}_2^{57}$ *consisting of the following five components:*

1. *the* Galois LFSR $g = (g_0 \dots g_{22}) \in \mathbb{F}_2^{23}$*;*
2. *the* non-linear FSR $h = (h_0 \dots h_{12}) \in \mathbb{F}_2^{13}$*;*
3. *the* first output register $l = (l_0 \dots l_6) \in \mathbb{F}_2^{7}$*;*
4. *the* second output register $m = (m_0 \dots m_6) \in \mathbb{F}_2^{7}$*;*
5. *the* third output register $r = (r_0 \dots r_6) \in \mathbb{F}_2^{7}$.

The following definitions describe the successor or feedback functions for each of these components.

Definition 3.2. *The successor function for the Galois linear feedback shift register* $G\colon \mathbb{F}_2^{23} \times \mathbb{F}_2 \times \mathbb{F}_2 \to \mathbb{F}_2^{23}$ *is defined as*

$$\begin{aligned} G(g_0 \dots g_{22}, i, j) = {} & (j \oplus g_{22}) g_0 g_1 g_2 (g_3 \oplus g_{22})(g_4 \oplus i) \\ & (g_5 \oplus g_{22})(g_6 \oplus g_{22}) g_7 \cdots g_{12} (g_{13} \oplus g_{22}) g_{14} g_{15} \\ & (g_{16} \oplus g_{22}) g_{17} \cdots g_{21} \end{aligned}$$

We also overload the function G *to multiple-bit input string* $G\colon \mathbb{F}_2^{23} \times \mathbb{F}_2 \times \mathbb{F}_2^{n+1} \to \mathbb{F}_2^{23}$ *as*

$$G(g, i, j_0 \dots j_n) = G(G(g, i, j_1 \dots j_n), i, j_0)$$

Definition 3.3. *The successor function for the non-linear feedback shift register* $H\colon \mathbb{F}_2^{13} \to \mathbb{F}_2^{13}$ *is defined as*

$$H(h_0 \dots h_{12}) = ((h_1 \wedge h_8) \oplus (h_9 \wedge h_{11}) \oplus \overline{h_{12}}) h_0 \dots h_{11}$$

Definition 3.4. *The feedback function for the first output register* $f_l\colon \mathbb{F}_2^{6} \to \mathbb{F}_2$ *is defined as*

$$\begin{aligned} f_l(x_0 \dots x_5) = {} & (\overline{x_0} \wedge \overline{x_2} \wedge x_3) \vee (x_2 \wedge x_4 \wedge \overline{x_5}) \vee \\ & (x_5 \wedge x_1 \wedge \overline{x_3}) \vee (x_0 \wedge \overline{x_1} \wedge \overline{x_4}). \end{aligned}$$

Definition 3.5. *The feedback function for the second output register* $f_m\colon \mathbb{F}_2^{6} \to \mathbb{F}_2$ *is defined as*

$$\begin{aligned} f_m(x_0 \dots x_5) = {} & (\overline{x_4} \wedge x_1 \wedge \overline{x_2}) \vee (x_5 \wedge \overline{x_1} \wedge x_3) \vee \\ & (x_0 \wedge x_2 \wedge \overline{x_3}) \vee (x_4 \wedge \overline{x_5} \wedge \overline{x_0}). \end{aligned}$$

Definition 3.6. *The feedback function for the third output register* $f_r\colon \mathbb{F}_2^{6} \to \mathbb{F}_2$ *is defined as*

$$\begin{aligned} f_r(x_0 \dots x_5) = {} & (x_5 \wedge \overline{x_0} \wedge \overline{x_2}) \vee (\overline{x_5} \wedge x_3 \wedge x_1) \vee \\ & (x_2 \wedge \overline{x_3} \wedge \overline{x_4}) \vee (x_0 \wedge x_4 \wedge \overline{x_1}). \end{aligned}$$

With every clock tick the cipher steps to its successor state and it (potentially) outputs one bit of keystream. The following precisely defines the successor state and the output of the cipher.

Definition 3.7 (Successor state)**.** *Let* $s = \langle g,h,l,m,r \rangle$ *be a cipher state and* $i \in \mathbb{F}_2$ *be an input bit. Then, the* successor cipher state $s' = \langle g',h',l',m',r' \rangle$ *is defined as*

$$\begin{aligned} g' &:= G(g, i, l_1 \oplus m_6 \oplus h_2 \oplus h_8 \oplus h_{12}) \\ h' &:= H(h) \\ l' &:= a l_0 \dots l_5 \\ m' &:= b m_0 \dots m_5 \\ r' &:= c r_0 \dots r_5 \end{aligned}$$

where

$$\begin{aligned} a &= f_l(g_0 g_4 g_6 g_{13} g_{18} h_3) \oplus g_{22} \oplus r_2 \oplus r_6 \\ b &= f_m(g_1 g_5 g_{10} g_{15} h_0 h_7) \oplus l_0 \oplus l_3 \oplus l_6 \\ c &= f_r(g_2 (g_3 \oplus i) g_9 g_{14} g_{16} h_1) \oplus m_0 \oplus m_3 \oplus m_6 \end{aligned}$$

We define the successor function suc: $\mathbb{F}_2^{57} \times \mathbb{F}_2 \to \mathbb{F}_2^{57}$ *which takes a state* s *and an input* $i \in \mathbb{F}_2$ *and outputs the successor state* s'*. We overload the function* suc *on multiple-bit input which takes a state* s *and an input* $i \in \mathbb{F}_2^{n+1}$ *as*

$$\begin{aligned} \mathrm{suc}(s, i_0 \dots i_n) &= \mathrm{suc}(s', i_n) \\ \textit{where } s' &= \mathrm{suc}(s, i_0 \dots i_{n-1}) \end{aligned}$$

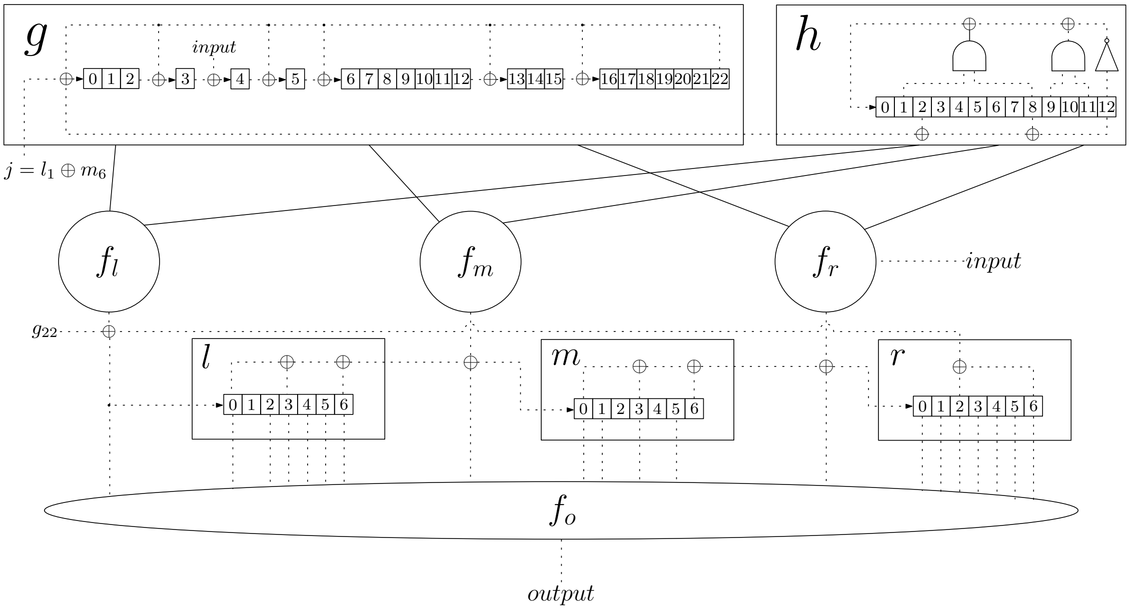


Figure 7: Schematic representation of the cipher

Definition 3.8. *The non-linear output filter function $f_o \colon \mathbb{F}_2^{20} \to \mathbb{F}_2$ has been deliberately omitted in this paper.*

Definition 3.9 (Output)**.** *Define the function output: $\mathbb{F}_2^{57} \times \mathbb{F}_2 \to \mathbb{F}_2$ which takes as input an internal state $s = \langle g,h,l,m,r \rangle$ and an input $i \in \mathbb{F}_2$ and returns the bit*

$$f_o(abcl_0l_2l_3l_4l_5l_6m_0m_1m_3m_5r_0r_1r_2r_3r_4r_5r_6)$$

where

$$a = f_l(g_0g_4g_6g_{13}g_{18}h_3) \oplus g_{22} \oplus r_2 \oplus r_6$$
$$b = f_m(g_1g_5g_{10}g_{15}h_0h_7) \oplus l_0 \oplus l_3 \oplus l_6$$
$$c = f_r(g_2(g_3 \oplus i)g_9g_{14}g_{16}h_1) \oplus m_0 \oplus m_3 \oplus m_6$$

We also overload the function output on multiple-bit input which takes a state s and an input $i \in \mathbb{F}_2^{n+1}$ as

$$output(s, i_0 \dots i_n) = output(s, i_0) \cdot output(s', i_1 \dots i_n)$$
$$\text{where } s' = \mathrm{suc}(s, i_0).$$

3.5 Cipher initialization

The following sequence of definitions describe how the cipher is initialized.

Definition 3.10. *Let init: $\mathbb{F}_2^{23} \times \mathbb{F}_2^{n+1} \to \mathbb{F}_2^{n+24}$ be defined as*

$$init(g, \varepsilon) := g$$
$$init(g, x_0 \dots x_n) := init(G(g, 0, x_n), x_0 \dots x_{n-1}) \cdot g_{22}$$

Definition 3.11. *Let $p \in \mathbb{F}_2^{56}, q \in \mathbb{F}_2^{44}$ and $t \in \mathbb{F}_2^{43}$ be defined as*

$$p := n_{C_0} \dots n_{C_{55}} + k_{40} \dots k_{95} \mod 2^{56}$$
$$q := (p_2 \dots p_{45}) \oplus (p_8 \dots p_{51}) \oplus (p_{12} \dots p_{55})$$
$$t := init(q_{20} \dots q_{42}, q_0 \dots q_{19})$$

Then, the initial cipher state *$s_0 = \langle g,h,l,m,r \rangle$ is defined as*

$$g := t_0 \dots t_{22}$$
$$h := 0p_0 \dots p_{11}$$
$$l := t_{23} \dots t_{29}$$
$$m := t_{30} \dots t_{36}$$
$$r := t_{37} \dots t_{42}q_{43}$$

3.6 Authentication protocol

This section describes the authentication protocol between a Megamos Crypto transponder and the vehicle immobilizer. This protocol is depicted in Figure 8. An annotated example trace is shown in Figure 5.

Definition 3.12. *Given a key $k = k_0 \dots k_{95} \in \mathbb{F}_2^{96}$ and an initial state s_0 as defined in Definition 3.11, the internal state of the cipher at step i is defined as*

$$s_i := \mathrm{suc}(s_{i-1}, k_{40-i}) \qquad \forall i \in [1 \dots 40]$$
$$s_{i+41} := \mathrm{suc}(s_{i+40}, 0) \qquad \forall i \in \mathbb{N}$$

During authentication, the immobilizer starts by sending an authenticate command to the transponder. This command includes a 56-bit nonce n_C and the 28 bits a_C output by the cipher from state s_7. Then, the transponder responds with the next 20 output bits a_T, i.e., produced from state s_{35}.

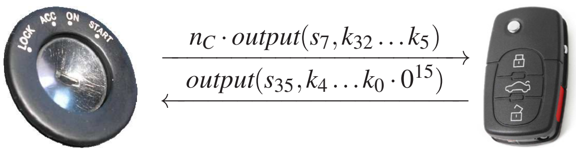

LOCK
ACC
ON
START
$n_C \cdot output(s_7, k_{32} \dots k_5)$
$output(s_{35}, k_4 \dots k_0 \cdot 0^{15})$

Figure 8: Megamos Crypto authentication protocol

4 Cipher Properties

This section describes several properties of the Megamos Crypto cipher which will be later used in the attacks.

4.1 Rollback

Given a cipher state it is possible to recover its previous state when this exists. Rolling-back the cipher is non-trivial due to the non-linear operations in the suc function. Next we describe precisely how to rollback the cipher to recover a predecessor state.

We start by rolling-back registers g and h. This computation is straightforward as described in the following definitions.

Definition 4.1. *The predecessor function for the non-linear feedback shift register $H^{-1}\colon \mathbb{F}_2^{13} \to \mathbb{F}_2^{13}$ is defined as*

$$H^{-1}(h_0 \dots h_{12}) = h_1 \dots h_{11}((h_2 \wedge h_9) \oplus (h_{10} \wedge h_{12}) \oplus \overline{h_0})$$

Definition 4.2. *The predecessor function for the Galois linear feedback shift register $G^{-1}\colon \mathbb{F}_2^{23} \times \mathbb{F}_2 \times \mathbb{F}_2 \to \mathbb{F}_2^{23}$ is defined as*

$$G^{-1}(g_0 \dots g_{22}, i, j) = g_1 g_2 (g_3 \oplus b)(g_4 \oplus i)(g_5 \oplus b)(g_6 \oplus b)$$
$$g_7 \dots g_{12}(g_{13} \oplus b) g_{14} g_{15} (g_{16} \oplus b) g_{17} \dots g_{22} b$$

where $b = g_0 \oplus j$

Next we describe how to rollback registers l, m and r. A difficulty in doing that arises from the fact that m_6 in the predecessor state is not determined. To circumvent this issue, we need to first guess the bit m_6 and then check whether this guess is consistent with the rest of the state. For 18.75% of the states this condition is not met for neither $m_6 = 0$ nor $m_6 = 1$, which means that the state has no predecessor. For 62.5% of the states there is only one value of m_6 satisfying this condition, which means that they have only one predecessor state. Finally, 18.75% of the states have two possible predecessor states, one for $m_6 = 0$ and one for $m_6 = 1$. In this case both states have to be considered as potentially being the predecessor state. Given the fact that the average probability of having two predecessors equals the probability of having none the list of candidate predecessor states remains of a constant size.

A precise description of how to compute a predecessor state follows.

Definition 4.3 (Predecessor state)**.** *Let $s' = \langle g', h', l', m', r' \rangle$ be a cipher state and $i \in \mathbb{F}_2$ be an input bit. Then, $s = \langle g, h, l, m, r \rangle$ is a* predecessor cipher state *of s' if it satisfies*

$$h = H^{-1}(h')$$
$$g = G^{-1}(g', i, h_{12} \oplus h_8 \oplus h_2 \oplus l'_2 \oplus m_6)$$
$$l = l'_1 \dots l'_6 (m'_0 \oplus f_m(g_1 g_5 g_{10} g_{15} h_0 h_7 h) \oplus l'_4 \oplus l'_1)$$
$$m_6 = r'_0 \oplus f_r(g_2 (g_3 \oplus i) g_9 g_{14} g_{16} h_1) \oplus m'_4 \oplus m'_1$$
$$m = m'_1 \dots m'_6 m_6$$
$$r = r'_1 \dots r'_6 (l'_0 \oplus f_l(g_0 g_4 g_6 g_{13} g_{18} h_3) \oplus r'_3 \oplus g_{22}).$$

4.2 Undoing cipher initialization

In this section we show that the cipher initialization procedure can be reverted. This means that given an initial state it is possible to recover the part of the secret key that was used for initialization. The following describes exactly how this can be achieved. We first introduce some auxiliary functions.

Definition 4.4. *Let $init^{-1}\colon \mathbb{F}_2^{23} \times \mathbb{F}_2^{n+1} \to \mathbb{F}_2^{n+1}$ be defined as*

$$init^{-1}(g, x_0) := g$$
$$init^{-1}(g, x_0 \dots x_n) := b \cdot init^{-1}(G^{-1}(g, 0, b), x_1 \dots x_n)$$
$$\text{where } b = g_0 \oplus x_0$$

Definition 4.5. *Let $Q^{-1}\colon \mathbb{F}_2^{n+12} \to \mathbb{F}_2^n$ be defined as*

$$Q^{-1}(p_0 \dots p_{11}) := \varepsilon$$
$$Q^{-1}(p_0 \dots p_n) := (p_2 \oplus p_8 \oplus p_{12}) \cdot Q^{-1}(p_1 \dots p_n)$$

Proposition 4.6. *Given an initial state $s_0 = \langle g, h, l, m, r \rangle$ it is possible to compute secret key bits $k_{40} \dots k_{95}$.*

The computation of the key bits is as follows.

$$t := g \cdot l \cdot m \cdot r$$
$$q := init^{-1}(t_0 \dots t_{22}, t_{23} \dots t_{42}) \cdot t_{43}$$
$$p := h \cdot Q^{-1}(h \cdot q)$$
$$k_{40} \dots k_{95} := p - n_C \mod 2^{56}.$$

4.3 Entropy of the non-linear feedback shift register

First and foremost, the initialization of the 13-bit non-linear feedback shift register (NLFSR) h is far from ideal. The NLFSR is initialized with only 12 bits by an almost linear function of the random nonce and the secret key. Adding upon the fact that, naturally, as the NLFSR h is not affected by other registers and the input, it is trivial to compute all successor states for a given h. Therefore, the search space for the 13-bit h register drops down to 2^{12}. Moreover, careful observation of the n_C value on the communication channel can leak information on whether the same value has been previously used for initializing h. For instance if the first 13 bits of n_C is the same for two different authentication attempts, depending on the rest of the bits, the attacker can conclude with a certain confidence that the same state is used for initializing h. This weakness can be later exploited in a differential attack.

5 Cryptanalysis of Megamos Crypto

This section describes a cryptanalysis of the Megamos Crypto cipher. We first introduce a simple cryptanalysis which is easier-to-grasp and recovers the 96-bit secret key with a computational complexity of 2^{56}. Then, in Section 5.1 we reduce its computational complexity down to 2^{48}.

This analysis requires two successful authentication traces $T = \langle n_C, output(s_7, k_{32} \ldots k_5), output(s_{35}, k_4 \ldots k_0 \cdot 0^{15})\rangle$ and $T' = \langle n'_C, output(s'_7, k_{32} \ldots k_5), output(s'_{35}, k_4 \ldots k_0 \cdot 0^{15})\rangle$. Discarding from all internal states $s_{40} \in \mathbb{F}_2^{56}$ those guesses which produce different 15 output bits than the trace T which leaves $2^{56-15} = 2^{41}$ candidate states for s_{40}. Rolling the cipher backwards for each candidate up to state s_7, as shown in Section 4.1, leaves—on average—the same number of candidate states for s_7, namely 2^{41}. Each step requires guessing one input bit k_i but at the same time the output provides one bit of information. Note that this determines a guess for key bits $k_0 \ldots k_{32}$. Rolling further the cipher backwards up to state s_0 requires guessing of $k_{33} \ldots k_{39}$ while no output bits are produced. This brings the number of candidate states for s_0 to $2^{41+7} = 2^{48}$. For each candidate s_0, the remaining key bits $k_{40} \ldots k_{95}$ can be recovered by undoing the initialization of the cipher as described in Section 4.2. This produces 2^{48} candidate keys $k_0 \ldots k_{95}$. On average, there is only one candidate secret key $k_0 \ldots k_{95}$ that together with n'_C produces the trace T'. This is because there are only 2^{48} candidates keys and 48 bits of information on the trace.

Time complexity on average, the aforementioned algorithm has a computational complexity of approximately 2^{56} encryptions. We have simulated an FPGA implementation of the algorithm on a Xilinx ISE 10.1 for synthesis and place & route. The results show that our implementation of a Megamos Crypto core covers approximately 1% of the Xilinx Spartan 3-1000 FPGA, the exact same chip that is employed in the COPACOBANA [42]. The maximum frequency that the core can run at is 160.33 MHz, which means we can test a single bit output in 6.237ns. Given this performance and area figures, a rough estimation suggests we can fit at least 50 Megamos Crypto cores in a Spartan 3-1000 FPGA. Considering that there are 120 such FPGA in a COPACOBANA, and since we can run them at 160.33MHz, we can run approximately $2^{39.8}$ tests per second. After every cycle, half of the candidate states are discarded, which means that a search takes less than two days on a COPACOBANA.

5.1 Reducing the computational complexity

Most of the computational complexity of the cryptanalysis described in Section 5 comes from iterating over all 2^{56} internal states s_{40}. In the following analysis we lower this complexity to 2^{48} by splitting the cipher state into two and using a time-memory trade-off. The main idea behind this optimization is to exploit the fact that components g and h are quite independent from components l, m and r. In fact, at each cipher step, there is only one bit of information from l, m, r which affects g, h, namely $l_1 \oplus m_6$. Conversely, there are only three bits of information from g, h that have an influence on components l, m, r.

In order reduce the complexity of the cryptanalysis an adversary $\mathscr{A}$ proceeds as follows.

1. Pre-computation: only once, and for each 2^{12} possible values of h, the adversary computes a table T_h as follows. For each $g \in \mathbb{F}_2^{23}$ and $j \in \mathbb{F}_2^{8}$ the adversary runs cipher components g and h one step forward. For this, $\mathscr{A}$ uses j_0 as a guess for $l_1 \oplus m_6$. At this stage $\mathscr{A}$ computes $f_0 := f_l(\cdot) f_m(\cdot) f_r(\cdot)$. From the resulting g and h, $\mathscr{A}$ repeats this procedure another 7 times, using j_i as a guess at step i and computing a three bit value f_i. At the end, she creates an entry in the table T_h of the form $< f_0 \ldots f_7, j_0 \ldots j_7, g >$. When the table is completed $\mathscr{A}$ sorts the table (on f, j).
2. As before $\mathscr{A}$ first eavesdrops one authentication trace between a legitimate transponder and an immobilizer. Thus $\mathscr{A}$ learns n_C, $output(s_7, k_{32} \ldots k_5)$ and $output(s_{35}, k_4 \ldots k_0 \cdot 0^{15})$.
3. Choose h.
4. Next the adversary will try to recover state s_{40}. For each $l, m, r \in \mathbb{F}_2^7$ the adversary runs these components 8 steps forward. At each step i she needs to guess 3 bits $f_i := f_l(\cdot) f_m(\cdot) f_r(\cdot)$ but she will be able to immediately discard half of these guesses as they will not produce the correct output bit $output(s_{40+i}, 0)$. At each step $\mathscr{A}$ will also compute $j_i : l_1 \oplus m_6$. At the end $\mathscr{A}$ has $2^{21+16} = 2^{37}$ bitstrings of the form $< f_0 \ldots f_7, j_0 \ldots j_7, l, m, r >$.
5. For each of these bitstrings $\mathscr{A}$ performs a lookup on $f_0 \ldots f_7, j_0 \ldots j_7$ in the table T_h and recovers g. On average, half of these lookups will not have a match in T_h. In that case the candidate state is discarded, leaving only 2^{36} full candidate states.
6. Each of these candidate states are then rolled forward another 7 steps. Only $2^{36-7} = 2^{29}$ of these states will produce the correct $output(s_{48}, 0^7)$ bits and the rest are discarded.

7. For each of these 2^{29} states the adversary proceeds as in Section 5, undoing the initialization and checking against a second trace.

Time and resource complexity

- Pre-computation: for building the tables T_h the adversary needs to run components g and h of the cipher 8 steps. This has a computational complexity of $2^{23+12+3} = 2^{38}$ cipher steps. The generated tables can be conveniently stored in memory using a structure for compression like $/n/f_0/f_1/\dots/f_7/j/$`g.dat`. Storing all these tables require 12 terabyte of memory.
- As before, this cryptanalysis requires two successful authentication traces to recover the secret key. The most time intensive operation of this analysis is performing the 2^{37} lookups in the table for each of the 2^{12} values of h, i.e., 2^{49} table lookups.

The time-memory trade-off proposed in this section requires many indirect memory lookups and is therefore difficult to mount in practice with ordinary consumer hardware.

6 Partial Key-Update Attack

As it was described in Section 3.2, when the transponder is not locked, the Megamos Crypto transponder does not require authentication in order to write to memory. This makes it vulnerable to a trivial denial of service attack. An adversary just needs to flip one bit of the secret key of the transponder to disable it.

Besides this obvious weakness, there is another weakness regarding the way in which the secret key is written to the transponder. The secret key of Megamos Crypto is 96 bits long. As described in Section 3.1, these 96 bits are stored in 6 memory blocks of 16 bits each (blocks 4 to 9), see Figure 4. It is only possible to write one block at a time to the transponder. This constitutes a serious weakness since a secure key-update must be an atomic operation.

Next, we mount an attack which exploits this weakness to recover the secret key. For this attack we assume that an adversary $\mathscr{A}$ is able communicate with the car and transponder. She proceeds as follows.

1. The adversary first eavesdrops a successful authentication trace, obtaining n_C and a_C from the car.
2. Then, for $k = 0$ to $2^{16} - 1$ the adversary writes k on memory block 4 of the transponder, where key bits $k_0 \dots k_{15}$ are stored. After each write command $\mathscr{A}$ initiates an authentication attempt with the transponder, replaying n_C and a_C (remember that the transponder does not challenge the car). For one value of k the transponder will accept a_C and give an answer. Then $\mathscr{A}$ knows that $k_0 \dots k_{15} = k$.
3. The adversary proceeds similarly for blocks $5 \dots 9$ thus recovering the complete secret key.

Attack complexity this attack requires 6×2^{16} key-updates and the same amount of authentication attempts. This takes approximately 25 minutes for each block which adds up to a total of two and a half hours.

6.1 Optimizing the attack

The above attack is very powerful, in the worst case, the attacker needs to update the key on the transponder and make an authentication attempt 2^{16} times. However, the same attack can be applied with only one key-update and 2^{16} authentication attempts, by choosing carefully the value of n_C. The optimized attack can be mounted as follows:

1. As before, the adversary first eavesdrops a successful authentication trace, obtaining n_C, a_C and a_T.
2. then, she writes `0x0000` on memory block 9 which contains key bits $k_{80} \dots k_{95}$.
3. The adversary then increments the observed n_C value and attempts an authentication for each
$$n_C + inc \pmod{2^{56}}, \text{ where } 0 \leq inc < 2^{16}.$$
4. Repeating step 3) at most 2^{16} times, the transponder will accept one a_C value for a particular increment value inc and give an answer. Then $\mathscr{A}$ knows that $k_{80} \dots k_{95} = inc$.
5. The adversary proceeds similarly for blocks 8 and 7. At this point the adversary has recovered key bits $k_{48} \dots k_{95}$.
6. Next, the adversary guesses 15 key bits $k_{33} \dots k_{47}$.
7. Having $k_{33} \dots k_{95}$ the adversary is now able to initialize the cipher, obtain the initial state s_0 and run it forward up to state s_7. At this point the adversary has 2^{15} candidates for state s_7.
8. For each of these candidates, she runs the cipher forward 33 steps up to state s_{40}. While running the cipher forward the adversary is able to determine input bits $k_{32} \dots k_0$ by comparing the output bits to a_C and a_T from the trace.
9. Then, forward each candidate state at s_{40} to s_{55} and produce another 15 output bits to test on, although this time, with the known input of 15 zero bits. On average only one candidate survives this test. The adversary has now recovered the complete key.

Attack complexity This attack requires only one successful authentication trace. In total, we need to write three times on the memory of the transponder and perform 3×2^{16} authentications with the transponder. This can be done within 30 minutes using a Proxmark III. The computational complexity of the last three steps is 2^{15} encryptions which takes less than a second on a laptop.

7 Weak-Key Attack

During our experiments we executed the previous attack on several cars of different make and model. Many of the keys we recovered were of the form $k_0 = \cdots = k_{31} = 0$ and more or less random looking bits for $k_{32} \dots k_{96}$ (although we have found keys where only ten of the 96 bits were ones). In the remainder of this paper we call such a key *weak*. Figure 9 shows some examples of weak keys we found during our experiments (on the vehicles indicated in Figure 2). To avoid naming concrete car models we use $A, B, C \dots$ to represent car makes. We write numbers $X.1, X.2, X.3 \dots$ to represent different car models of make X.

Car	Secret key
$A.1$	00000000**d**8 **b**3**9**6**7**c**5**a**3**c**3**b**2**9
$A.2$	00000000**d**9 **b**7**9**d**7**a**5**b**3**c**3**b**2**8
$B.1$	00000000**0**0 **0**0**0**1**0**4**0**5**0**5**0**9**0**5

Figure 9: Recovered keys from our own cars. Besides the evident 32 leading zero bits, every second nibble seems to encode a manufacturer dependant value, which further reduces the entropy of the key.

Apparently, the automotive industry has decided to use only 64 bits of the secret key, probably due to compatibility issues with legacy immobilizer systems. If a Megamos Crypto transponder uses such a weak key it is possible to recover this key quickly, even when the memory of the transponder is locked with a PIN code. To be concrete, a weak secret key with the bits $k_0 \dots k_{31}$ fixed by the car manufacturer allows an adversary know the input bits of the cipher states $s_8 \dots s_{55}$.

With known input to the cipher at states $s_8 \dots s_{55}$, it is possible to pre-compute and sort on a 47 contiguous output bits for each internal state at s_8. However, such a table with 2^{56} entries requires a huge amount of storage. There are many time-memory tradeoff methods proposed in the literature over the last decades [2–5, 10, 33, 34, 49]. For example, a rainbow table shrinks the storage significantly, while requiring only a modest amount of computation for a lookup.

Concretely, in order to mount such an attack, an adversary $\mathscr{A}$ proceeds as follows.

1. Pre-computation: only once, the adversary computes the following rainbow table. First, she chooses n random permutations $R_0 \dots R_{n-1}$ of $\mathbb{F}_2^{56} \to \mathbb{F}_2^{56}$ which she uses as reduction functions (colors). To compute a chain, the intermediate states are generated by

$$s_{i+1} = R_j(output(s_i, 0^{56})).$$

The chain begins with the first reduction function R_0. When a distinguished point (i.e., a state with a specific pattern like a prefix of z zero bits) is reached then the next reduction function R_{j+1} is used, in order to prevent chain merges. The chain is completed once a distinguished point is reached while using the last reduction function R_{n-1}, see Figure 10. The start and end values of each chain are stored in the rainbow table which is sorted on end values.

$$\boxed{\begin{array}{ccccc} sp_0 \overset{R_0(\cdot)}{\leadsto} d_1 \overset{R_1(\cdot)}{\leadsto} & \dots & \overset{R_{n-2}(\cdot)}{\leadsto} d_{n-1} \overset{R_{n-1}(\cdot)}{\leadsto} d_n = ep_0 \\ \vdots & & \vdots \\ sp_m \overset{R_0(\cdot)}{\leadsto} d'_1 \overset{R_1(\cdot)}{\leadsto} & \dots & \overset{R_{n-2}(\cdot)}{\leadsto} d'_{n-1} \overset{R_{n-1}(\cdot)}{\leadsto} d'_n = ep_m \end{array}}$$

Figure 10: Construction of the rainbow table

2. As before $\mathscr{A}$ first eavesdrops one authentication trace between a legitimate transponder and an immobilizer. Thus $\mathscr{A}$ learns the car nonce n_C and 47 output bits $o_0 \dots o_{46} = output(s_7, k_{32} \dots k_5) \cdot output(s_{35}, k_4 \dots k_0 \cdot 0^{15})$.
3. For each value of $u_0 \dots u_8 \in \mathbb{F}_2^9$ and each reduction function R_j the adversary looks up $o_0 \dots o_{46} u_0 \dots u_8$ in the Rainbow table. In order to look up such a value she sets it as a state and runs the chain until the last distinguished point is reached at the last reduction function. Then, it performs n lookups in the rainbow table (one for each reduction function) to find the corresponding end point ep.
4. When the end point is found then the corresponding start point sp is used to find the previous internal state which generates $o_0 \dots o_{46} u_0 \dots u_8$ as output. Since we guessed the last 9 bits $u_0 \dots u_8$, we should consider this as a candidate state.
5. Then, the adversary rolls back each of those states seven steps, guessing the input $k_{32} \dots k_{39}$. This produces 2^8 candidate states for s_0. As before, for each candidate s_0 she undoes the initialization of the cipher and recovers the remaining key bits $k_{40} \dots k_{95}$. These need to be tested with another trace.
6. If the test is passed then we have recovered the secret key. Otherwise the next $u_0 \dots u_8$ should be considered at step 3).

Attack complexity This attack requires two successful authentication traces. This attack allows for a trade-off between memory and computational complexity. The longer the chains the smaller the table gets but more computation is needed for each lookup. Just to give an impression of the feasibility of the attack we consider the following configuration. Take z to be 10 bits, therefore our distinguished states have 10 zero bits followed by other 46 bits. We also take $64 = 2^6$ random permutations $R_0 \ldots R_{63}$. Then, following the computations of Oechslin [49], we get that the size of the rainbow table is $(2^{56})^{\frac{2}{3}} \approx 2^{37}$ entries of 12 bytes which is 1.5 TB. Regarding its computational complexity, we need to compute at step 3) 2^9 candidates for which we compute, for all 2^6 reduction functions and for all offsets, the end point. Since a chain has length at most 2^{16}, this takes at most $2^9 \times 2^6 \times 2^6 \times 2^{16} = 2^{37}$ encryptions. This can be computed within a few minutes on a laptop.

For building the rainbow table (needed only once), computation of the chains is sped up considerably by using FPGAs. Recently Kalenderi et. al. showed in [37] that a single FPGA (similar to the ones used in the COPACOBANA) computes chains 2824 times faster than a single 3GHz processor. They computed rainbow tables for the A5/1 cipher, which is reasonably similar to the Megamos Crypto cipher. Although the internal state of A5/1 is with 64 bits considerably larger than the 56 bits of Megamos Crypto, they are both designed for hardware implementation and both embed a non-linear component that causes some internal states to merge. Their experimental setup generates 345 chains for A5/1 in 830 milliseconds, which is roughly $\frac{345}{0.830} \approx 415$ chains per second. If we compute an estimate with respect to the difference in complexity, the COPACOBANA with a 120 FPGA-array can compute $415 \times (2^8)^{\frac{1}{3}} \times 120 \approx 2^{18.3}$ chains per second. That means it takes only $2^{37-18.3} \approx 2^{18.7}$ seconds, which is less than 5 days, to build the complete rainbow table.

8 Practical considerations and mitigation

Our attacks require close range wireless communication with both the immobilizer unit and the transponder. It is not hard to imagine real-life situations like valet parking or car rental where an adversary has access to both for a period of time. It is also possible to foresee a setup with two perpetrators, one interacting with the car and one wirelessly pickpocketing the car key from the victims pocket.

As mitigating measure, car manufacturers should set uniformly generated secret keys and for the devices which are not locked yet, set PIN codes and write-lock their memory after initialization. This obvious measures would prevent a denial of service attack, our partial key-update attack from Section 6 and our weak-key attack from Section 7.

Car owners can protect their own vehicles against a denial of service and the partial key-update attack, described in Section 6. These attacks only work if the adversary has write access to the memory of the transponder, which means that the lock-bit l_0 is set to zero. It is possible for a user to test for this property with any compatible RFID reader, like the Proxmark III, using our communication library. If $l_0 = 0$, then you should set the lock-bit l_0 to one. It is possible to set this bit without knowing the secret key or the PIN code. When dealing with the more recent version of the Megamos Crypto transponder (EM4170), users should also update the PIN code to a random bit-string before locking the transponder.

On the positive side, our first (cryptographic) attack is more computationally intensive than the attacks from Section 6 and 7 which makes it important to take the aforementioned mitigating measures in order to prevent the more inexpensive attacks. Unfortunately, our first attack is also hard to mitigate when the adversary has access to the car and the transponder (e.g., Valet and car rental). It seems infeasible to prevent an adversary from gathering two authentication traces. Furthermore, this attack exploits weaknesses in the core of the cipher's design (e.g., the size of the internal state). It would require a complete redesign of the cipher to fix these weaknesses. To that purpose, lightweight ciphers like Grain [32], Present [7] and KATAN [15] have been proposed in the literature and could be considered as suitable replacements for Megamos Crypto. Also, immobilizer products implementing AES are currently available in the market.

9 Conclusions

The implications of the attacks presented in this paper are especially serious for those vehicles with keyless ignition. At some point the mechanical key was removed from the vehicle but the cryptographic mechanisms were not strengthened to compensate.

We want to emphasize that it is important for the automotive industry to migrate from weak proprietary ciphers like this to community-reviewed ciphers such as AES [14] and use it according to the guidelines. For a few years already, there are contactless smart cards on the market [48, 50] which implement AES and have a fairly good pseudo-random number generator. It is surprising that the automotive industry is reluctant to migrate to such transponders considering the cost difference of a better chip ($\leq$ 1 USD) in relation to the prices of high-end car models ($\geq 50{,}000$ USD). Since most car keys are actually fairly big, the transponder design does not really have to comply with the (legacy) constraints of minimal size.

Following the principle of responsible disclosure, we have notified the manufacturer of our findings back in November 2012. Since then we have an open communication channel with them. We understand that measures have been taken to prevent the weak-key and partial key-update attacks when the transponder was improperly configured.

10 Acknowledgments

The authors would like to thank Bart Jacobs for his firm support.

References

[1] Embedded avr microcontroller including rf transmitter and immobilizer lf functionality for remote keyless entry - ATA5795C. Product Datasheet, November 2011. Atmel Corporation.

[2] AVOINE, G., JUNOD, P., AND OECHSLIN, P. Characterization and improvement of time-memory trade-off based on perfect tables. *ACM Transactions on Information and System Security (TISSEC 2008) 11*, 4 (2008), 1–22.

[3] BABBAGE, S. A space/time tradeoff in exhaustive search attacks on stream ciphers. In *European Convention on Security and Detection* (1995), vol. 408 of *Conference Publications*, IEEE Computer Society, pp. 161–166.

[4] BIRYUKOV, A., MUKHOPADHYAY, S., AND SARKAR, P. Improved time-memory trade-offs with multiple data. In *13th International Workshop on Selected Areas in Cryptography (SAC 2006)* (2006), vol. 3897 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 110–127.

[5] BIRYUKOV, A., AND SHAMIR, A. Cryptanalytic time/memory/data tradeoffs for stream ciphers. In *6th International Conference on the Theory and Application of Cryptology and Information Security, Advances in Cryptology (ASIACRYPT 2000)* (2000), vol. 1976 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–13.

[6] BOGDANOV, A. Linear slide attacks on the KeeLoq block cipher. In *3rd International Conference on Information Security and Cryptology (INSCRYPT 2007)* (2007), vol. 4990 of *Lecture Notes in Computer Science*, Springer, pp. 66–80.

[7] BOGDANOV, A., KNUDSEN, L. R., LEANDER, G., PAAR, C., POSCHMANN, A., ROBSHAW, M. J., SEURIN, Y., AND VIKKELSOE, C. PRESENT: An ultra-lightweight block cipher. In *Cryptographic Hardware and Embedded Systems-CHES 2007*. Springer, 2007, pp. 450–466.

[8] BOGDANOV, A., AND PAAR, C. On the security and efficiency of real-world lightweight authentication protocols. In *1st Workshop on Secure Component and System Identification (SECSI 2008)* (2008), ECRYPT.

[9] BONO, S. C., GREEN, M., STUBBLEFIELD, A., JUELS, A., RUBIN, A. D., AND SZYDLO, M. Security analysis of a cryptographically-enabled RFID device. In *14th USENIX Security Symposium (USENIX Security 2005)* (2005), USENIX Association, pp. 1–16.

[10] BORST, J., PRENEEL, B., VANDEWALLE, J., AND V, J. On the time-memory tradeoff between exhaustive key search and table precomputation. In *19th Symposium in Information Theory in the Benelux* (1998), pp. 111–118.

[11] CHECKOWAY, S., MCCOY, D., KANTOR, B., ANDERSON, D., SHACHAM, H., SAVAGE, S., KOSCHER, K., CZESKIS, A., ROESNER, F., AND KOHNO, T. Comprehensive experimental analyses of automotive attack surfaces. In *20th USENIX Security Symposium (USENIX Security 2011)* (2011), USENIX Association, pp. 77–92.

[12] COURTOIS, N. T., BARD, G. V., AND WAGNER, D. Algebraic and slide attacks on KeeLoq. In *15th International Workshop on Fast Software Encryption (FSE 2008)* (2008), vol. 5086 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 97–115.

[13] COURTOIS, N. T., O'NEIL, S., AND QUISQUATER, J.-J. Practical algebraic attacks on the Hitag2 stream cipher. In *12th Information Security Conference (ISC 2009)* (2009), vol. 5735 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 167–176.

[14] DAEMEN, J., AND RIJMEN, V. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer-Verlag, 2002.

[15] DE CANNIERE, C., DUNKELMAN, O., AND KNEŽEVIĆ, M. KATAN and KTANTANa family of small and efficient hardware-oriented block ciphers. In *Cryptographic Hardware and Embedded Systems-CHES 2009*. Springer, 2009, pp. 272–288.

[16] DE KONING GANS, G., HOEPMAN, J.-H., AND GARCIA, F. D. A practical attack on the MIFARE Classic. In *8th Smart Card Research and Advanced Applications Conference (CARDIS 2008)* (2008), vol. 5189 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 267–282.

[17] DIAGNOSTICS, A. AD900Pro transponder duplicating system, operation manual, unlocking technology, March 2011.

[18] DRIESSEN, B., HUND, R., WILLEMS, C., PAAR, C., AND HOLZ, T. Don't trust satellite phones: A security analysis of two satphone standards. In *33rd IEEE Symposium on Security and Privacy (S&P 2012)* (2012), IEEE Computer Society, pp. 128–142.

[19] EM Microelectronic-Marin SA. http://www.emmicroelectronic.com. CH-2074 Marin/Switzerland.

[20] Crypto contactless identification device, V4070. Product Datasheet, Oct 1997. EM Microelectronic-Marin SA.

[21] 125khz crypto read/write contactless identification device, EM4170. Product Datasheet, Mar 2002.

EM Microelectronic-Marin SA.

[22] Custom automotive. retrieved at December 12th, 2014, from http://www.datasheetarchive.com/EM+MICROELECTRONIC-MARIN-datasheet.html, September 2002. EM Microelectronic-Marin SA.

[23] EM4170 application note, AN407. RFID Application Note 407, Sep 2002. EM Microelectronic-Marin SA.

[24] Fluhrer, S., Mantin, I., and Shamir, A. Weaknesses in the key scheduling algorithm of RC4. In *8th International Workshop on Selected Areas in Cryptography (SAC 2001)* (2001), vol. 2259 of *Lecture Notes in Computer Science*, pp. 1–24.

[25] Francillon, A., Danev, B., and Čapkun, S. Relay attacks on passive keyless entry and start systems in modern cars. In *18th Network and Distributed System Security Symposium (NDSS 2011)* (2011), The Internet Society.

[26] Garcia, F. D., de Koning Gans, G., Muijrers, R., van Rossum, P., Verdult, R., Wichers Schreur, R., and Jacobs, B. Dismantling MIFARE Classic. In *13th European Symposium on Research in Computer Security (ESORICS 2008)* (2008), vol. 5283 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 97–114.

[27] Garcia, F. D., de Koning Gans, G., and Verdult, R. Exposing iClass key diversification. In *5th USENIX Workshop on Offensive Technologies (WOOT 2011)* (2011), USENIX Association, pp. 128–136.

[28] Garcia, F. D., de Koning Gans, G., Verdult, R., and Meriac, M. Dismantling iClass and iClass Elite. In *17th European Symposium on Research in Computer Security (ESORICS 2012)* (2012), vol. 7459 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 697–715.

[29] Garcia, F. D., van Rossum, P., Verdult, R., and Wichers Schreur, R. Wirelessly pickpocketing a MIFARE Classic card. In *30th IEEE Symposium on Security and Privacy (S&P 2009)* (2009), IEEE Computer Society, pp. 3–15.

[30] Garcia, F. D., van Rossum, P., Verdult, R., and Wichers Schreur, R. Dismantling SecureMemory, CryptoMemory and CryptoRF. In *17th ACM Conference on Computer and Communications Security (CCS 2010)* (2010), ACM, pp. 250–259.

[31] Golić, J. D. Cryptanalysis of alleged A5 stream cipher. In *16th International Conference on the Theory and Application of Cryptographic Techniques, Advances in Cryptology (EUROCRYPT 1997)* (1997), vol. 1233 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 239–255.

[32] Hell, M., Johansson, T., and Meier, W. Grain: a stream cipher for constrained environments. *International Journal of Wireless and Mobile Computing 2*, 1 (2007), 86–93.

[33] Hellman, M. E. A cryptanalytic time-memory trade-off. *IEEE Transactions on Information Theory 26*, 4 (1980), 401–406.

[34] Hong, J., and Moon, S. A comparison of cryptanalytic tradeoff algorithms. *Journal of Cryptology* (2010), 1–79.

[35] Immler, V. Breaking hitag 2 revisited. *Security, Privacy, and Applied Cryptography Engineering (SPACE 2012) 7644* (2012), 126–143.

[36] Indesteege, S., Keller, N., Dunkelmann, O., Biham, E., and Preneel, B. A practical attack on KeeLoq. In *27th International Conference on the Theory and Application of Cryptographic Techniques, Advances in Cryptology (EUROCRYPT 2008)* (2008), vol. 4965 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–8.

[37] Kalenderi, M., Pnevmatikatos, D., Papaefstathiou, I., and Manifavas, C. Breaking the gsm a5/1 cryptography algorithm with wainbow tables and high-end FPGAS. In *22nd International Conference on Field Programmable Logic and Applications (FPL 2012)* (2012), IEEE Computer Society, pp. 747–753.

[38] Kasper, M., Kasper, T., Moradi, A., and Paar, C. Breaking KeeLoq in a flash: on extracting keys at lightning speed. In *2nd International Conference on Cryptology in Africa, Progress in Cryptology (AFRICACRYPT 2009)* (2009), vol. 5580 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 403–420.

[39] Keyline. Transponder guide. http://www.keyline.it/files/884/transponder_guide_16729.pdf, 2012.

[40] Koscher, K., Czeskis, A., Roesner, F., Patel, F., Kohno, T., Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., and Savage, S. Experimental security analysis of a modern automobile. In *31rd IEEE Symposium on Security and Privacy (S&P 2010)* (2010), IEEE Computer Society, pp. 447–462.

[41] Koushanfar, F., Sadeghi, A.-R., and Seudie, H. Eda for secure and dependable cybercars: Challenges and opportunities. In *49th Design Automation Conference (DAC 2012)* (2012), ACM, pp. 220–228.

[42] Kumar, S., Paar, C., Pelzl, J., Pfeiffer, G., and Schimmler, M. Breaking ciphers with COPACOBANA—a cost-optimized parallel code breaker. In *Cryptographic Hardware and Embedded Systems (CHES 2006)* (2006), vol. 4249 of *Lec-*

ture Notes in Computer Science, Springer-Verlag, pp. 101–118.

[43] Lemke, K., Sadeghi, A.-R., and Stüble, C. Anti-theft protection: Electronic immobilizers. *Embedded Security in Cars* (2006), 51–67.

[44] Lemke, K., Sadeghi, A.-R., and Stble, C. An open approach for designing secure electronic immobilizers. In *Information Security Practice and Experience (ISPEC 2005)* (2005), vol. 3439 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 230–242.

[45] Lucks, S., Schuler, A., Tews, E., Weinmann, R.-P., and Wenzel, M. Attacks on the DECT authentication mechanisms. In *9th Cryptographers' Track at the RSA Conference (CT-RSA 2009)* (2009), vol. 5473 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 48–65.

[46] Nohl, K., Evans, D., Starbug, and Plötz, H. Reverse engineering a cryptographic RFID tag. In *17th USENIX Security Symposium (USENIX Security 2008)* (2008), USENIX Association, pp. 185–193.

[47] Nohl, K., Tews, E., and Weinmann, R.-P. Cryptanalysis of the DECT standard cipher. In *17th International Workshop on Fast Software Encryption (FSE 2010)* (2010), vol. 6147 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–18.

[48] MIFARE DESFire EV1 contactless multi-application IC, MF3ICDx21. Product short data sheet, December 2010. NXP Semiconductors.

[49] Oechslin, P. Making a faster cryptanalytic time-memory trade-off. In *23rd International Cryptology Conference, Advances in Cryptology (CRYPTO 2003)* (2003), vol. 2729 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 617–630.

[50] Smart MX secure smart card controller IC, P5CC018. Objective Specification, Revision 1.0, April 2003. Philips Semiconductors.

[51] Rolles, R. Unpacking virtualization obfuscators. In *3rd USENIX Workshop on Offensive Technologies (WOOT 2009)* (2009).

[52] Soos, M., Nohl, K., and Castelluccia, C. Extending SAT solvers to cryptographic problems. In *12th International Conference on Theory and Applications of Satisfiability Testing (SAT 2009)* (2009), vol. 5584 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 244–257.

[53] Sun, S., Hu, L., Xie, Y., and Zeng, X. Cube cryptanalysis of Hitag2 stream cipher. In *10th International Conference on Cryptology and Network Security (CANS 2011)* (2011), vol. 7092 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 15–25.

[54] Tillich, S., and Wójcik, M. Security analysis of an open car immobilizer protocol stack. In *10th International Conference on Applied Cryptograpy and Network Security (ACNS 2012)* (2012).

[55] van Ours, J. C., and Vollaard, B. The engine immobilizer: a non-starter for car thieves, 2011.

[56] Verdult, R., de Koning Gans, G., and Garcia, F. D. A toolbox for RFID protocol analysis. In *4th International EURASIP Workshop on RFID Technology (EURASIP RFID 2012)* (2012), IEEE Computer Society, pp. 27–34.

[57] Verdult, R., Garcia, F. D., and Balasch, J. Gone in 360 seconds: Hijacking with Hitag2. In *21st USENIX Security Symposium (USENIX Security 2012)* (2012), USENIX Association, pp. 237–252.

[58] Štembera, P., and Novotný, M. Breaking Hitag2 with reconfigurable hardware. In *14th Euromicro Conference on Digital System Design (DSD 2011)* (2011), IEEE Computer Society, pp. 558–563.

[59] Wang, P.-C., Hou, T.-W., Wu, J.-H., and Chen, B.-C. A security module for car appliances. *International Journal of World Academy Of Science, Engineering and Technology 26* (2007), 155–160.

[60] Wiener, I. Philips/NXP Hitag2 PCF7936/46/47/52 stream cipher reference implementation. http://cryptolib.com/ciphers/hitag2/, 2007.

[61] Wolf, M., Weimerskirch, A., and Wollinger, T. State of the art: Embedding security in vehicles. *EURASIP Journal on Embedded Systems 2007* (2007), 074706.

[62] Wu, J.-H., Kung, C.-C., Rao, J.-H., Wang, P.-C., Lin, C.-L., and Hou, T.-W. Design of an in-vehicle anti-theft component. In *8th International Conference on Intelligent Systems Design and Applications (ISDA 2008)* (2008), vol. 1, IEEE