XTRec: Secure Real-time Execution Trace Recording on Commodity Platforms

Amit Vasudevan
CyLab/CMU
amitvasudevan@acm.org

Ning Qu
CyLab/CMU
quning@gmail.com

Adrian Perrig
CyLab/CMU
perrig@cmu.edu

Abstract

We propose XTRec, a primitive that can record the instruction-level execution trace of a commodity computing system. Our primitive is resilient to compromise to provide integrity of the recorded execution trace. We implement XTRec on the AMD platform running the Windows OS. The only software component that is trusted in the system during runtime is XTRec itself, which contains only 2,195 lines of code permitting manual audits to ensure security and safety. We use XTRec to show whether a particular code has been executed on a system, or conversely to prove that some malware has not executed on the system. This is a highly desirable property to ensure information assurance, especially in critical e-government infrastructure. Our experimental results show that the imposed overhead is 2x–4x for real-world applications. This overhead is primarily due to CPU Branch Trace Messages(BTM), a ubiquitous debugging feature used to record control-flow instructions. Hardware improvements to BTM would therefore enable XTRec to run with minimal overhead.

1 Introduction

Current operating systems and applications continue to be plagued by security vulnerabilities that enable an attacker to compromise a system and execute arbitrary operations. In fact, new vulnerabilities that enable a remote attacker to compromise a system are still discovered on a weekly basis. In such an environment, an operator of applications with security-sensitive data would like to answer two important questions: given a newly discovered vulnerability, did attackers exploit this vulnerability to compromise his/her systems; and if yes, what operations did they perform?

Execution trace recording helps answer these questions. Given a complete and exact trace of execution of an entire system, it is possible to deduce all operations that took place. Such information can even be used to demonstrate that some operation *did not happen*, which is a highly desirable property to ensure information assurance. This is especially true in the context of critical e-government infrastructure such as systems used for lodging tax returns or those employed for online voting. A breach in such security critical systems can result in the loss and/or manipulation of highly sensitive user information.

To illustrate these points, we consider the Haxdoor malware [13] which compromised several systems within Europe in 2006–2007 resulting in massive data theft and financial loss [12]. The attacks made use of the A-311 Death Backdoor [33] which enables a remote attacker to take complete control of a system, perform attacks, and remove itself completely leaving no evidence that it ever executed. In the case of Swedish bank Nordea, attackers used the malware to steal relevant information from systems within the bank and performed internal transfers over the next 15 months. Execution trace recording would enable concerned enterprises to know whether they had been affected after the malware was documented, and what operations the attackers performed.

Given the importance of execution trace recording, the goal of this work is to provide an unobtrusive and efficient mechanism that provides this property for current legacy computing environments, such as Windows and Linux. We seek an approach that does not mandate ad hoc hardware or changes to existing operating systems or applications. We present XTRec, a primitive which records the execution trace of the entire system (including the OS and applications) *in real time at the instruction level*. Further, XTRec provides robustness against subversion and integrity of the recorded execution trace.

Current approaches for execution trace recording are largely based on the concept of *deterministic replay*, where coarse-grained deterministic and non-deterministic events and memory snapshots (checkpoint) of the system are recorded using a virtual machine monitor (VMM). These events are then replayed at a later time in order to reconstruct the entire system state. However, this approach has several shortcomings. First, for server platforms, the replay time is proportional to the length of time the system has been running since the last checkpoint. Since checkpoints incur significant latency and are minimally used, replay increases analysis time considerably thereby defeating the goal of forensic analysis where minimizing time is of utmost importance. Second, non-determinism such as network and disk data result in huge overhead in terms of space and time, especially for server environments. Further, it is prohibitively slow to accurately capture all forms of non-determinism (e.g, device states, device interrupts and CPU micro-architectural states) at runtime on current commodity platforms without employing ad hoc platform hardware [29]. Finally, current approaches rely on general purpose VMMs which have a large Trusted Computing Base (TCB). This renders them suceptible to various attacks owing to their size and complexity [14], thereby enabling an attacker to tamper with the recorded events. We discuss deterministic replay approaches in further detail in Section 7.

In contrast to deterministic replay systems which aim at reconstructing the entire system state by replay, XTRec focuses on obtaining only the execution trace in real time. Thus, there is no replay overhead to perform trace analysis. Also, we do not record any data. Based on recent findings which show how to accurately detect malware using only instruction traces [7, 6], we argue that the execution trace information in itself is sufficient to show if a system has been compromised.

XTRec uses several novel approaches for real-time execution trace recording. First, XTRec employs the Branch Trace Message (BTM) and physical memory virtualization features available in all commodity x86-class CPUs to obtain the instruction-level execution trace. Since trace recording is done in real time, it results in smaller space and time overhead as there is no need to store any non-deterministic data (e.g., data from network and disk accesses). Second, XTRec reserves a network interface for its exclusive use on an untrusted system where it is deployed and stores the execution trace on a remote trusted system. Our primitive employs a tiny hypervisor and uses commodity hardware features to provide robustness against subversion thereby guaranteeing the integrity of the recorded execution trace. Finally, XTRec relies on a small subset of system hardware for all its functionality. This helps keep XTRec's codebase very simple and small thereby making it amenable to formal verification to rule out potential vulnerabilities.

2 Problem Statement

Goals: Our primary goal is to record the execution trace of a system at the instruction level in real time while simultaneously ensuring the integrity of the recorded information. Further, we wish to achieve the above goal on commodity platforms without adding specialized hardware or modifiying any software including the operating system.

Adversary Model: We consider an attacker without physical access to tamper with the CPU, chipset, memory or the network interface/connection used by XTRec. The attacker can use any method to take control of the system (XTRec is assumed to be secure) and can execute any arbitrary code within the system.

Assumptions: We assume the following: (a) system support for branch trace messages (BTM), which all x86-class CPUs provide, (b) support for hardware CPU virtualization including hardware-based physical memory virtualization (e.g., Nested Page Tables) and support for hardware-attestation (e.g., TPM chip). Both technologies are rapidly becoming ubiquitous, (c) support for a storage area network (SAN) for recording purposes, a reliable and loss-less gigabit-speed network connection (optical fiber/EMI-resistant ethernet cabling) to the SAN and a dedicated gigabit network interface to the SAN. These are typical enterprise settings, (d) CPU features (hardware virtualization, BTMs and physical memory virtualization) and TPM hardware attestation are free of vulnerabilities, and (e) XTRec's code does not contain vulnerabilities. Given the dramatically reduced TCB of XTRec (2,195 lines) compared to previous work, we argue that manual audits can validate this assumption.

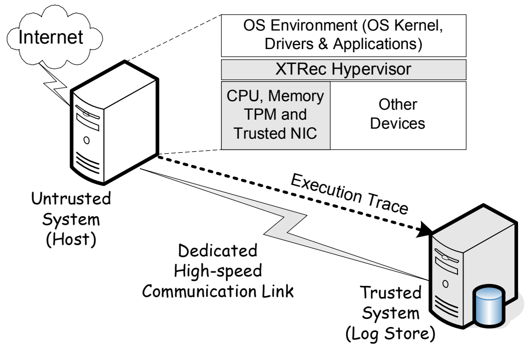

Internet
OS Environment (OS Kernel, Drivers & Applications)
XTRec Hypervisor
CPU, Memory TPM and Trusted NIC
Other Devices
Execution Trace
Untrusted System (Host)
Dedicated High-speed Communication Link
Trusted System (Log Store)

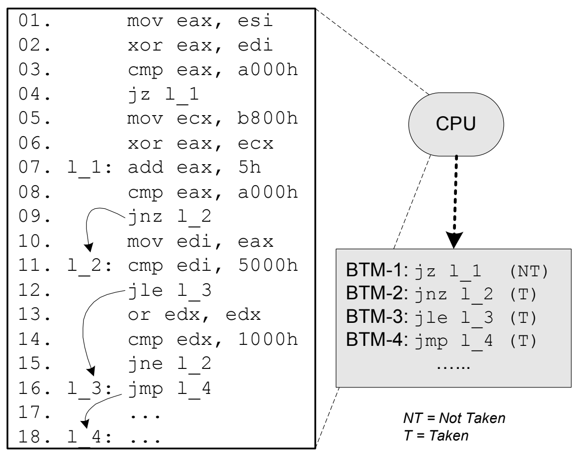
Figure 1: XTRec is deployed on an untrusted system (host) and records host execution trace to an external trusted system (log-store) in real-time. Shaded portions represent trusted components.

3 Design

3.1 Overview

XTRec is intended for deployment on untrusted systems or hosts (Figure 1). Since XTRec collects large volumes of real-time information, our system currently relies on a fast transmission medium such as a gigabit network interface/connection and stores the collected information on a trusted entity (log-store). A host would typically be an enterprise server while the log-store would be a storage area network or an inexpensive disk array system such as ATA-over-Ethernet.

The XTRec architecture is based on two central concepts: (i) **real-time recording:** the execution trace information is recorded in *real time* using the CPU as opposed to event record and replay approaches employing a VMM. This eliminates system replay, thereby enabling fast forensic analysis and does away with issues associated with non-determinism (e.g., reconstructing CPU micro-architectural states and storing data for all network and disk accesses); (ii) **secure recording:** XTRec maintains the integrity of the recorded execution trace by storing the information on an external log-store and by preventing access to the network interface, CPU mechanisms and memory areas used for trace collection within the host. Further, since XTRec relies on CPU hardware to perform most of the functionality, its codebase is simple and tiny to rule out potential vulnerabilities. In contrast, current deterministic replay approaches use general-purpose VMMs which are suceptible to various attacks due to their size and complexity [14, 19].



```
     mov eax, esi
     xor eax, edi
     cmp eax, a000h
     jz l_1
     mov ecx, b800h
     xor eax, ecx
l_1: add eax, 5h
     cmp eax, a000h
     jnz l_2
     mov edi, eax
l_2: cmp edi, 5000h
     jle l_3
     or edx, edx
     cmp edx, 1000h
     jne l_2
l_3: jmp l_4
     ...
l_4: ...
```

```
BTM-1: jz l_1  (NT)
BTM-2: jnz l_2 (T)
BTM-3: jle l_3 (T)
BTM-4: jmp l_4 (T)
       ......
```

Figure 2: CPU Branch Trace Messages record control-flow instructions which are then superimposed over dynamically captured code for a complete execution trace.

3.2 Real-time Recording

XTRec leverages a special feature in a regular x86-class CPU, whereby the CPU emits details about every control-flow instruction that is encountered during program execution. XTRec also leverages physical memory virtualization to obtain non control-flow instructions. Thus, a complete instruction level execution trace is recorded in real time. Finally, XTRec simultaneously transmits the collected execution trace to the log-store over a trusted network.

3.2.1 Branch Trace Messages

All x86-class CPUs contain support for a debugging feature called a Branch Trace Message (BTM). A BTM enabled CPU emits a special message for every branch (conditional or unconditional) that is decoded at the current instruction pointer. This includes conditional jumps, unconditional jumps, loops, procedure invocations, returns from procedures, interrupts, exceptions, and return from interrupts and exceptions. BTMs are usually sent out on the system bus, but the CPU can also be configured to send the BTMs to system physical memory. Figure 2 shows a code fragment and the BTMs generated by the CPU.

BTMs occur irrespective of the operating mode or privilege level of the CPU. This is crucial to our architecture, since we do not need to modify any code executing in the host to enable recording the execution trace. Further, BTMs are directly generated from the Execution Unit of the CPU, so the CPU will always generate a BTM irrespective of the type of control-flow transfer. Thus XTRec can support any form of commodity code including obfuscated, polymorphic and metamorphic code found in all current generation malware. Further, the BTM feature is controlled with a very small set of CPU registers and involves very little intervention at runtime. This helps keep XTRec's codebase simple and tiny.

3.2.2 Dynamic Code Capture

BTMs enable recording of only control-flow instructions. This information can then be superimposed over static code (executables or libraries) in order to obtain an instruction-level execution trace. However, most if not all malicious code employs some form of dynamic code generation (e.g., polymorphism and metamorphism). Further, certain classes of malware employ sophisticated coding mechanisms which closely mimic the control-flow instruction sequences of a benign program. XTRec uses hardware physical memory virtualization to obtain the code corresponding to the BTMs during runtime.

XTRec uses page protections within hardware managed physical memory page tables to enforce a W $\oplus$ X policy on physical memory pages used within the host OS environment. Thus, a page within the host OS environment may be executable or writable, but not both. XTRec always records the contents of a page prior to converting the page to executable status. Thus, at any given point in the recorded information, a BTM can always be superimposed on the corresponding code page contents to obtain the complete execution trace.

Previous work use a similar page protection mechanism for approved code execution in the kernel [31], or to determine what applications are running [23]. In contrast, XTRec uses these page protections to record the contents of a page where code is executed.

3.2.3 Recording to the Log-Store

XTRec records the execution trace to an external trusted log-store over a trusted network. Given the fine granularity at which the execution trace information is recorded, it is very important to eliminate any bottleneck as a result of the network transmission. XTRec leverages commodity gigabit network interface features in order to ensure simultaneous execution trace collection and network transmission.

All gigabit network interfaces contain support for a ring of descriptors, where each descriptor describes a range of physical memory to be transferred using DMA. Further, once these descriptors are setup, the entire ring can be transmitted in one shot via DMA. Our implementation and experimental results show that by carefully choosing the size and arrangement of the ring of descriptors and the physical memory buffers, the network transmission can proceed simultaneously with the collection of execution trace information and stalls due to network transmission can be avoided.

3.3 Secure Recording

A host with XTRec runs a commodity OS and untrusted applications. Thus, vulnerabilities within the OS or an application can be exploited either locally or remotely to execute malicious code. Such malicious code could: (a) tamper with the BTM functionality or memory regions belonging to XTRec, and/or (b) impersonate XTRec to log incorrect execution trace information to the log-store. XTRec handles both cases using a combination of CPU protections and hardware attestation (via a TPM).

Memory Protection: XTRec partitions the available physical memory within the host into two areas: the XTRec memory region and the host OS memory region. XTRec employs CPU physical memory page tables (PMPT) to restrict the host OS to its own physical memory region. In other words, the PMPT for the host OS does not map physical memory pages that belong to XTRec. Further, XTRec employs hardware-based DMA protection to prevent DMA-based access to its memory regions. This prevents malicious devices in the host from directly accessing memory regions belonging to XTRec.

BTM Protection: BTM features on x86-class CPUs are controlled by CPU registers. XTRec uses CPU hardware virtualization to trap any access to CPU BTM registers from the host OS environment.

Secure Communication with the Log-Store: To create a secure channel for communicating with the log-store, XTRec uses CPU hardware virtualization to prevent the host OS from accessing the network interface that connects the host to the log-store.

To convince the log-store that it is communicating with XTRec, we use TPM-based attestation. Initially, XTRec is started at system boot using a *late launch* operation, which records a measurement of XTRec in the TPM. XTRec then waits for a challenge (a cryptographic nonce) from the log-store. XTRec uses the TPM to generate a quote (essentially a signed statement describing the software state of the system and the challenge nonce) that it transmits to the log-store, which checks the attestation. The correct attestation ensures the log-store that XTRec is correctly executing with the correct protections in place. The log-store starts accepting data from the host only if the verification succeeds.

XTRec uses CPU hardware virtualization mechanisms to detect when the host is restarted or shutdown. When XTRec detects such an event, it sends the log-store a special network packet signifying the end of recording. Upon receipt of this special packet, the log-store stops accepting any further data from the host and performs the procedure described in the previous paragraph to verify that XTRec has started again.

4 Implementation

We implemented a prototype of XTRec with Windows 2003 Server SP1 as the OS on a host running the AMD Opteron CPU with SVM extensions. Our prototype currently supports execution trace recording for a single CPU.

4.1 XTRec Components

XTRec consists of two pieces: the Loader and the Runtime. The Loader uses the SKINIT instruction to run in a hardware-protected environment and to store a measurement (cryptographic hash) of its memory regions in TPM PCR 17. The Loader loads the Runtime and protects its memory regions from DMA access using SVM Device Exclusion Vector (DEV) protections. It then verifies the integrity of the Runtime and extends a measurement (a cryptographic hash) of the Runtime memory regions into TPM PCR 18. The Loader initializes the dedicated network interface within the host for communication with the log-store, creates the Nested Page Tables (NPTs) for the host OS environment, and transfers control to the Runtime.

The Runtime implements all of XTRec's functionality. When first launched, the Runtime waits for a challenge from the Log-store. The Runtime and the Log-store then engage in the authentication protocol described in Section 3.3. The Runtime then starts the host OS with CPU Branch Trace Messaging (BTM) enabled. Runtime's role in real-time recording and protection of recording is described below.

4.2 Real-time Recording

4.2.1 Branch Trace Messages

On AMD CPUs, BTMs are managed using a set of Machine Specific Registers (MSR) [1] . These MSRs are used to control BTM generation and specify the region of physical memory that the CPU can use for storing BTMs. The MSRs are accessed via the *RDMSR* and *WRMSR* instructions. The BTM control logic also provides an option to generate a #DB exception when the BTM physical memory region is full.

Our current implementation uses a 128MB contiguous physical memory region within the host for execution trace recording. This region, called the *execution trace buffer*, is further divided into two 64MB logical regions. At any given point in time one area is a *collect buffer* (where trace information is collected) while the other area is designated as a *transmit buffer* (which is transmitted to the log-store).

4.2.2 Dynamic Code Capture

Prior to executing the host OS environment, XTRec sets its NPT entries to prevent execution of all physical pages belonging to host OS. When the host OS environment attempts to execute a page, it causes a nested page-fault (NPF) that returns control to XTRec. XTRec then copies the contents of the page and updates the NPT entry of the page to allow execution but prevent writes. If the host OS environment later writes to this page, a write fault will be generated. XTRec will re-enable writing but disable execution.

4.2.3 Recording to the Log-store

We used an Intel 82572EI gigabit network interface within the host to transmit execution traces to the log-store. We developed a tiny 82572EI driver within XTRec for this purpose. Our driver leverages the jumbo and packet-split features that are common to all gigabit network interfaces in order to transmit the execution trace buffer via Direct Memory Acess (DMA) at runtime. Our experimental results

[1]The BTM feature on x86 CPUs is sparsely documented today. While Intel documents the BTM feature as part of the *Debugging* section in their developer's manuals [17], AMD does not document the BTM feature at all. All information pertaining to the AMD BTMs in this section are a result of system-level exploration efforts spanning several months. As a result, we also found that BTMs were supported on all recent Intel and AMD x86 CPUs.

(Section 6.2.2) show that the network transmission occurs in parallel with execution trace collection.

4.3 Secure Recording

Memory Protection: In our current implementation on a host with 4 GB of physical memory, XTRec allocates 3.5 GB of physical memory to the host OS environment, while it reserves 136 MB for itself and 258 MB for the systems firmware. Memory protection is maintained by creating the NPTs. The NPT entries which point to XTRec's physical memory regions are marked not-present.

BTM Protection: XTRec sets up MSR intercepts in the VMCB for the BTM Machine Specific Registers (MSR). Our current implementation disallows any access to such MSRs upon the intercept triggering.

Secure Communication with the Log-store: The Intel 82572EI Gigabit network interface (NI) uses memory-mapped registers. The location and size of the memory range is exposed through the PCI configuration space which is accessed using the PCI address and data registers. XTRec sets up the NPTs such that entries for the memory range for the dedicated NI are marked absent. XTRec also sets up an I/O intercept on the PCI data register. Upon this intercept, the PCI address register is examined to obtain the bus, device and function of the request. If it matches that of the dedicated NI, a device not present result is returned.

We used an Intel Dual Xeon (2 x quad-core) system equipped with a gigabit NI as the Log-store. This system is connected to the host via a gigabit switch. The XTRec runtime contains a tiny network driver that is used to control the dedicated NI and communicate with the Log-store. The XTRec runtime also contains a small TPM v1.2 driver which communicates with the TPM.

The Log-store waits for a challenge request from XTRec. Upon receiving the challenge request, the Log-store transmits a cryptographic nonce and receives a TPM-generated attestation from XTRec. The attestation contains the TPMs signature over the current values of PCRs 17 and 18, as well as the nonce that was provided. The Log-store uses the TPMs public-key (obtained using any of the numerous existing methods [26]) to verify the attestation. If the verification succeeds, the Log-store enters a trusted communication mode with XTRec and begins accepting and storing data until a end of recording packet is received.

5 Security Analysis

XTRec uses CPU BTMs and physical memory virtualization in order to obtain a complete instrution-level execution trace dynamically. XTRec also uses a dedicated network interface within the host to transmit the execution trace information to a trusted log-store. In order to ensure the integrity of the recorded execution trace, these operations and the memory areas they use must be protected at all times.

XTRec's use of hardware physical memory page tables (PMPT) ensures that software in the host OS environment cannot address XTRec's physical memory region, thus protecting its secrecy and integrity. To prevent device-based attacks, XTRec uses hardware-based DMA protections to prevent DMA-based reads and writes to its memory region.

XTRec uses the CPU to generate BTMs to a portion of its own protected memory region and uses CPU hardware virtualization to trap access to CPU registers that control BTM functionality. Thus, adversarial code within the host OS cannot manipulate BTMs or prevent BTMs from being generated. XTRec uses the PMPT to set page protections for dynamic code capture (Section 3.2.2). The PMPT are also stored within XTRec's protected memory region, thereby ensuring that dynamic code capture is always in effect.

Finally, XTRec records the execution trace to the log-store over a secure channel that is established during its startup (Section 3.3). This channel is protected by XTRec's exclusive access to the dedicated network interface that connects to the log-store combined with the TPM's ability to provide a verifiable summary of the host system's software. Therefore, an adversary cannot record incorrect execution trace information to the log-store.

5.1 Other Attacks

Denial of Service Attack Since, XTRec resides on an untrusted host and is loaded on host startup, malware in the untrusted host can modify the startup sequence to prevent XTRec from loading. However, it cannot do so undetectably.

The log-store always verifies that XTRec has been loaded on the host before starting to accept any information from it. If the host does not load XTRec, then the verification will never complete and the log-store will not accept any information from the host. While this results in no forensic evidence, it is a clear indication of unintended (probably malicious) activity. This is a form of denial-of-service, but one that can be detected.

Alias Attack CPU BTMs contain linear addresses which need to be translated to physical addresses to tie a BTM to a corresponding code page. This information is obtained by XTRec on a Nested Page Fault (NPF) during dynamic code capture (Section 3.2.2). However, once a physical page is set to execute during dynamic code capture, it may no longer generate a NPF due to code execution. An attacker can then modify the host OS page tables to *alias* the physical page with different linear addresses with the result that BTMs can no longer be superimposed on corresponding code. We are currently working on an implementation using shadow page tables [34, 37] to prevent such attacks.

6 Evaluation

In this section we first evaluate our XTRec prototype using two metrics: code size and performance. We follow that with a case study of the HaxDoor.KI malware.

6.1 Trusted Computing Base (TCB)

XTRec must assume the correctness and security of its core components. This assumption can hold if the code-base is tiny, to reduce potential vulnerabilities and make

XTRec Component	Lines of Code (LOC)
Loader	1,435
Runtime	
Core	185
TPM v1.2	298
Dynamic Code Capture	84
Branch Trace Messages	104
NIC Transmission	41
Common Library	48
Total LOC	**2,195**

Figure 3: Code-base of XTRec is small to permit formal analysis to rule out vulnerabilities.

it amenable to formal analysis. We use the codecount [2] utility to measure the logical Lines of Code (LOC) of our prototype implementation (Figure 3). The prototype code consists of three components. The Loader initializes the CPU, memory, NIC and Runtime. The Runtime handles all of XTRec's functionality. Finally, the common library code is used by both the Loader and the Runtime for certain memory copy and move operations. Combining all components, XTRec's total TCB is only 2,195 LOC, placing it within means of manual and analytical audit techniques.

XTRec's TCB is an order of magnitude smaller than the TCB of hypervisors that are used in deterministic replay approaches. VMware ESX includes a large TCB (approx. 500,000 LOC [35]) as it employs full virtualization of devices and hence must include device drivers for all of the platforms it wishes to support. Xen on the other hand includes an entire Linux OS for administrative purposes, dramatically increasing its TCB [35]. Note that we exclude the log-store from our TCB. In practice, the log-store is physically and remotely isolated from an attacker (access-controlled room and disconnected from the general-purpose network).

6.2 Performance Measurements

We now present the runtime microbenchmarks and application benchmarks for XTRec. For all our experiments we used a Dell 740 Workstation as the host running Windows 2003 Server SP1. The system was equipped with an AMD Phenom 2.4Ghz CPU, 500GB hard disk, 4GB physical memory, and a v1.2 Broadcom TPM chip. We used an Intel 82572EI PCI-Express gigabit NIC to connect to the log-store. The log-store was an Intel Dual Xeon (2 x quad-core, 3.0GHz) system with 16GB physical memory, and 1TB harddisk running Linux 2.6.23.1. Note that we do not compare against current deterministic replay systems as such a comparison would not be meaningful. Deterministic replay systems record coarse-grained events and replay them to reconstruct the system state whereas XTRec directly records the complete execution trace without the need to replay. We discuss the performance of deterministic replay systems in Section 7.

6.2.1 Microbenchmarks

XTRec adds overhead to system operation in three ways: (a) BTMs handled by the CPU, (b) dynamic code capture, and (c) network transmission.

[2] http://sunset.usc.edu/research/CODECOUNT/

BTM latency is due to the CPU emitting BTMs to memory and due to the #DB exception that is raised by the CPU when the BTM buffer is full. We employed a tight loop with a `CALL` instruction and measured the time before the `CALL` and at the start of the subroutine. We measured the #DB exception latency by setting the OS trap-flag and measuring the round-trip time from XTRec to the OS and back. The BTM emission latency and the #DB exception latency were on average 2.72 and 1120 clock cycles respectively. The dynamic code capture latency is due to the triggering of NPF and the copying of the corresponding memory page contents and averaged 945 and 512 clock cycles respectively. We measured this latency by setting up the NPTs to generate a NPF exception for a single memory page within the host OS and resuming the host OS on that memory page. Finally, the network transmission latency averaged 769,019,125 clock cycles. This is the time taken by the trusted NIC to finish a DMA transfer and was measured by invoking the trusted NIC transmission function in a tight loop.

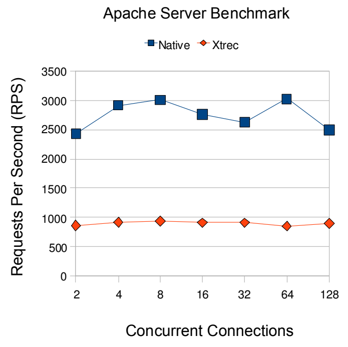
Note that network transmission incurs a large clock cycle count. However, this only occurs when the collect buffer is full. Further, network transmission occurs in parallel with execution trace collection for all our application benchmarks, resulting in no perceptible latency.

6.2.2 Application Benchmarks

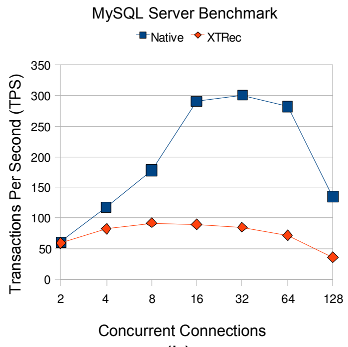
We hypothesize that when XTRec is used, the overhead of an application will be directly proportional to the number of control-flow instructions that are encountered by the CPU during application execution. Based on our hypothesis, I/O bound applications with few control decisions will have the lowest overhead. On the other hand, compute bound applications with high branching will have the highest overhead. Consequently, compute bound applications with high branching will require more log space when compared to I/O bound applications.

To test our hypothesis we execute both I/O bound and compute bound applications on a Native system and a system running XTRec. For our I/O bound applications we chose an Apache 2.2 webserver, a MySQL 5.1.33 database server, Postmark (with 10000 files and 10000 transactions), IoZone (read and write benchmarks), Bonnie (with a 100MB file and read and write) and Tar (on the Windows system folder). We used the cygwin runtime environment to run Postmark, Tar and Bonnie. We used the SPEC CINT 2006 benchmarks for our compute bound applications. We used the Apache ab benchmarking tool to benchmark the webserver and employed the sysbench utility to benchmark the database server using the oltp option and the innodb database engine.

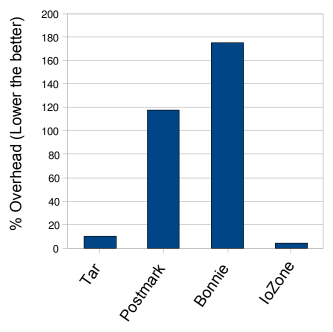
Figures 4a–c show the result of our I/O bound application benchmarks. The Apache webserver running under XTRec exhibits a slowdown of 2.7x–3.2x while the MySQL server exhibits a slowdown of 0–3.3x. The overhead increases with the number of concurrent connections. Certain I/O bound applications such as tar and iozone run with low overheads (7% and 10% respectively). We attribute this to the benchmarks having low control flow de-



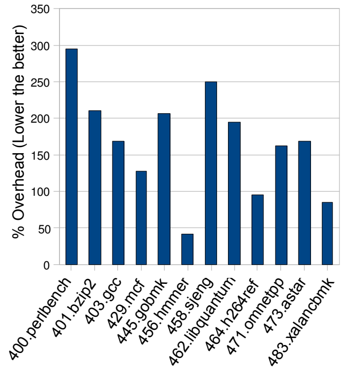
(a)



(b)



(c)



(d)

Benchmark	Runtime (seconds)	Compressed Trace Size (GB)	Comp. Ratio	% NIC B/W
I/O Bound Applications				
Apache Server	135	0.7	5.5	25
MySQL Server	119	0.7	7.8	42
Tar	83	0.1	9.7	13
Postmark	144	0.4	20.0	47
Bonnie	557	0.3	104.0	44
IoZone	954	0.3	27.0	7
Compute Bound Applications				
400 perlbench	3490	15.7	13.5	52
401.bzip2	3260	19.9	8.2	43
403.gcc	2390	8.0	13.5	39
429.mcf	2190	3.8	7.7	12
445.gobmk	2450	37.3	3.5	46
456.hmmer	3360	20.0	4.1	21
458.sjeng	3710	46.8	4.3	46
462.libquantum	4300	2.0	92.4	37
464.h264ref	2670	9.0	12.0	35
471.omnetpp	2150	5.7	11.7	27
473.astar	2510	14.4	5.5	27
483.xalancbmk	16600	8.5	51.10	22
Others				
Idle Server	117	0.007	18	1

B/W = Bandwidth, Compression used: gzip -5

(e)

Figure 4: XTRec Application Benchmarks: (a)–(d) I/O bound and Compute-bound applications execute with a 2x–4x overhead. The majority of the overhead is due to CPU Branch Traces Messages, and (e) Log-sizes generated by the benchmarks

cisions. Bonnie and Postmark on the other hand incur considerable latency. We attribute this to the internal architecture of these benchmarks which issues reads and writes to the disk system one byte at a time using a loop structure. This results in a large amount of branch instructions that contributes towards the overhead.

Compute bound applications result in higher latency and their latency is proportional to the number of branch instructions (Figure 4d). As an example, 456.hmmer has the smallest slowdown when compared to other benchmarks. This is due to a much smaller number of branch instructions in 456.hmmer when compared to the other benchmarks [27].

Figure 4e shows the log sizes generated by various applications with XTRec. For the first set of experiments, we ran the Apache ab benchmark tool with 128 simultaneous clients simulating a real world webserver access. We also ran the sysbench tool with 16 concurrent connections. Finally we let the host idle. As seen from the figure, the log sizes for these experiments is within limits for an enterprise storage area network. While the time period for which logs should be maintained, depends upon the task for which XTRec is used and the available resources, given the size of our logs, enterprises could easily maintain logs for a period of 3–4 months. It is important to note that the XTRec's log size is an order of magnitude smaller than deterministic replay approaches. For example, a real-world apache webserver operating 24 hours a day with for a period of 3 months would result in a huge amount of log with deterministic replay as all network and disk access data for the entire 3 month period must be captured and stored. On the contrary, the same webserver under XTRec will result in around 40 TB of log irrespective of the network traffic and disk accesses.

For the second set of experiments we ran various compute bound and I/O bound applications. As seen, the log size of an application is proportional to the number of branch instructions. As examples, 445.gobmk and 458.sjeng which have higher number branch instructions generate larger logs. Though the compression ratio is application specific, the average compression ratio during our experiments, as seen from Figure 4e is around 14 for I/O bound applications and around 8.5 for compute bound applications. The compression is done in real time by the log store.

Figure 4e also shows that on an average only 1/3rd of a gigabit network interface (NI) bandwidth is used for execution trace transmission for a single CPU. This makes XTRec readily scalable to 3–4 CPUs with a single gigabit NI. We postulate that with high-end gigabit NIs (e.g., 10 gigabit) XTRec can easily scale to future multicore CPUs with 8–12 cores.

For all our benchmarks, we noted that hardware physical memory virtualization and dynamic code capture contributed minimally to the results. A code page is logged only during the loading of a benchmark executable and represents a small set of memory pages corresponding to the benchmark code. For example, for the SPEC benchmarks in Figure 4e, only an average of 292 code-pages (0.001% of the log-size) were recorded. Further, for all our benchmarks, we noted that when the collect buffer was full, the polling of the network interface (NI) status register always indicated that the transmit buffer was successfully transmitted. In other words, the network transmission was occuring in parallel to the execution trace collection. This is affirmed by the log rates of all our application benchmarks which average only 1/3rd of a gigabit NI transmission bandwidth. Thus, we conclude that CPU BTMs are the primary source of latency in our experiments. Hardware enhancements to BTM generation would therefore greatly improve XTRec's performance.

6.3 Case Study of Haxdoor.KI

In this section we discuss the *Haxdoor.KI* malware and show how one can determine it executed on a host using XTRec. We use the technique of code normalization [6] to compare the execution trace obtained by XTRec to reference traces which describe a particular malware. Code normalization decomposes a binary instruction stream into a high level representation which can then be semantically compared without any need for operand values. It has been successfully used to identify malware and its variants [6].

We obtained the reference execution traces of Haxdoor from the documentation of its internal workings [20]. We then downloaded the Haxdoor sample from Offensive Computing [9] and deployed it on the host system running XTRec. We used the A311 Remote Administration Toolkit on another system to connect to the backdoor in the host. The compressed log-size generated by XTRec for our experiment was 643 MB.

Haxdoor protects its own threads and processes from being accessed by the system. It does so by rewriting the OS kernel functions NtOpenThread and NtOpenProcess. If the hook function notices any access to Haxdoor's thread it simply swaps the target thread/process identifiers which will lead to the operations being performed on the calling thread/process instead. Figure 5a shows the reference trace which is representative of this mechanism. Its high-level representation and normalized version are shown in Figure 5b and Figure 5c respectively.

We searched through the XTRec execution trace by normalizing sets of instructions and comparing them to the normalized reference trace. Our search resulted in a unique match to the fragment of code shown in Figure 5d. As seen, the code fragment consists of different instructions which accomplish the same result. This is due to instruction permutations that were present in the sample we downloaded. However, as seen from Figures 5e and 5f, the normalized version of this trace and the normalized version of the reference trace result in a match. We also confirmed the overwriting of the NtOpenProcess API by looking at the execution trace which logged the code page due to the write.

Using the above technique on various reference traces describing the Haxdoor malware [20], and knowing that the information recorded by XTRec is accurate and uncompromised, we can determine that *Haxdoor.KI* (or an equivalent) was therefore executed in the host.

7 Related Work

In this section we discuss related work in the area of execution logging. Current approaches in the area of execution logging can be broadly divided into three categories: deterministic replay, OS level logging, and debuggers and specialized hardware.

Deterministic Replay: In deterministic replay approaches, a technique called VM introspection is employed, whereby general-purpose VMMs are extended to deliver events during guest execution. These events are recorded and replayed at a later time in order to reconstruct the entire system state. To reduce the time taken to replay to a particular state, deterministic replay systems introduce the notion of a checkpoint which saves the complete system state (including memory contents) as a snapshot. Replay requires time that is proportional to the length of time the system has been running since the last checkpoint.

Aftersight [8] employs VMWare [36] to decouple analysis from normal execution by logging non-deterministic VM inputs and replaying them on a separate analysis platform for intrusion detection and bug detection. ReVirt [11] employs UML Linux [10] for intrusion detection and replay. It logs deterministic events such as interrupts and system calls, and is able to replay these events thereby replaying the execution control flow. Flight data recorder [38] extends this scheme to multiprocessors. Other intrusion detection tools such as Backtracker [21] and Introvirt [18] also work on the same principle and employ ReVirt as their base. VMIIDS [15] and Psyco-Virt [1] are some more examples of systems employing VM introspection for intrusion detection. Time Traveling Virtual Machine (TTVM) [22] logs execution control flow and uses it to detect bugs in a OS Kernel.

With deterministic replay, a sequence of events are replayed for a significant period of time to reconstruct a given system state. Such an approach defeats the goal of forensic analysis where minimizing time is of utmost importance. While checkpoints can be used to minimize replay time, they are heavyweight and unsuitable for server platforms.

Deterministic approaches also need to store all non-deterministic data in order to generate a particular system state. For network and disk accesses, this incurs a large overhead in terms of runtime and disk space, especially for server environments. As examples ReVirt incurs runtime overheads of 1.2x–1.8x. Aftersight uses ReTrace [39], a deterministic replay system built on VMWare and reports a small runtime overhead of 5%. However, ReVirt is only targetted at desktop environments and assumes network and disk data to be very minimal and to the best of our knowledge there is no evaluation of network and disk logging for ReTrace for real world server deployments. Further, it is prohibitively slow to capture *all* forms of execution non-determinism (e.g., device states accessed via I/O locations and device interrupts) in current commodity platforms [29].

Finally, VMMs employed in current deterministic replay approaches are complex and have a large trusted computing base. An attack exploiting a vulnerability within the VMM[14] or any of the host OS components (in case of

```
Mov eax, [eax+Client_ID.UniqueProcess]
Mov ecx, dwProtectedProcCount
Mov edi, offset pProtectedProcBuffer
Repne scasd
Or ecx, ecx
Jz do_nothing
Cmp eax, dwBackdoorProcId
Jz do_nothing
Mov eax, fs:18h
Mov eax, [eax+NT_TEB.Cid.UniqueProcessId]
Cmp eax, pProtectedProcBuffer
Jz deny_operation
Cmp eax, pProtectedProcBuffer+4
Jz deny_operation
Mov ecx, [esp+4+ClientID]
Mov [ecx], eax
```

(a)

```
R01 = [R01+Client_ID.UniqueProcess]
R03 = [dwProtectedProcCount]
R06 = pProtectedProcBuffer
X: tmp = [r06] – r01; ZF= (tmp ?1: 0)
Jump (ZF=1) Y
R06 = r06 + 4
R03 = R03 -1
Jump X
Y: R03 = R03 | R03 ; ZF = (R03?1:0)
Jump (ZF=1) do_nothing
Tmp = R01 – [dwBackdoorProcId]; ZF = (tmp?1:0)
Jump (ZF=1) do_nothing
R01 = [FS:18h]
R01 = [r01 + NT_TEB.Cid.UniqueProcessId]
Tmp = R01 – pProtectedProcBuffer; ZF=(tmp?1:0)
Jump (ZF=1) deny_operation
Tmp = R01 – pProtectedProcBuffer-4; ZF=(tmp?1:0)
R03 = [r08 + 4 + ClientId]
[R03] = R01
```

(b)

```
R01 = [R01+Client_ID.UniqueProcess]
R03 = [dwProtectedProcCount]
R06 = pProtectedProcBuffer
X: tmp = [r06] – r01; ZF= (tmp ?1: 0)
Jump (ZF=1) Y
R06 = r06 + 4
R03 = R03 -1
Jump X
Y: R03 = R03 | R03 ; ZF = (R03?1:0)
Jump (ZF=1) do_nothing
Tmp = R01 – [dwBackdoorProcId]; ZF = (tmp?1:0)
Jump (ZF=1) do_nothing
R01 = [[FS:18h] + NT_TEB.Cid.UniqueProcessId]
Tmp = R01 – pProtectedProcBuffer; ZF=(tmp?1:0)
Jump (ZF=1) deny_operation
Tmp = R01 – pProtectedProcBuffer-4; ZF=(tmp?1:0)
R03 = [r08 + 4 + ClientId]
[R03] = R01
```

(c)

```
Mov ebx, [ebx+Client_ID.UniquePRocessId]
Mov ecx, [20102000h]
lea esi, [20101000h]
X: Cmp [esi], ebx
Je Y
Add esi, 4
Loop X
Jmp 200105A
Y: cmp ebx, 20002000h
Jz 2000105A
Mov edx, fs:18h
Mov eax, [edx+NT_TEB]
Mov ebx, [23001020h]
Cmp eax, ebx
Jz 2000112B
Add ebx, 4
Cmp eax, ebx
Jz 2000112B
Mov ebp, [esp+4+clientID]
Mov [ebp], eax
```

(d)

```
R02 = [r02 + Client_Id_UniqueProcess]
R03 = [20102000h]
R05 = 20101000h
X: tmp = R05 – R02; ZF = (tmp?1:0)
Jump(ZF=1) Y
R05 = R05 + 4
R03 = R03 -1
Jump(Zf=0) X
Jump do_nothing
Y: tmp= R02 – 20002000h; ZF=(tmp?1:0)
Jump(Zf=1) 2000105A
R04 = [Fs:18h]
R01 = [R04 + NT_TEB.Cid.UniqueProcessId]
R02 = [23001020h]
Tmp = R01-R02; ZF=(tmp?1:0)
Jump(ZF=1) 2000112B
R02 = R02 + 4
Tmp = R01 – R02; ZF = (tmp?1:0)
Jump(ZF=1) 2000112B
R07 = [R08 + 4 + ClientID]
[R07] = R01
```

(e)

```
R02 = [r02 + Client_Id_UniqueProcess]
R03 = [20102000h]
R05 = 20101000h
X: tmp = [R05] – R02; ZF = (tmp?1:0)
Jump(ZF=1) Y
R05 = R05 + 4
R03 = R03 -1
Jump X
Y: R03 = R03 | R03 ; ZF = (R03?1:0)
Jump (ZF=1) 2000105A
tmp= R02 – 20002000h; ZF=(tmp?1:0)
Jump(Zf=1) 2000105A
R01 = [[Fs:18h] + NT_TEB.Cid.UniqueProcessId]
Tmp = R01-[23001020h]; ZF=(tmp?1:0)
Jump(ZF=1) 2000112B
Tmp = R01 – 23001020h - 4; ZF = (tmp?1:0)
Jump(ZF=1) 2000112B
R07 = [R08 + 4 + ClientID]
[R07] = R01
```

R07 <-> R03, pProtectedProcBuffer <-> 23001020h, dwProtectedProcCount <-> 20102000h, do_nothing <-> 2000105A, dwBackdoorProcId <-> 23001020h

(f)

EAX=R01, EBX=R02, ECX=R03, EDX=R04, ESI=R05, EDI=R06, EBP=R07, ESP=R08

Figure 5: Using code normalization on execution traces recorded by XTRec to detect the presence of Haxdoor.KI on a host: (a)–(c) Reference trace, its high-level trace representation and corresponding normalized representation and (d)–(e) Execution Trace from XTRec, its high-level trace representation and corresponding normalized representation. The normalized representations of the reference trace and execution trace from XTRec match, thus showing that Haxdoor was executed on the host.

a hosted VMM) can result in the integrity of the recorded trace being compromised. While, the past log can be prevented from being tampered with [3], nothing prevents the attacker from inserting new entries and deleting the log.

OS Level Logging: In OS level logging, the host OS is modified to trigger messages during the operation of the OS Kernel. Taser [16] is an intrusion recovery system that captures and transmits system calls, their parameters, and their return values across a private network interface to a secure, append-only, backend system. Syslogd [30] provides support for system logging on Unix-based systems. Filemon [28] and Regmon [28] are system utilities which log real-time file and registry activities under Windows. Tools such as Valgrind [25], and DynamoRIO [5] can run a specified program within an OS using dynamic binary translation and can be used for instruction-level execution logging.

OS level logging, in addition to being coarse-grained, suffers from the following drawbacks: (a) the log is stored within the host OS, which, as discussed previously, cannot guarantee log integrity, and (b) the framework itself resides within the host OS which makes it easily susceptible to attacks. Recent approaches such as SIM [32] combine OS level logging with VMM based memory protections for efficient system call logging. While such an approach can protect the monitoring framework from attacks, an attacker can still manipulate the OS data being logged.

Debuggers and Specialized Hardware: Debuggers such as WinDbg [24] allow recording of execution control flow by single-stepping instructions. Specialized hardware such as Logic Probes and In-Circuit Emulators (ICE) can be configured to read processor BTMs from the system bus as demonstrated by Bosch et al. [4]. However, these approaches are designed for manual debugging and are not suited for real-time logging and online deployment. CADRE [29] is a cycle accurate deterministic replay system that can accurately recreate the execution control-flow of a system. However, it uses a hardware platform that is very different from commodity systems. Software cycle accurate deterministic simulators such as PTLSim [40] and AMD SimNow [2] are very slow (typically 50 to 1000x slowdown) and do not contain adequate support for commodity hardware making them unsuitable for online deployments.

8 Conclusions

With the rapid creation of new malware, XTRec offers the useful property to perform *fast forensic analysis a posteriori*. Based on our experimental results we find that XTRec is viable on current enterprise systems. We explored a debugging feature that is present on all x86 CPUs, namely Branch Trace Messages (BTM) and found that it can be used to show whether a particular set of code *has been* executed on a system, or conversely to prove that some code *has not* executed, a highly desirable property to ensure information assurance, especially in critical e-government infrastructure. In the process, we found that current hardware implementation of BTMs do not target any form of optimization and their documentation is sparse to non-existent. Indeed, we hope that BTM applications such as the one demonstrated in our paper will drive CPU vendors to be open about BTM functionality and make necessary hardware changes to improve BTM performance.

References

[1] F. Baiardi and D. Sgandurra. Building trustworthy intrusion detection through vm introspection. In *IAS 2007*.

[2] R. Bedichek. Simnow: Fast platform simulation purely in software. *HotChips: A Symposium on High Performance Chips*, August 2004.

[3] M. Bellare and B. Yee. Forward integrity for secure audit logs. *Technical Report, CSE Department, University of California at San Diego*, November 1997.

[4] P. Bosch, A. Carloganu, and D. Etiemble. Complete x86 instruction trace generation from hardware bus collect. In *23rd EUROMICRO Conference*, pages 402–408, 1997.

[5] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation.* PhD thesis, MIT, 2004.

[6] D. Bruschi, L. Martignoni, and M. Monga. Using code normalization for fighting self-mutating malware. In *International Symposium on Secure Software Engineering*, pages 37–44, 2006.

[7] D. Bruschi, L. Martignoni, and M. Monga. Code normalization for self-mutating malware. *IEEE Security and Privacy*, 5(2):46–54, 2007.

[8] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX Annual Technical Conference*, pages 1–14, 2008.

[9] O. Computing. Community malicious code research and analysis. *http://www.offensivecomputing.net*.

[10] J. Dike. Uml, http://user-mode-linux.sourceforge.net/.

[11] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, 2002.

[12] T. Espiner. Swedish bank hit by biggest ever online heist. *ZDNet News Archives http://news.zdnet.co.uk/security/*, January 2007.

[13] F-Secure. Haxdoor.ki. *F-Secure Trojan Information*, Aug '06.

[14] P. Ferrie. Attacks on virtual machine emulators. *Symantec Advanced Threat Research*, January 2007.

[15] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *NDSS*, 2003.

[16] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara. The taser intrusion recovery system. In *ACM SOSP*, 2005.

[17] Intel Corp. *IA-64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide*, 253668-026us edition, February 2008.

[18] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *SOSP '05*, pages 91–104.

[19] P. Karger and D. Safford. I/o for virtual machine monitors: Security and performance issues. *IEEE Security and Privacy*, 6(5):16–23, 2008.

[20] K. Kasslin. Kernel malware: The attack from within. *Association of Anti Virus Asia Researchers (http: www.aavar.org)*, December 2006.

[21] S. T. King and P. M. Chen. Backtracking intrusions. In *SOSP '03*, pages 223–236.

[22] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX Annual Technical Conference*, 2005.

[23] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In *USENIX Security Symposium*, 2008.

[24] Microsoft Corp. *The Windows Debugger, http://www.-microsoft.com/whdc/DevTools/Debugging/default.mspx*.

[25] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07*, pages 89–100.

[26] B. Parno. Bootstrapping trust in a trusted platform. In *USENIX Workshop on Hot Topics in Security*, 2008.

[27] A. Phansalkar, A. Joshi, and L. K. John. Analysis of redundancy and application balance in the spec cpu2006 benchmark suite. In *ISCA '07*, pages 412–423.

[28] M. Russinovich and D. Solomon. *Microsoft Windows Internals (4th Edition): Microsoft Windows Server 2003, Windows XP, and Windows 2000.* Microsoft Press, 2004.

[29] S. R. Sarangi, B. Greskamp, and J. Torrellas. Cadre: Cycle-accurate deterministic replay for hardware debugging. In *DSN'06*, pages 301–312.

[30] M. Schwarz. The system logging daemons, syslogd and klogd. *Linux Journal*, July 2000.

[31] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *SOSP*, 2007.

[32] M. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-vm monitoring using hardware virtualization. In *ACM Conference on Computer and Communications Security*, 2009.

[33] SpywareGuide. A311 death. August 2004.

[34] E. P. Traut, M. D. Hendel, and R. A. Vega. Enhanced shadow page table algorithms. *United States Patent 7650482*, 2010.

[35] VMware Communities. ESX 3.5 or Xen 4.1? `http://communities.vmware.com/message/900657`, 2008.

[36] VMware Inc. *Understanding Full Virtualization, Paravirtualization and Hardware Assist*, November 2007.

[37] Xen.Org. Xen 3.3 feature: Shadow 3. `http://blog.xen.org/index.php/2008/08/27/xen-33-feature-shadow-3/`.

[38] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA '03*, pages 122–135.

[39] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weiss-man. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Proc. 2007 Workshop on Modeling, Benchmarking and Simulation (MoBS)*, 2007.

[40] M. T. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *ISPASS '07*.