XTREC: Secure Real–time Instruction-level Control Flow Recording on Commodity Platforms

Amit Vasudevan, Ning Qu, Adrian Perrig

March 17, 2009

CMU-CyLab-09-007

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

XTREC: Secure Real–time Instruction-level Control Flow Recording on Commodity Platforms

(A video of XTREC in action can be found at https://cid-bd9d14a3cac05810.skydrive.live.com/browse.aspx/XTRECVideo, username: xtrec@live.com, password: 01092009)

Amit Vasudevan
CyLab/CMU

Ning Qu
CyLab/CMU

Adrian Perrig
CyLab/CMU

Abstract

We propose a primitive that can record the execution control flow information of a system with 100% accuracy. Furthermore, our primitive is robust to compromise which provides integrity of the control flow log. We implement the primitive on the AMD Secure Virtual Machine uniprocessor platform running the Windows 2003 Server OS. The only software component that is trusted in the system during runtime is *XTREC* itself, whose security sensitive portion is less than 1100 lines of code which makes it amenable to formal verification to ensure security and safety. Our experimental results show that our framework latency is minimal for realistic applications and that the approach is viable for enterprise settings.

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 use this vulnerability to compromise his/her systems; and if yes, what operations did the attackers perform?

Instruction-level control flow logging helps answer these questions. Given a log of the exact execution control flow of an entire system, all operations can later be reconstructed. Such information can even be used to demonstrate that some operation *did not happen*, which is a highly desirable property to ensure information assurance.

To illustrate these points, we consider the Haxdoor malware [15] which compromised over 60% of systems in Europe in 2006–2007 resulting in massive data theft and financial loss [14]. The attacks made use of the A-311 Death Backdoor [32] which enables a remote attacker to take complete control of a system, perform attacks, and remove itself completely leaving no trace 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, resulting in the loss of over a million dollars [14]. Instruction-level control flow logging 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 instruction-level control flow logging, 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 large-scale design changes to existing operating systems or applications.

We propose a framework codenamed *XTREC*, which logs the control flow of the entire system (including the OS and applications) *in real-time at the instruction-level*. Furthermore, *XTREC* also collects important data from the OS to *annotate* the log at a coarser level (such as an executable file name, kernel driver or the memory region a set of control flow instructions belongs to). *XTREC* features 100% accuracy, while providing robustness against subversion and integrity of the control flow log.

We implement *XTREC* as a tiny hypervisor on the AMD Secure Virtual Machine (SVM) platform running the Windows 2003 Server OS. *XTREC* uses the Branch Trace Message (BTM) feature (available in all x86-class processors) in addition to secure software-based annotations to provide accurate fine-grained execution control flow recording. Our system uses a trusted logging interface on the target system where it is deployed (typically a gigabit Network Interface Card (NIC) on a PCI-

Express Bus) and stores the log on a trusted system (typically a storage area network or disk array). Furthermore, *XTREC* employs hardware virtualization features and DMA protection which are built into current processors and chipsets to defend itself against any form of attack.

This paper makes the following contributions: (a) we design and implement a real-time instruction-level control flow logging mechanism which is 100% accurate while simultaneously ensuring log integrity; (b) high-level annotations separate control flow information into individual memory regions as perceived by the OS during execution; (c) minimal code size facilitates formal verification; (d) minimal OS Kernel interface requires minimal changes to the OS Kernel; (e) we show the efficacy of the framework by analyzing the Haxdoor/A-311 attack and show how *XTREC* can be used to prove that the system was compromised.

2 Assumptions and Threat Model

In the following description, we mark the requirements that we can relax with "*"; in Section 6.2 we then describe how they can be relaxed.

We assume the following hardware: x86-class uniprocessor, hardware support for CPU and physical memory virtualization, hardware support for attestation*, hardware support for branch trace messages, storage area network for logging purposes, reliable high–speed network (optical fiber/EMI–resistant ethernet), and a gigabit NIC.

We furthermore make the following trust assumptions: processor features (hardware virtualization, hardware attestation, branch trace messages and physical memory virtualization) that our security relies on are free of vulnerabilities, pre-boot stage including the BIOS is not malicious*, BIOS only accepts signed updates*, BIOS does not allow third party SMM handlers*, gigabit NIC is not malicious, and *XTREC* is free of vulnerabilities. Given the dramatically reduced TCB of *XTREC* compared to previous work, we argue that this last trust assumption is much easier to achieve for us than for previous works. We further discuss this point in Section 7.

Finally, we assume that the OS Kernel does not modify itself during runtime, but this assumption is not needed for any other privileged or unprivileged code (kernel modules, kernel libraries, applications, etc.). We elaborate on this assumption in Section 3.7.2.

We consider a system which can execute any form of code in either real or protected mode with or without virtual memory. The code executing in the system (apart from the OS Kernel) can be of any nature (polymorphic, obfuscating, self-modifying etc.) and can execute at any privilege level.

We consider an attacker without physical access to tamper with the CPU, chipset, memory controller, memory or the trusted NIC–which form our hardware TCB. The attacker can use any method to take control of the system (*XTREC* is assumed to be secure) either locally or remotely. As a result, the attacker can execute any arbitrary code within the system.

3 Design

In this section, we present the design of *XTREC*. We start by describing our design principles and goals. We then give an overview of *XTREC*, followed by the detailed design of its components.

3.1 Principles and Goals

We aim to achieve the following design principles. (a) Ensure execution control flow logging at the instruction–level, (b) Ensure complete accuracy of information logged (the execution control flow information logged is *exactly* what took place on a system), and (c) Ensure integrity of logged information. Based on these principles, our goals are to: (i) provide a control flow logging mechanism that is suitable for online deployment in untrusted systems, (ii) ensure that the framework always starts during system startup and always operates during the lifetime of the system execution, (iii) be able to classify the logged information at a virtual address space level, (iv) be able to classify the code regions within a virtual address space into those that are OS allocated (process, arbitrary memory region, device driver, user or kernel mode library) vs. direct manipulation of system structures, (v) keep the code size of *XTREC* small to make it amenable for formal verification, and (vi) keep the changes to the OS Kernel minimal enabling ease of portability and keeping the interface of *XTREC* to the OS Kernel minimal to reduce the attack surface.

3.2 Overview

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

Figure 1 shows the position where *XTREC* resides in a host and its internal architecture. *XTREC* resides at the lowest level between the underlying hardware and the host OS. The framework consists of a Loader and a Runtime. The Loader is only used during system startup and is discarded thereafter.

The Loader gets control during system startup from a trusted pre–boot stage (e.g., TCG compliant BIOS, Secure Boot, etc.). Thus, we can guarantee that *XTREC*'s

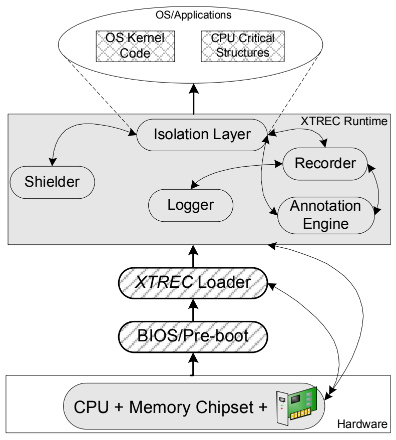


Figure 1: *XTREC* Architecture (Greyed portions represent Trusted Components during both startup and runtime, Shaded portions represent Trusted Components only during startup and Cross–Hatched portions represent Untrusted Components that are Write–Protected)

Loader always gets control when the system is powered up. *XTREC*'s Loader then employs hardware attestation (e.g., TPM–based attestation) to establish a dynamic root of trust (as a result also ensuring that it is running in its unmodified form). It then initializes the CPU, chipset, memory controller and the trusted NIC and loads the Runtime into memory and protects it against any modifications (such as via DMA). Finally, it hashes the Runtime to verify the integrity of its code and data, and transfers control to it.

The Runtime is composed of an Isolation Layer, a Control Flow Recorder, an Annotation Engine, a Logger and a Shielder. The Isolation Layer isolates the execution of the host OS from *XTREC* and its components and ensures that *XTREC* has complete control over the host at all times. The Control Flow Recorder is responsible for recording the execution control flow of the host system at an instruction level. The Annotation Engine inserts details which helps in giving a high–level connotation to the recorded control flow information. The Logger is responsible for transmitting the recorded information to the log store, while the Shielder protects *XTREC* from subversion. The *XTREC* Runtime initializes itself, deallocates all memory consumed by the Loader, and begins execution of the host OS boot code. The following sections describe each component of *XTREC*'s Runtime in detail. Note that even though the Isolation Layer and the Shielder may seem to do a similar task, we keep the Isolation Layer separate to aid in porting our system to various hardware virtualized architectures (e.g., Intel VT).

3.3 Isolation Layer

Our TCB consists of the CPU, chipset, memory controller, memory and the trusted NIC used for transmission. *XTREC* needs to be isolated from the host OS so that it can record the control flow of the host, while simultaneously defending against any attacks potentially originating from a compromised OS. Based on our TCB, we rely on CPU-based protections to provide this isolation. CPU-based protections are based on privilege levels whereby higher privilege code can modify its own protections as well as the protections of lower privileged code. Usually, the OS Kernel is the highest privileged component in the system. Thus, *XTREC* must execute at a higher privilege level than the OS Kernel itself. We use the hardware virtualization features built into commodity processors to execute at the privilege level of a Virtual Machine Monitor (VMM).

The Isolation Layer component of *XTREC* acts as a bridge between the host OS environment and *XTREC*'s internal components for various events that occur inside the host OS that need intervention by *XTREC*. The Isolation Layer component also ensures that such events are first delivered to *XTREC*, which in turn can choose to let the host OS see the event if needed. The Isolation Layer ensures that all operating modes of the host OS are run with hardware–assisted physical memory virtualization. This ensures that any address translation inside the OS resulting in a physical memory address can be processed by *XTREC* before it is passed onto the memory controller. This in turn enables the framework to set desired protections on physical memory regions within the host OS without having to manipulate the host OS paging structures.

3.4 Control Flow Recorder

Our design goal with instruction–level control flow recording is not to modify any code executing in the host in any fashion so as to support any form of commodity code. As a solution, we could employ dynamic binary translation within our hypervisor to execute the host OS and any code within it and track control flow instructions. However, this solution leads to a huge increase in our framework code size making it more susceptible to vulnerabilities.

Instead, we use the processor–supported BTM mechanism for instruction–level control flow recording. BTMs, supported by all x86–class of processors. BTMs are emitted by the processor for every branch (conditional or unconditional) that is taken by the CPU. These include conditional jumps, unconditional jumps, loops, procedure invocations, returns from procedures, interrupts, exceptions, and return from interrupts and exceptions. The BTMs are usually sent out on the system bus, but the processor can also be configured to send the BTMs to

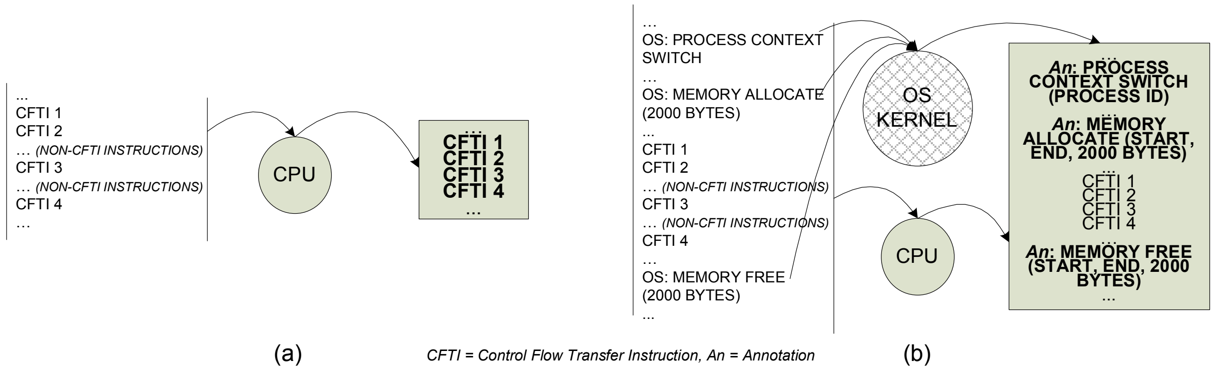

...
CFTI 1
CFTI 2
... (NON-CFTI INSTRUCTIONS)
CFTI 3
... (NON-CFTI INSTRUCTIONS)
CFTI 4
...
CPU
...
CFTI 1
CFTI 2
CFTI 3
CFTI 4
...
(a)
CFTI = Control Flow Transfer Instruction, An = Annotation
...
OS: PROCESS CONTEXT SWITCH
...
OS: MEMORY ALLOCATE (2000 BYTES)
...
CFTI 1
CFTI 2
... (NON-CFTI INSTRUCTIONS)
CFTI 3
... (NON-CFTI INSTRUCTIONS)
CFTI 4
...
OS: MEMORY FREE (2000 BYTES)
...
OS KERNEL
CPU
An: PROCESS CONTEXT SWITCH (PROCESS ID)
An: MEMORY ALLOCATE (START, END, 2000 BYTES)
...
CFTI 1
CFTI 2
CFTI 3
CFTI 4
An: MEMORY FREE (START, END, 2000 BYTES)
...
(b)

Figure 2: *XTREC* Control Flow Recording: (a) Using CPU Branch Trace Messages for Instruction-level Control Flow Recording, and (b) Using OS Kernel to add Annotations for high–level details. (Greyed Areas represent Trusted Components while Cross–Hatched Areas represent Untrusted Components that are Write–Protected)

system physical memory. Figure 2a shows a fragment of code in memory and the corresponding BTMs generated by the processor.

BTMs occur irrespective of the operating mode or privilege level of the CPU. This is crucial to our design, since we do not need to modify any code executing in the host to enable recording control flow instructions. 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 such as self–modifying, polymorphic or obfuscated, code which exploits processor specific microarchitectural features (e.g., branch prediction), etc. Also, the BTM feature is controlled with a very small set of processor registers which helps keep our framework code base tiny. Note that we do not allow the OS kernel to be self–modifying. This preserves the integrity of Annotators that are deployed within the OS Kernel as explained in Section 3.7.2.

3.5 Annotation Engine

It is very difficult to perform log analysis by solely relying on instruction–level control flow recording. While we can obtain memory addresses of control flow transitions from the log, it would be beneficial if the log contains additional information such as the executable file, library, driver or memory region (process space, user or kernel stack, kernel memory pool etc.) the memory addresses belong to. This is important since an OS can have various processes that are executed concurrently and can dynamically map the same executable, library or driver to a different memory range during execution. Having such high level information eases the process of log analysis by separating the control flow into individual memory regions as perceived by the OS during execution. We rely on the host OS kernel to provide us with such information. However, we do not trust the OS Kernel in any fashion to keep our TCB minimal.

3.5.1 Annotators and Annotator Control Flow Set

We modify the host OS kernel and insert what we term *Annotators* at appropriate locations. An Annotator qualifies a range of allocated memory by associating it with an entity which can be a process, kernel driver, kernel library, user library, dynamic memory allocation or stack. It gathers appropriate information (*Annotation*) about the entity such as process, driver or library names, memory base and size when control flow reaches the point where it is inserted and transfers that data to *XTREC*.

An annotator is split into three parts. The first part is inserted at the beginning of the top–level function that is responsible for creation and deletion of the entities and grabs the input parameters (annotation-data). The second part is inserted in the locked operation of actually updating the kernel data-structure for the entity and the third part is inserted at the end of the top–level function before it successfully returns.

An *Annotator Control Flow Set* (ACFS) is a set of all pre–computed legitimate control flow transitions from the entry to the top–level function containing the Annotator until its return for a particular OS Kernel build.

3.5.2 Bypassing Annotations

Since the host OS is untrusted, an attacker can bypass an Annotator for a particular operation. As an example, let us consider the operation of allocating a range of memory for executing code. There are two ways in which an attacker could allocate memory while bypassing Annotators: (i) the attacker could allocate the range of memory by directly manipulating the host OS paging structures, and (ii) the attacker could use the host OS to allocate a region of memory for code execution, by executing a copy of the host OS code within another memory region until the Annotator.

While the attacker can manage to bypass an Annotator, he/she cannot prevent control flow instructions that are recorded when code is executed within a memory region (see Section 3.4). Thus, during log analysis, *XTREC* will not find any Annotations corresponding to the memory regions of the control flow instructions. *XTREC* will then classify the memory regions as being allocated via direct manipulation as was the case (the attacker directly manipulated the OS paging structures in (i) and executed a *copy* of the OS code in (ii)).

3.5.3 Inserting Malicious Annotations

An attacker can insert malicious annotations by manipulating annotation–data either through code or via DMA. Using code, an attacker can jump directly to an Annotator or exploit a control flow bug in the host OS Kernel code resulting in an arbitrary Annotator being executed. Note that an attacker cannot directly modify the host OS Kernel to change Annotators (see Section 3.7.2). Using DMA an attacker can use a device to write to a specified memory region without employing the CPU.

It is important that such malicious or faulty annotations be removed since it can lead to inaccurate analysis. As an example, consider a legitimate annotation which records details about a process creation. Consider, a malicious annotation which records the exact same details, but changes the process name. Thus, during analysis there would be no way to tell if a given set of control flow transitions belonged to the legitimate process or to the process recorded by the malicious annotation.

XTREC handles DMA–based malicious annotations by using hardware DMA-protections. It marks all other memory–regions in the host OS non–DMA except the DMA physical–memory pool. Code–based malicious annotations are handled during log analysis. A legitimate annotation is one which has all three annotator parts in the log and whose control–flow does not deviate from the ACFS encompassing the three parts and whose first and second part have the same annotation–data.

3.6 Logger

XTREC's logger is responsible for sending the recorded control flow information to the log store. Our design is such that recording and logging can proceed simultaneously in the host in real–time. We use a fast gigabit NIC for transmission purposes. We also use Direct Memory Access (DMA) transfers to transfer data between the host memory to the log store using the trusted NIC. DMA transfers occur directly between the host memory chipset and the device and do not involve the CPU.

Our design uses two types of memory buffers. A *Record Buffer* is a region of memory that *XTREC* uses to record control flow information in a host. Once a record buffer is full, it is marked as a *Log Buffer* and recording continues at the next available record buffer. If there are no record buffers remaining, the logger polls the trusted NIC (polling phase) until it gets the signal that the previous DMA transfer was successful. Once it gets the signal, it marks that log buffer as a record buffer. When a buffer is marked a log buffer, a DMA transfer is simultaneously initiated between the trusted NIC and the buffer in the host. Our experimental results show that current gigabit NICs are fast enough not to cause any latency in the polling phase.

3.7 Shielder

Figure 3a shows the possible points of attack against *XTREC*. These include the host OS kernel code, processor critical structures managed by the host OS, CPU, memory, trusted NIC registers and *XTREC* itself. The shielder component of *XTREC* protects against these attacks and ensures the persistence and integrity of control flow recording, annotators, logging and *XTREC* memory regions.

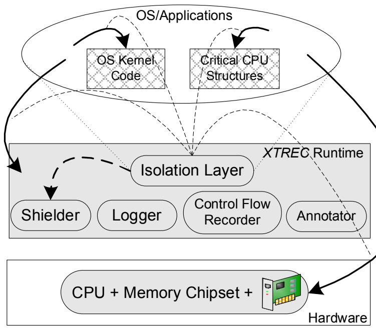
3.7.1 Protecting Control Flow Recording

The execution control flow recording mechanism used by *XTREC* relies on processor BTMs. The BTM feature is controlled by certain processor registers and events. The shielder intercepts any access to such registers and events via the Isolation Layer and denies access to them (usually simply returning back to the code). Further, we assume that processor–supported BTMs are free of vulnerabilities and that *XTREC* gets control from a trusted pre-boot stage, verifies the integrity of its code and operates without any flaws. Thus, we can ensure that control flow recording is always in effect and is uncompromised.

3.7.2 Protecting Annotators

To ensure that Annotators are always in effect and are uncompromised we need to ensure the following: (a) the host OS Kernel with Annotators always gets control during host startup, (b) the host OS Kernel always remains in control, and (c) the host OS Kernel is not modified in any form. Cases (a) and (b) address the situation where an attacker can install his/her own OS Kernel without Annotators or use malicious Annotators. Case (c) addresses the situation where an attacker can modify the existing OS Kernel to achieve the same effect.

Ensure Host OS Kernel with Annotators always gets control during Host Startup: If the host OS Kernel code is the first to get control when *XTREC*'s Runtime runs the host OS for the first time (e.g., Linux), we can protect the OS Kernel code from any modifications, verify its integrity and transfer control to it. However, for some OSes (e.g., Windows) the OS startup process consists of intermediate steps before the actual OS Kernel gets control. So we adopt a more general design. Fig-



(a)

```
ENSURE_OSKERNEL_GETS_CONTROL(codeRegionList){
  startCodeRegion =
      getFirstElement(codeRegionList);
  do{
    nextCodeRegion =
      EXECUTE_CODEREGION(startCodeRegion);
    if (nextCodeRegion is not within
         codeRegionList)
     return ERROR
    else if (nextCodeRegion is
           osKernelCodeRegion)
     return SUCCESS
  }while(TRUE);
}

EXECUTE_CODEREGION(codeRegion){
  writeProtectAndVerify(codeRegion);
  setExecuteDisable( totalMemoryRegions -
      codeRegion);
  setExecuteDisableExceptionHandler(
      EXECUTECODEREGION_cb);
  transferControlTo(codeRegion);
  EXECUTECODEREGION_cb:
    return (region that caused exception);
}
```

(b)

```
PROTECT_CPU_CRITICAL_STRUCTURES(
          criticalStructRegions){

writeProtectAndVerify(criticalStructRegions);
setWriteExceptionHandler(criticalStructRegion,
   PROTECT_CPU_CRITICAL_STRUCTURES_CALLBACK);
return;
}

PROTECT_CPU_CRITICAL_STRUCTURES_CALLBACK(){
  setReadWrite(criticalStructRegions);
  setSingleStepExceptionHandleR(SSHANDLER);
  resumeOS();
  SSHANDLER:

writeProtectAndVerify(criticalStructRegions);
    resumeOS();
}
```

(c)

Figure 3: *XTREC* Shielder: (a) Protects the Host OS Kernel Code and Critical Processor Structures in the Host OS Kernel Data along with its Trusted Computing Base (The solid lines denote access for modification while the dotted lines denote the transfer of control to *XTREC* for such accesses), (b) Ensures Host OS Kernel with Annotators always gets control during system startup and (c) Ensures Host OS Kernel with Annotators always remains in control during runtime by protecting Critical Processor Structures

ure 3b shows the pseudo-code of our design. We enumerate all the possible execution code areas and their physical memory ranges before the host OS Kernel gets control. We term these code areas the *boot–code areas* for the host OS. Before we transfer execution to the first code area in the boot-code areas, we write protect the code, hash the code area to verify its integrity and setup all other memory regions to generate a processor fault on execution. Thus, when the first code region transfers control to the next, it *will* generate a processor fault. At that point we repeat the same step for the next code region and continue doing so until the generated fault corresponds to the host OS Kernel code range—which signifies the host OS Kernel getting control. This design ensures that control flow cannot be subverted before the host OS kernel reaches its execution as executing any other code apart from the boot–code areas for the host OS will cause a processor fault which would otherwise not happen during normal execution.

Ensure Host OS Kernel with Annotators always remains in control during runtime: On the x86–class of processors, the CPU consults certain critical memory structures such as the IDT (for interrupt/system service handlers) and GDT (for code and data selectors) during the execution of OS Kernel code. An attacker, for example, could replace the host OS system service handler and the host OS scheduling interrupt handler (usually located within the IDT) to point to replacement handlers, effectively installing a new host OS Kernel in operation without Annotators. Thus, to ensure that the host OS Kernel always remains in control during runtime, we need to ensure that such CPU critical structures are protected.

We could compare contents of OS Kernel specific regions within the IDT and GDT to default values and write protect them once they are initialized by the OS Kernel during kernel initialization. However pocessor structures such as the IDT and GDT can be written to after their initialization by the OS Kernel (e.g., Windows). Thus, we adopt a more general technique to protect the processor critical structures (Figure 3c).

XTREC write–protects the processor critical structures at the start of the host OS Kernel execution. When there is a write to the structures, *XTREC* gets control via its Isolation Layer, marks the structures read-write, sets up the Isolation Layer to get control after a single instruction has been executed and resumes the host OS. The single instruction is executed with interrupts disabled to prevent any interrupts from being generated during the single–stepping. When *XTREC* gets control after the write instruction has been executed, it write protects the structures once again. *XTREC* then checks the structures for any inconsistencies and resumes the host OS normally. The consistency check compares locations within these structures that are essential for the functioning of the OS Kernel (scheduling interrupt, page–fault handler, system service handlers etc.) — after the write — to the default value for the host OS Kernel build in execution. Note that *XTREC* will always get control after the instruction performing the write irrespective of the instruction, since the Isolation Layer is always the first to get control on any event generated by the processor (in this case a single–step event).

Ensure that Host OS Kernel Code cannot be Modified: *XTREC* marks the OS Kernel code region as *READONLY* to prevent any modifications to Annotators or other parts of the OS Kernel. This does not affect the normal behavior of the OS Kernel, since as per our assumption the OS Kernel will not be modified in any fashion during normal execution. We could relax our assumption and allow modifications to the OS Kernel by trapping the write, and doing the write on the OS behalf (e.g., using single–stepping). However, the approach is not straightforward since the write could change an ACFS that we rely on to ensure Annotator persistence and to remove malicious annotations.

Note that the techniques presented in this section rely on write–protect and execute-disable features supported by the underlying hardware MMU. These features operate at a page granularity (usually 4Kb or 2MB) and hence require the target memory region to be aligned at a page and span a size that is a multiple of a page. This is not a limitation since we only perform such operations on the OS boot–code areas, OS Kernel code and the CPU critical structures which are in the OS Kernel data area. The modifications required are minimal as detailed in our implementation in Section 4.

3.7.3 Protecting Logging

To ensure that Logging is always in effect and is not compromised, we need to ensure that any access to the trusted NIC originates from the *XTREC* Runtime alone. To do this, the trusted NIC control and data locations (usually in memory mapped registers) are intercepted using the Isolation Layer to identify any access to the trusted NIC by the host OS. In such a case, a device not-existing message or a dummy value is returned when a device is probed or accessed at the specified location.

3.7.4 Protecting *XTREC* Memory Regions

To protect *XTREC* memory regions, we need to ensure that its code and data cannot be accessed directly by code executing in the host. At any given point the host OS can be in any of the three execution modes: Real, Protected without paging and Protected with paging (SMM mode is the same as real mode with access to 4GB of data). Hardware assisted memory virtualization is always in effect irrespective of the operating mode of the guest. *XTREC* changes the attributes in the hardware MMU paging structures classfying its range of code and data as inaccessible, thereby getting control if there is any access to its memory regions. Legitimate memory requests (memory requests that do not fall in the range of *XTREC*'s code and data) are directly handled by the hardware.

4 Implementation Details

In this section we discuss our *XTREC* prototype implementation on the AMD SVM uniprocessor platform running Windows 2003 SP1 OS in 32–bit mode. Features and details of the AMD SVM can be found in the AMD Developers Manual [2]. We use `GRUB` [18] as our boot–loader and load the *XTREC* Loader via `GRUB`. The *XTREC* Runtime is specified as a `GRUB` module. Once the *XTREC* Loader gets control, it executes the *SKINIT* SVM instruction which uses TPM–based hardware attestation in order to establish a dynamic root of trust. The *XTREC* Loader then relocates the Runtime above the physical memory range allocated to the host OS. The *XTREC* Loader then protects the complete Runtime from any writes, verifies its integrity, and if successful transfers control to the Runtime which in turn runs the host OS. We use the SHA-1 hash algorithm for verifying the integrity of code and data regions.

4.1 Isolation Layer

The *XTREC* Isolation Layer sets up a SVM Virtual Machine Control Block (VMCB) specifying the processor state upon bootup [19]. It then transfers control to the host OS boot–code within a Hardware Virtual Machine (HVM) using the VMRUN instruction. The Isolation Layer sets up the VMCB for the following intercepts: processor debug (#DB) exception, *VMMCALL* instruction execution, I/O data access to PCI address space, nested paging fault, and processor machine specific registers (MSR) associated with BTM. The Isolation Layer gets control via a *VMEXIT* from the HVM upon which it invokes the appropriate internal component depending on the exit condition, e.g., on *VMMCALL* instruction execution, it invokes the Annotation Engine to record the appropriate details. The Isolation Layer also sets up the SVM Device Exclusion Vector (DEV) bit vector and Nested Page Tables (NPT) to address the complete physical memory in the host.

4.2 Control Flow Recorder

On AMD CPUs, BTMs are managed using a set of 4 MSRs: *BTM Base* MSR, *BTM Limit* MSR, *BTM Ptr* MSR and *BTM Control* MSR. These MSRs are modified using the *RDMSR* and *WRMSR* instructions. Together the *BTM Base* and the *BTM Limit* MSRs specify the region of physical memory that the processor can use for storing BTMs, known as the *BTM Buffer*. The *BTM Ptr* MSR contains the physical memory address within the *BTM Buffer* where the next BTM will be stored. The *BTM Control* MSR is used to control the BTM generation and has an option called *Interrupt on BTM Buffer End*. When this option is set in the *BTM Control* MSR, the processor will generate a #DB exception when *BTM Ptr* MSR equals the *BTM Limit* MSR. The processor will also set the *BTM Control* MSR to stop BTM generation simultaneously.

Each BTM is a 96-bit datum which, includes the instruction pointer (segment/offset) value of the location of the control flow instruction and the type of control flow instruction (*UNCONDITIONAL BRANCH*, *CONDITIONAL BRANCH*, *INTERRUPT*, etc.). The first field of the datum is a tag or identifier which identifies the BTM. Though the exact values for the tags are processor–specific, the two common tags are *0x2* (instruction) and *0xD* (user-defined).

The Control Flow Recorder currently uses two 64MB BTM buffers for its operations. It initially marks both buffers as record buffers and sets the *BTM Buffer* to one of the record buffers. It then sets the *BTM Control* MSR to begin recording just before execution is transferred back to the OS via the Isolation Layer. The Recorder processes the #DB event from the Isolation Layer in order to switch between recorder and log buffers. Upon a #DB event, the Control Flow Recorder reads the trusted NIC status to see if it is currently in a DMA operation. If so, it polls the NIC status until the DMA operation completes, else it marks the current record buffer as the log buffer and marks the log buffer as the record buffer and initiates a DMA operation. It also sets the *BTM Base* MSR to point to the record buffer, sets the *BTM Control* MSR to begin recording and returns control to the host OS via the Isolation Layer.

4.3 Annotation Engine

The implementation of the Annotation Engine consists of two components: (i) the Annotators which are a part of the untrusted host OS Kernel, and (ii) the Annotation Engine which is a component of *XTREC*.

We used the Windows Research Kernel Sources in order to insert Annotators in the Windows Kernel. The Windows 2003 SP1 Kernel is an object–oriented kernel where every component of the OS (processes, files, etc.) is represented internally as an object. It is a pre–emptive kernel, and uses the timer interrupt for scheduling. The unit of scheduling is a thread. A process is an entity that consists of a group of threads and has a private virtual address space. The OS divides code into two domains: user–mode (least privileged) and kernel–mode (privileged).

We modify functions *MmMapViewOfSection*, *MmUnmapViewOfSection*, *MmCreateTeb*, trMmCreateKernelStack, trNtAllocateVirtualMemory and *MiDeleteVirtualAddresses* such that they transfer control to *XTREC* using the *VMMCALL* instruction. We pass parameters directly via processor registers for all Annotators. When the Annotation Engine gets control within *XTREC*, it uses the user–defined tag for BTM and records the Annotation within the current recorder buffer. It then increments the *BTM Ptr* MSR by the size of the Annotation (in bytes) aligned to 96–bits.

4.4 Logger

We use the Intel 82572EI gigabit NIC as our trusted NIC. It has provisions for a ring buffer–based transmit mechanism. It supports ring buffer–based descriptor mangement. The ring–buffer is composed of 16–byte entries with a maximum size of 1MB. Each 16–byte entry points to a data buffer that needs to be transmitted, which includes the data buffer size and other control information for the data buffer. The 82572EI has support for transmitting up to 9K bytes of data buffer per descriptor (called Jumbo Frames). The ring–buffer has a *head pointer* and a *tail pointer*. Once the ring–buffer descriptors are setup, the NIC control register can be written to in order to transmit all data buffers pointed to by the head pointer and tail pointer using a single DMA operation.

The Logger sets up the 82572EI's ring buffer to point to a total of 128MB of physical memory during initialization. This encompasses both the 64MB record buffers for *XTREC*. When the logger is used transmit a log buffer, it simply adjusts the tail pointer to point to the end of the appropriate 64MB block and writes to the control register to start the DMA.

4.5 Shielder

Protection of Control Flow Recording: *XTREC* sets up MSR intercepts in the VMCB for the *BTM Base*, *BTM Limit* , *BTM Ptr* and *BTM Control* MSRs. We notice that these MSRs are never accessed in current OSes (Windows and Linux), applications or debugging tools. Our system currently disallows any access to such MSRs upon the intercept triggering.

Protection of Annotators: The Windows boot process uses 3 *boot–sectors* that setup and transfer control to *ntldr* which is the Windows boot–loader. *ntldr* is made up of a real mode component and a protected mode component. The protected mode component is a regular portable executable. The real mode component does some initialization and transfers control to the protected mode component which then loads the actual Windows kernel. The protected mode component also invokes *ntdetect.com*, a real mode executable for hardware detection during bootup.

The *boot–sectors*, basically contain code to access the filesystem in order to read *ntldr* into memory. We discard the boot sector code and directly load *ntldr* as a module to *GRUB*. The *XTREC* loader then copies the *ntldr* module to its starting location (logical address *2000:0000* in real mode). We also modifed *ntldr* and *ntdetect.com* such that their code were page–aligned in memory. Since we did not have access to the sources of the boot phase of Windows 2003 SP1, the modification was carried out in binary. However, since the real mode portion of *ntldr* and *ntdetect.com* are 16–bit real mode executables, they use data segment relative addressing to address their data. This is carried out using a single *MOV* instruction at the start of their code. We padded the code regions of the real mode portion of *ntldr* and *ntdetect.com* with zeros to ensure they were page–aligned and modified the

MOV instruction to load the offset of the data segment accordingly. The protected mode portion of *ntldr* is a regular portable executable loaded at physical address *0x00400000* and adheres to page alignments for its code region. These modifications ensure that the boot phase of Windows conforms to our design as discussed in Section 3.7.2

We modified the Windows Kernel (*ntoskrnl.exe*) to include some initial code and also inserted page–aligned data for the GDT and the IDT. The initial code then copies the IDT given by *ntldr* (which is not page–aligned) into the page aligned IDT and sets up the IDTR to point to the new IDT using the *LIDT* instruction. This code also sets up the page–aligned GDT so that the first 3 entries (corresponding to the NULL selector, the kernel code and kernel data selectors) fall on a separate memory page while the rest of the original GDT (passed by *ntldr*) falls on another memory page. It then uses the *LGDT* instruction to load the page–aligned GDT. The initial code finally transfers control to *XTREC* via a VMMCALL.

At this point the *XTREC* runtime write–protects the physical address range of the code regions of *ntoskrnl.exe*, *hal.dll*, *bootvid.dll* and *kdcom.dll*, which collectively form the Windows kernel (and whose physical memory addresses are constant for a given kernel build) by setting the corresponding entries in the NPT to *READ-ONLY*. The runtime also DEV protects the OS kernel code and data regions. The runtime then hashes the Windows Kernel code and data regions and compares it against the stored hash within *XTREC*. If the comparison is successful, the runtime obtains the GDT and IDT physical addresses write–protects their memory pages. Note that our implementation splits the GDT into two memory pages and only write–protects the first memory page. This conforms to our design, and at the same time is very efficient, since Windows modifies GDT entries on every context switch. However, since the first three entries of the GDT are never modified, we never incur any runtime overhead due to GDT modifications.

Protection of Logging The Intel 82572EI Gigabit NIC uses the PCI space for access to its register. The PCI space is accessed using two I/O registers, the PCI address and the PCI data register. *XTREC* sets up an MSR intercept on the PCI data register. When the intercept triggers and control is transferred to *XTREC*, the PCI address register is examined to obtain information about the bus, device and function that is used to address the device. If it matches that of the trusted NIC, and if the access is for the *IDENTIFIER* offset in its PCI configuration space, we return the value *0xFFFFFFFF* to indicate no device at that location. For any other offset, we return the value *0x00000000* signifying uninitialized.

Protection of *XTREC* *XTREC* sets up the attributes of the memory pages corresponding to the physical memory region occupied by its code and data to *NOT-PRESENT* in the NPT. Thus, any access to *XTREC*'s code and/or data will result in a nested page fault exit, which is handled by the framework to disallow any form of access (read or write).

5 Evaluation

In this section we evaluate our implementation. We start by presenting our evaluation testbed. We then present details on the code size of our implementation and the modifications made to the OS Kernel. We then report the runtime performance of our system and follow that with a case study of the HaxDoor.KI malware.

5.1 Testbed

Our testbed consists of three components: a system with *XTREC* which forms the host, a system which forms the trusted log store and a gigabit switch and link to connect them together. The host is an AMD system running the Barcelona Stepping B3 Quad–Core CPU at 2.00Ghz (with multicore support turned off in the BIOS), 4GB DDR2 667 RAM, PCI-Express BUS, 250GB Hard disk with Windows 2003 SP1 in uniprocessor mode. We configured *XTREC* to use 256MB and the rest was used by Windows Kernel. We used an Intel 82572EI gigabit NIC as our trusted NIC on the PCI- express Bus. The system used to store the log is an Intel system running the Core 2 Duo at 2.66GHz, 4GB DDR2 667 RAM, 1TB Hard disk, Intel 82572EI gigabit NIC and Linux 2.6.23.1. The gigabit switch we used was Dell PowerConnect 5324.

5.2 Code Size and OS Modifications

In this section we present details on the code size and modifications made to the OS kernel and show that the sizes are within means of performing a manual and analytical audit on the codebase of *XTREC* to rule out potential vulnerabilities.

The *XTREC* codebase consists of: debug code (433 lines), header files (3540 lines), loader code (1307 lines) and runtime code (1047 lines). We use the sloccount [36] utility to count the source lines. The debug code is only used on test systems and is used to print debug messages to a serial console. It would not be used on a production system. The header files have C language declarations, constants and structures and have no executable code or macros that expand to code. The loader code is only used during the startup of the system after which it is discarded. The runtime code is responsible for control flow recording and logging. Most importantly, the security sensitive portion of *XTREC*, the runtime, is less than 1100 lines of executable code.

XTREC interfaces to the OS kernel via only 1 hypercall interface that is used to add Annotations to the log during execution. Also, the parameters passed to this interface are very well-defined which makes it possible for the framework to validate the arguments. Thus, the attack surface through the kernel is minimal. We modified the Windows 2003 SP1 Kernel to insert the *XTREC* hypercall interface to get information about process, thread and memory regions. There are 12 uses of the hypercall interface and we inserted less than 100 lines into the Windows Kernel.

5.3 Performance Measurements

We now report the runtime performance of *XTREC*. Our experiments consist of microbenchmarks and application benchmarks.

5.3.1 Microbenchmarks

XTREC adds overhead to system operation in four ways: (a) due to BTMs emitted by the processor, (b) due to annotators, (c) due to BTM buffer overflow, and (d) due to transmission of log buffer.

BTM latency depends on the CPU and the memory. To measure the BTM latency, we used a tight loop with a `CALL` instruction and measured the time before the `CALL` and at the start of the subroutine. The BTM Buffer overflow latency is the latency due to the triggering of a #DB exception. We wrote a small kernel driver which invokes the #DB exception and measured the round–trip time from the OS to *XTREC* and back. The Annotator latency is due to the `VMMCALL` instruction which transfers control to *XTREC*, coupled with inserting the corresponding Annotation. As discussed in Section 4.3, Annotators in *XTREC* fall into two categories: those that record parameters directly from registers and those that parse the OS paging structures in addition. We used the kernel driver to invoke an Annotator in a tight loop and measured the round–trip time for both categories. Similarly, we invoked the NIC transmission function by measuring the time before and after the successful DMA transmission of the record buffer to measure the latency of logging. We used processor clock cycles as the unit for our measurements and employ the `RDTSC` instruction to obtain the clock cycle count. Figure 4a shows the results of our microbenchmarks.

A point to note from the microbenchmarks is that, though the log buffer transmission incurs a huge clock cycle count, it only occurs once the recording buffer is full. Further, as we show in the next section, the DMA transfer always occurs in parallel with control flow recording for all our experiments, resulting in no perceptible latency.

5.3.2 Application Benchmarks

We hypothesize that when *XTREC* is used, the overhead of an application will be directly proportional to the number of branch instructions it uses and the number of times it uses the host OS to create processes, allocate memory, etc. Based on our hypothesis, I/O–bound applications that have very little control decisions will have the lowest overhead. On the other hand, compute–bound applications with high branching and memory allocations will have the highest overhead. Also, 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 compute–bound and I/O–bound applications. For our compute–bound applications, we choose benchmarks from the SPEC INT and FP suite. Our I/O bound applications consist of Postmark (with 10000 files and 10000 transactions), IoZone (with defaults and read and write benchmarks), Bonnie (with a 100MB file and read and write), Tar (on the Linux Kernel 2.6.23 source tree) and Ttcp (with 16MB send and receive). For comparison purposes we execute these applications on the native system and the system with *XTREC*. We ran each of these applications 3 times on each platform. Figure 4b shows the result of our application benchmarks.

From the application benchmark results, we observe that all I/O–bound applications run close to their native speed with *XTREC*. The only exception is Bonnie, which incurs a latency similar to that of compute–bound applications. We attribute this to the internal architecture of Bonnie, which issues reads and writes to the disk system one byte at a time using a loop structure. Thus, in our tests with a 100MB file, it results in a lot of branch instructions.

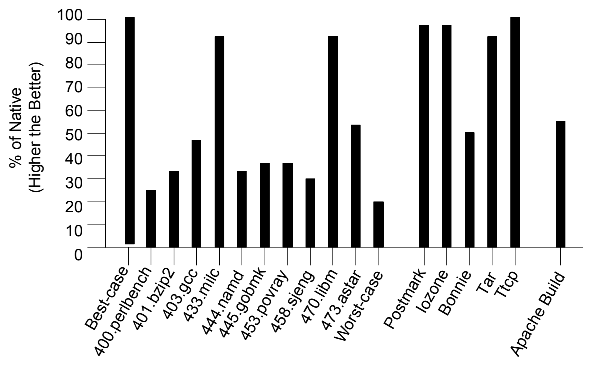
Compute–bound applications result in higher latency and their latency is proportional to the number of branch instructions. As examples, 433.milc and 470.libm almost run at native speed when compared to the other compute bound benchmarks. This is due to the fact that the number of dynamic branch instructions in those benchmarks is very minimal as compared to the others [27]. At the extreme ends of the spectrum are the best–case application (with no branches) and the worst–case application which just performs a tight loop generating the maximum number of BTMs. The former executes at native speed while the latter runs at 20% of native speed.

Figure 4c shows the log size generated by the compute and I/O bound applications. As seen from the figure, the log size of an application is proportional to the number of branch instructions. As examples, 400.perlbench and 458.sjeng which have the higher number branch instructions generate larger logs. Note that even though the worst–case application has the highest number of branch instructions, the log generated by it is very small. This is due to the fact that the Worst-case application in our case was a tight loop which resulted in the same BTM being emitted for each iteration of the loop which results in a very high compression ratio. Though the compression ratio is application specific, the average compres-

	BTM	Qualifier		#DB	NIC Buffer
		P*	No P*		Transmit (64MB)
Latency (in clock cycles)	2.72	1143	652	201	769,019,125

** P = Parsing OS Paging Structures*

(a)



(b)

Benchmark	Log Size (in GB)		Compression
	Normal	Compressed*	Ratio
Compute Bound Applications			
Best-case	~0	~0	-
400.perlbench	238.145	10.824	22.06
401.bzip2	220.534	8.482	26.05
403.gcc	131.812	5.492	24.01
433.milc	11.895	0.626	19.02
444.namd	103.062	4.122	25.08
445.gobmk	234.498	8.685	27.18
453.povray	73.492	3.449	21.30
458.sjeng	256.196	10.674	24.18
470.libm	13.654	0.620	21.40
473.astar	106.942	4.113	25.41
Worst-case	0.562	0.001	562
I/O Bound Applications			
Postmark	0.812	0.173	4.69
Iozone	1.062	0.041	25.90
Bonnie	2.687	0.034	79.02
Tar	0.625	0.078	8.01
Ttcp	0.007	0.0002	35
Mixed Applications			
Apachebuild	16.637	0.780	21.32
Webserver Access for 1 Day	1980.133	358.435	5.52
Idle System for 1 Day	15.436	1.406	10.97

** Compressed Log Size is using Gzip with compression level 5*

(c)

Figure 4: *XTREC* Benchmarks: (a) Micro–Benchmarks showing various ways in which *XTREC* adds overhead to system operation, (b) Application–Benchmarks showing that Compute Bound Applications with high branching logic result in higher overheads when compared to I/O Bound Applications which run almost at native speed, and (c) Log-Size Measurements showing that the log generated by *XTREC* is well within limits for current enterprise settings

sion ratio during our experiments, as seen from Figure 4c is around 20 with gzip level 5 compression. Note that the compression is done in real–time by the log store.

We also performed a couple of experiments to test the sustained log size of some applications with *XTREC*. For the first experiement, we ran the apache ab benchmark tool for a period of 24 hours with 128 simultaneous clients simulating a real–world webserver access. For the second experiment, we let the host idle for a period of 24 hours. As seen from the table, the log sizes for both experiments over a 24 hour period 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, we believe, given the size of our logs, that enterprises should be easily able to maintain logs for a period of 3–4 months.

Finally, we performed another experiment to isolate the component(s) of *XTREC* responsible for the latency. We modified the logger code to test for the DMA transfer to be successful or not before switching to the recording buffer on a BTM buffer overflow. We noted that the logger code never had to wait during all our application benchmarks which showed that the DMA was occuring in parallel to the recording and contributed nothing to the latency. Further, we also noted that Annotators contributed very minimally to the results. Hence we conclude that the processor BTM is the source of the majority of latency in our experiments.

5.4 Case Study of Haxdoor.KI

In this section we discuss the *Haxdoor.KI* malware [15] employing the *A-311 Death Backdoor* [32] and show how one can determine that it executed on a host using *XTREC*.

Haxdoor.KI is both a rootkit and a backdoor allowing a remote attacker to take complete control of the system and steal important information such as keystrokes, password and sensitive files. Upon infection, the rootkit component (named *xopptp.sys*) is registered with the OS registry and started. The registering process ensures that the malware is always loaded on the next OS bootup. Also, a backdoor DLL (named *xopptp.dll*) is injected into the *winlogon.exe* process and also registered with the winlogon notify DLLs (a set of DLLs that are automatically loaded by the winlogon process on startup). This allows the backdoor to be resident and active in system memory without any user intervention. The rootkit also hides critical files and processes belonging to the malware, from the user. The backdoor DLL uses the *A-311 Death Backdoor* library and listens on port *16661* for a remote attacker. The port itself is password protected and can only be activated by the *A-311 Death Remote Administration Toolkit (RAT)*. The backdoor allows a variety of operations to be performed in the system such as: file uploads and downloads, trojan updates, activating keyboard hooks, stealing clipboard data, and desktop screenshots. One of the most interesting feature of the backdoor is the ability to uninstall itself from the system leaving no trace that it was ever there.

We obtained a sample of *Haxdoor.KI* from Offensive Computing [11] and deployed it on our test host system by executing the malware executable. We then connected to the host on port *16661* from another system using the *A-311 RAT*. We then performed the following actions using the RAT: (a) connected, (b) uploaded and installed the keylogger on the host, (c) launched internet explorer on the host and entered a site that was password protected, (d) downloaded the captured key log file, and (e) uninstalled and removed. We also obtained functional and internal details of the Haxdoor.KI malware [15, 21] and then searched the *XTREC* log for characteristic information pertaining to the malware as well as our actions. Figure 5 shows the log we reconstructed for our actions (a), (b) and (e).

Lines 1–2 of Figure 5 correspond to the host OS Kernel initializing and creating the *winlogon.exe* process. Lines 3–8 correspond to the *winlogon.exe* process address space being initialized by the OS including mapping the malware DLL *xopptp.dll*. Lines 9–14, show the control flow trace of the connect request performed to port *16661*. As anticipated, the control flow consists of modules *ws2_32.dll* and *wsock32.dll* which form the BSD socket layer in Windows. Line 14 shows the control flow into the malware DLL *xopptp.dll* as a result of a successful *accept* function.

Lines 15–20 of Figure 5 correspond to the keylogger (*loggerB.exe* and *loggerB.dll*) being installed and initialized on the host. Lines 15–17 correspond to the keylogger process creation using the *ShellExecute* API while lines 18–20 correspond to the keylogger initialization by setting a system wide keyboard hook using the *SetWindowsHookEx* API.

Lines 20–27 of Figure 5 correspond to the uninstall and remove action of the malware. The unload operation of the malware consists of two steps. In the first step, a *DeviceIOControl* API call is issued to the rootkit (*xopptp.sys*) which causes its unload function to be executed resulting in its removal from memory and registry. In the second step, a *FreeLibrary* API call is issued which causes the backdoor DLL (*xopptp.dll*) to be unloaded from the *winlogon.exe* process context. After the above two steps are completed, the malware files *xopptp.dll* and *xopptp.sys* are deleted with other supporting files from the system thereby wiping its trace from the system.

Using details similar to that shown in the log of Figure 5 and knowing that the information logged by *XTREC* is accurate and uncompromised, we can determine that *Haxdoor.KI* (or an equivalent) was therefore executed in the host.

6 Limitations and Future work

6.1 Limitations

XTREC write–protects the OS Kernel code and critical CPU structures to ensure Annotator persistence (see Section 3.7.2). Thus, any code that modifies the OS Kernel code cannot be logged currently. However, we consider this as a positive side effect since, on contemporary OSes the Kernel code is never modified during runtime and hence such an activity can be considered malicious. We also realize that in certain situations, such as using *XTREC* to document a vulnerability which employs such mechanisms, this would be a limitation.

XTREC currently only logs control flow information and not data flow. While control flow information is sufficient to identify a large class of code, it becomes diffucult to identify certain malicious code employing mimicry attacks where the control flow closely mimics a benign application [34]. We are currently working on adding data logging support to address this limitation.

6.2 Future Work

Trusted BIOS/Pre-boot:*XTREC* relies on hardware–based attestation and assumes that the Loader is always executed by the BIOS/pre–boot stage. Further, it trusts the BIOS in its approach to ensure that the OS Kernel gets control directly from its boot code (for applicable OSes). *XTREC* could use remote verification instead of relying on a trusted BIOS and hardware–based attestation. When the Loader gets control it calls upon an external trusted entity (or the log store) to verify its integrity. Also, the log store, upon receiving the initial network negotiation could remotely verify the integrity of *XTREC*. The boot phase of the OS can be removed and replaced with code which does not employ the BIOS. As an example, the boot stage of Windows can be replaced by ReactOS WinLdr [28] which is capable of booting windows without accessing the BIOS.

SMM Mode:SMM code execute typically execute as a result of a System Management Interrupt (SMI) in an autonomous fashion in real mode. As AMD SVM provides intercepts to handle SMI, *XTREC* could support third party SMM by creating another HVM to run the SMI handler within it. Another option is to use an opensource BIOS such as CoreBoot [1] which does not use SMM.

7 Related Work

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

Software VM Based: In software VM based ap-

```
    ...
01. An: PROCESS CREATE -> Pid=0x8089AA80, Ppid=0x00000000, Range=0x80000000-
    0x80050000, Name=\%SYSTEMROOT%\SYSTEM32\NTOSKRNL.EXE
    ...
02. An: PROCESS CREATE -> Pid=0x89985140, Ppid=0x8997D140, Range=0x01000000-
    0x01082000, Name=\%SYSTEMROOT%\SYSTEM32\WINLOGON.EXE
    ...
03. An: MAP VIEW -> Context=0x89985140, Range=0x7C800000-0x7C8C0000,
    Name=\%SYSTEMROOT%\SYSTEM32\NTDLL.DLL
    ...
04. An: MAP VIEW -> Context=0x89985140, Range=0x77E40000-0x77F42000,
    Name=\%SYSTEMROOT%\SYSTEM32\KERNEL32.DLL
    ...
05. An: MAP VIEW -> Context=0x89985140, Range=0x71C00000-0x71C17000,
    Name=\%SYSTEMROOT%\SYSTEM32\WS2_32.DLL
    ...
06. An: MAP VIEW -> Context=0x89985140, Range=0x10000000-0x10052000,
    Name=\%SYSTEMROOT%\SYSTEM32\XOPPTP.DLL
    ...
07. An: MAP VIEW -> Context=0x89985140, Range=0x00DF0000-0x00E92000,
    Name=\%SYSTEMROOT%\SYSTEM32\WININET.DLL
    ...
08. An: MAP VIEW -> Context=0x89985140, Range=0x7C8D0000-0x7D0D3000,
    Name=\%SYSTEMROOT%\SYSTEM32\SHELL32.DLL
    ...
09. An: CONTEXT SWAP -> Pid=0x89985140 (*winlogon.exe*)
    ...
10. BTM: 0x001B:0x000000007C802150 (ntdll.dll)
    ...
11. BTM: 0x0008:0x000000008001345A (ntoskrnl.exe)
    ...
12. BTM: 0x001B:0x0000000071C03562 (ws2_32.dll)
    ...

13. BTM: 0x001B:0x0000000000E17860 (wininet.dll)
    ...
14. BTM: 0x001B:0x000000001000235B (xopptp.dll) (*connect request*)
    ...
15. BTM: 0x001B:0x000000007C8D4358 (shell32.dll) (*executing keylogger*)
    ...
16. An: PROCESS CREATE -> Pid=0x88321D88, Ppid=0x890A3730, Range=0x0040000
    -0x0040D000, Name=\%SYSTEMROOT%\SYSTEM32\LOGGERB.EXE
    ...
17. An: MAP VIEW -> Context=0x88321D88, Range=0x77E40000-0x77F42000,
    Name=\%SYSTEMROOT%\SYSTEM32\KERNEL32.DLL
    ...
18. An: CONTEXT SWAP -> Pid=0x88321D88 (*loggerb.exe*)
    ...
19. An: MAP VIEW -> Context=0x88321D88, Range=0x10000000-0x1000E000,
    Name=\%SYSTEMROOT%\SYSTEM32\LOGGERB.DLL
    ...
20. BTM: 0x001B:0x0000000077E48910 (kernel32.dll) (*install keylogging*)
    ...
21. An: CONTEXT SWAP -> Pid=0x89985140 (*winlogon.exe*)
    ...
22. BTM: 0x001B:0x00000000100083A4 (xopptp.dll)
    ...
23. BTM: 0x001B:0x0000000077EA8940 (kernel32.dll) (*device io control*)
    ...
24. BTM: 0x0008:0x00000000BAC1D670 (xopptp.sys) (*unload function*)
    ...
25. An: UNLOAD SYSIMAGE -> RANGE=0xBAC1B000-0xBAC20140 (*unload driver*)
    ...
26. BTM: 0x001B:0x0000000077EB1454 (kernel32.dll) (*unload library*)
    ...
27. An: UNMAP VIEW -> Range=0x10000000-0x10052000 (xopptp.dll)
    ...
```

Figure 5: XTRAC Log for *HaxDoor.KI/A-311 Death BackDoor*: Revealing Actions of the Attacker such as a Establishing a Connection to the BackDoor and the Complete Removal of the Malware from the Host

proaches, a technique called VM introspection is employed, whereby software VMs such as VMWare [33], Xen [5], UML Linux [12], etc. are extended to deliver events to the framework which then acts upon them. Aftersight [10] employs VMWare to decouple analysis from normal execution by logging nondeterministic VM inputs and replaying them on a separate analysis platform for intrusion detection and bug detection. ReVirt [13] employs UML Linux 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 [37] extends this scheme to multiprocessors. Other intrusion detection tools such as Backtracker [22] and [20] also work on the same principle and employ ReVirt as their base. VMIIDS [17] and Psyco-Virt [4] are some more examples of systems employing VM introspection for execution control flow logging. Time Traveling Virtual Machine (TTVM) [23] logs execution control flow and uses it to detect bugs in a OS Kernel.

Software VM based approaches rely on a host OS for storing the log. Thus, their TCB is an order of magnitude greater than ours (since a typical OS has lots of supporting drivers, services and dynamic components). An attack exploiting a vulnerability in any of the host OS components can result in the integrity of the log being compromised. While, the past log can be prevented from being tampered with [7], nothing prevents the attacker from inserting new entries and even deleting the log itself. Also, the logging and replay approach can result in inaccurate execution during replay due to nondeterminism [30]. As an example, ReVirt does not handle instructions that result in non–determinism such as *RDTSC* and *RDPMC*. Furthermore, the software VMs employed are either too lightwieght to host a commodity OS (e.g., ReVirt uses UMLLinux which cannot run Windows) or too big (e.g., Xen, Vmware) giving rise to vulnerabilities within the software VM itself [16].

OS Level Logging: In OS level logging, the host OS is modified to trigger messages during the operation of the OS Kernel. Syslogd [31] and Windows Management Instrumentation [25] provide support for system logging on Unix–based systems and Windows. Filemon [29] and Regmon [29] are system utilities which log real-time file and registry activities. Tools such as Valgrind [26], and DynamoRIO [9] can run a specified program within an OS using dynamic binary translation and can be used for instruction–level execution control flow logging. Tools such as Strace [35] on the other hand dynamically rewrite portions of the running binary in order to log the sequence of systems calls made by a process.

Approaches based on OS level logging, store the log in the host OS which cannot guarantee the intergrity of the log as mentioned previously. Further, the framework itself resides within the host OS which makes it easily susceptible to attacks. As an example, the hooks deployed by tools such as Filemon can be removed by restoring the OS system service table to default values.

Debuggers and Specialized Hardware: Software debuggers such as WinDbg [24] allow recording of execution control flow by single–stepping instructions. A similar capability is provided by hardware debuggers such as ones manufactured for AMD and Intel processors by American Arium [3]. 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. [8]. However, these approaches are designed for manual debugging and are not suited for real–time logging and online deployment. CADRE [30] is a cycle accurate deterministic replay system that can recreate the execution control–flow of a system accurately. However, it uses a platform that is ad-hoc and very different from commodity systems. Software cycle accurate deterministic simulators such as PTLSim [38] and AMD SimNow [6] are very slow (typically 50 to 1000x slowdown) and do not contain adequate support for commodity hardware making them unsuitable for online deployments. Further, they suffer from the same drawbacks as software VM based approaches as the simulation is done on a host OS.

8 Conclusion

With the rapid creation of new malware, XTREC offers the useful property to perform forensic analysis *a posteriori*. Based on our experimental results we find that *XTREC* is viable on current enterprise systems, and postulate that minor hardware changes could considerably improve performance. *XTREC* 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.

References

[1] Agnew, A., Sulmicki, A., Minnich, R., and Arbaugh, W. Flexibility in rom: A stackable open source bios. In *USENIX 2003 Annual Technical Conference, FREENIX Track* (2003), pp. 115–124.

[2] AMD. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, 3.12 ed., September 2006.

[3] American Arium Inc. *Hardware and Software Debugging Tools for Intel and AMD Processors, http://www.arium.com/.*

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

[5] 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 *SOSP '03*, pp. 164–177.

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

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

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

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

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

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

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

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

[14] Espiner, T. Swedish bank hit by biggest ever online heist. *ZDNet News Archives http://news.zdnet.co.uk/security/* (December 2006).

[15] F-Secure. Haxdoor.ki. *F-Secure Trojan Information*.

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

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

[18] GNU Software. *GNU Grub Manual*, 0.97 ed., May 2005.

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

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

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

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

[23] King, S. T., Dunlap, G. W., and Chen, P. M. Debugging operating systems with time-traveling virtual machines. In *ATEC '05*, pp. 1–1.

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

[25] Microsoft Corp. *Windows Management Instrumentation: Background and Overview, MSDN Library*, November 2000.

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

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

[28] ReactOS Foundation. *The ReactOS Project, http://www.reactos.org*.

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

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

[31] Schwarz, M. The system logging daemons, syslogd and klogd. *Linux Journal, http://www.linuxjournal.com* (July 2000).

[32] SpywareGuide. A311 death.

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

[34] Wagner, D., and Soto, P. Mimicry attacks on host–based intrusion detection systems. In *9th ACM Conference on Computer and Communications Security* (2002), pp. 255–264.

[35] Weimer, H. Dissecting programs. *OS Reviews, http://www.osreviews.net* (September 2006).

[36] Wheeler, D. Counting source lines of code SLOC. `http://www.dwheeler.com/sloc/`.

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

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