The Difference Engine

Geoffey Lefebvre, Brendan Cully, Dutch Meyer, Gitika Aggarwal,
Gang Peng, Mike Feeley, Norm Hutchinson
University of British Columbia

Andrew Warfield
University of British Columbia / XenSource

1 Introduction

All systems are a product of their histories. Events over time shape the state of OS and application software both for good and bad; while users make forward progress on productive work, bugs and malicious software may destabilize and corrupt their efforts. The notion of considering system state as mutable through time has been the subject of many recent projects. As examples, efforts have considered replaying slight permutations of recent events to recover from timing-related crashes[5], revisiting historical system states to identify the introduction of configuration errors[6], and rewinding execution to assist in debugging[2].

In all of these examples, revisiting—and in some cases even modifying—history allows the exploration of a system's state space of which the "current" incarnation is but one instance. We believe that there is broad benefit in providing general techniques to allow the more thorough exploration and analysis of the execution states of a given system. We are developing a tool to assist this exploration, which we have dubbed the *Difference Engine*.

The difference engine is primarily concerned with understanding the *divergence* between alternate states of a given system. For example, we may choose to create an alternate instance of a desktop OS in which a malicious network packet had never arrived, but for which the remainder of history had proceeded identically. The engine allows the controlled creation, and thorough analysis, of such divergence. It exploits the ability of virtual machines (VMs) to be check-pointed, rolled back and replayed to create alternate but plausible new outcomes. This set of outcomes, each represented by an independent VM instance, can be viewed as parallel universes where history occurred in subtly different ways.

In the case of the malicious packet just mentioned, the engine allows us to consider a large number of alternate universes that might result as a consequence of the packet's delivery being prevented, and to build insight into the specific mutations that its arrival induced.

The operation of the difference engine involves two phases, generational and analytical. In the generational stage, the difference engine allows the *replay* of logged external events to a historical version of a virtual machine. Replay is intentionally non-deterministic, and may be parametrized as to modify the stream of events that are delivered. In the second, *analysis* stage, the engine provides tools to assist with semantic comparisons between the resulting alternate states. These two stages are illustrated in Figure 1.

We are currently developing the difference engine as a tool based on the Xen virtual machine monitor. To date, we have had to grapple with two fundamental challenges: non-determinism of replay, and system semantics. These issues are closely related and introduce interesting obstacles in both the replay and analysis stages. Non-determinism is clearly necessary in order for replay to explore alternate states, but it demands that the replay support be tolerant of externally visible permutations of a system that impact the event log. Similarly, presenting a meaningful understanding of the divergence between a set of alternate instances requires sufficient semantic comprehension of a systems state as to recognize and summarize its differences.

We now discuss some specific challenges that non-determinism and semantics present in the replay and analysis stages and then provides an overview of some example applications. Broadly speaking, we believe that the difference engine represents a broadly useful tool for assisting in the exploration of "What if..." questions for large and complex software systems.

2 Replay

The challenge in replaying network traffic is dealing with non-determinism, both intentional and inherent. Intentional nondeterminism is caused through modifications (i.e. through reordering or otherwise altering the log of external events) to the replayed system in an attempt to guide the replaying instance to an alternate state. Inherent non-determinism results from within the system itself. Examples include both explicit calls to sources of entropy, for

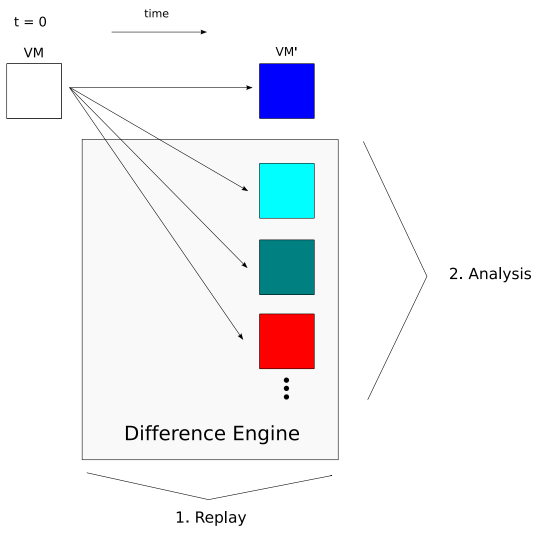

Figure 1: The Difference Engine

instance the use of random number generator to obtain initial TCP sequence numbers, and implicit sources such as memory races due to scheduling artifacts. In all of these cases non-determinism results in subtle (and not-so-subtle) permutions to a system's state that may make replay difficult: In the example of TCP sequence numbers for instance, simply replaying a log of traffic will be ineffective as connections simply do not succeed to the replaying host.

There are two ways of dealing with this non-determinism. First, it can be handled by the replay tool. By maintaining a richer understanding of the semantics of traffic being delivered to a replayed system, network traffic may be modified to mimic the behavior of live external clients. For example, our replay tool supports rewriting TCP headers to handle randomly chosen initial sequence number and comprehends TCP stream semantics sufficiently to tolerate alternate payload segmentations.

More generally dealing with non-determinism during replay requires the replay component to be extensible in order to handle the application and protocol semantics. Our current implementation is built using Click[3] which allows to easily add custom elements to handle the semantics of different protocols. We are also actively integrating protocol parser generated with binpac[4] into click elements to facilitate the construction of custom protocol replay handlers.

In order to address inherent non-determinism, we plan to use a technique that we have dubbed *paradeterminism*: we modify systems to become intentionally more deterministic. Library and operating system code will be altered to log the result of important non-deterministic events and use this log on replay[1].

3 Analysis

The goal of the analysis phase is to provide a useful summary of divergence. We aim to summarize the difference between hundreds of alternate replayed instances. The need to bridge the semantic gap represents a formidable challenge here. However, the degree of understanding varies with the type of state exploration being performed. We have implemented prototypes for both block- and file system-level differencing and plan to provide hooks for additional analysis "plug-ins".

The block-level analyzer compares mappings between files and blocks across divergent instances. This allows us to evaluate things such as block placements strategies, and to understand differences in how storage systems age.

File level semantics allow us to, for example, identify the set of files that were only modified in a specific subset of replays. This type of analysis can provide forensic insight in understanding the damage caused by an intrusion.

4 Applications

An interesting application of the difference engine is to answer the question what would my system look like if a specific intrusion never occurred. By rolling back to a checkpoint prior to the intrusion and replaying forward all input minus the packet that actually triggered the intrusion, the difference engine would create a "clean" version of the infected system. Using these different versions of the world, file level state analysis can provide insight on the extent of the damage done by the intrusion but also on the potential modification needed to be applied to the clean system in order for the latter to replace the original infected version.

References

[1] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, 1978.

[2] S. King, G. Dunlap, and P. Chen. Debugging operating systems with time-traveling virtual machines. In *Usenix ATC*, 2005.

[3] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[4] R. Pang, V. Paxson, R. Sommer, and L. Peterson. binpac: a yacc for writing application protocol parsers. In *IMC '06*, pages 289–300, New York, NY, USA, 2006.

[5] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *SOSP '05*, 2005.

[6] A. Whitaker, R. Cox, and S. Gribble. Configuration debugging as search: Finding the needle in the haystack, 2004.