

USING HARDWARE ISOLATED EXECUTION ENVIRONMENTS
FOR SECURING SYSTEMS

by

Fengwei Zhang
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial Fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Computer Science

Committee:



Dr. Angelos Stavrou, Dissertation Director



Dr. Fei Li, Committee Member



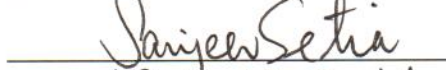
Dr. Duminda Wijesekera, Committee Member



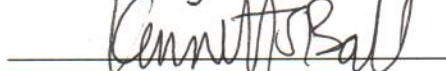
Dr. Houman Homayoun, Committee Member



Dr. Haining Wang, Committee Member



Dr. Sanjeev Setia, Department Chair



Dr. Kenneth S. Ball, Dean, The Volgenau School
of Engineering

Date:

04/23/2015

Spring Semester 2015
George Mason University
Fairfax, VA

Using Hardware Isolated Execution Environments for Securing Systems

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Fengwei Zhang

Master of Science

Columbia University, 2010

Bachelor of Science

North China University of Technology and Southern Polytechnic State University, 2008

Director: Dr. Angelos Stavrou
Department of Computer Science

Spring Semester 2015
George Mason University
Fairfax, VA

Copyright © 2015 by Fengwei Zhang
All Rights Reserved

Table of Contents

	Page
List of Tables	v
List of Figures	vi
Abstract	vii
1 Introduction	1
1.1 Problem Statement	3
1.2 Thesis Statement	3
2 Background	4
2.1 Computer Hardware Architecture	4
2.2 BIOS and Coreboot	4
3 Related Work	6
3.1 Memory-based Attacks and Detection	6
3.2 Bridging the Semantic Gap in VMI Systems	7
3.3 Malware Debugging and Analysis	7
3.4 Trusted Execution Environments	9
3.5 SMM-based Systems	10
4 Hardware Isolated Execution Environments	12
4.1 System Management Mode	12
4.2 SecureSwitch: BIOS-Assisted Isolation Environment	13
5 Using Hardware Isolated Execution Environments for Malware Detection	16
5.1 Spectre: Application- and OS-level Malware Detector	16
5.1.1 Introduction	16
5.1.2 Threat Model and Assumptions	17
5.1.3 System Architecture	17
5.1.4 Implementation	22
5.1.5 Evaluation	30
5.2 HyperCheck: Hypervisor-level Malware Detector	37
5.2.1 Introduction	37
5.2.2 Threat Model	38
5.2.3 System Architecture	39

5.2.4	Implementation	43
5.2.5	Evaluation	49
5.3	IOCheck: Firmware-level Malware Detector	56
5.3.1	Introduction	56
5.3.2	Threat Model and Assumptions	59
5.3.3	System Architecture	59
5.3.4	Implementation	65
5.3.5	Evaluation	70
6	Using Hardware Isolated Execution Environments for Malware Debugging	75
6.1	MalT: Towards Transparent Debugging	75
6.1.1	Introduction	75
6.1.2	Threat Model and Assumptions	77
6.1.3	System Architecture	79
6.1.4	Implementation	81
6.1.5	Transparency Analysis	89
6.1.6	Evaluation	96
7	Using Hardware Isolated Execution Environments for Executing Sensitive Workloads	100
7.1	TrustLogin: Securing Password-Login	100
7.1.1	Introduction	100
7.1.2	Threat Model and Assumptions	102
7.1.3	System Architecture	103
7.1.4	Case Study	114
8	Conclusions and Future Work	123
8.1	Conclusions	123
8.1.1	System Introspection for Malware Detection	123
8.1.2	Transparent Malware Debugging	124
8.1.3	Executing Sensitive Workloads	125
8.2	Future Work	125
	Bibliography	128

List of Tables

Table	Page
3.1 Summary of SMM Attacks and Solutions	10
5.1 Heap Spray Attack Detection Time (n=25)	33
5.2 Runtime Comparison of Introspection Programs Between Spectre and Virtuosio	37
5.3 Symbols for Xen hypervisor, Domain 0, Linux and Windows	50
5.4 Time Measurements for Variable Packet Sizes in HyperCheck-II	54
5.5 Comparison on Time Overhead	55
5.6 Comparison on Capability and Overhead	56
5.7 IOMMU Configurations	63
5.8 PCI Expansion ROM Header Format for x86	68
5.9 PCI Data Structure Format	69
5.10 Breakdown of SMI Handler Runtime (Time: μs)	71
5.11 Random Polling Overhead Introduced on Microsoft Windows and Linux . .	72
5.12 Comparison between SMM-based and DRTM-based Approaches	74
6.1 Communication Protocol Commands	83
6.2 Stepping Methods in MalT	88
6.3 Summary of Anti-debugging, Anti-VM, and Anti-emulation Techniques . .	94
6.4 Running Packed <code>Notepad.exe</code> under Different Environments	95
6.5 Breakdown of SMI Handler (Time: μs)	98
6.6 Stepping Overhead on Windows and Linux (Unit: Times of Slowdown) . . .	98
7.1 Musical Notes of an Octave	113
7.2 Breakdown of the SMI Handler Runtime (Time: μs)	120
7.3 Comparison between Linear Searching and Semantic Searching	121

List of Figures

Figure	Page
2.1 Typical Hardware Layout of a Computer	5
4.1 Architecture of SecureSwitch System	13
5.1 Operation of Spectre	18
5.2 Translating Addresses from Virtual to Physical	20
5.3 Finding the List of Processes in Windows	23
5.4 Finding Heap Data in Windows	25
5.5 Finding the List of Tasks in Linux	27
5.6 Breakdown of SMI Handler Runtime	30
5.7 Overhead Introduced in Microsoft Windows	35
5.8 Overhead Introduced in Linux	35
5.9 Architecture of HyperCheck	40
5.10 Breakdown of HyperCheck Runtime	51
5.11 Network Time Delay for Variable Data Size in HyperCheck	52
5.12 Network Time Delay for Variable Packet Size in HyperCheck	53
5.13 Overhead Introduced in HyperCheck-II with Different Sampling Intervals .	55
5.14 Architecture of IOCheck	60
5.15 Architecture Block Diagram of Intel 82574L	67
6.1 Architecture of MalT	80
6.2 Finding a Target Application in Windows	84
7.1 Architecture of TrustLogin	103
7.2 Keystroke Handling in TrustLogin	105
7.3 Transmit Descriptor Format	110
7.4 FTP Login With and Without TrustLogin on Windows and Linux	116
7.5 Filling the Semantic Gap by Using Kernel Data Structures in Linux	117

Abstract

USING HARDWARE ISOLATED EXECUTION ENVIRONMENTS FOR SECURING SYSTEMS

Fengwei Zhang, PhD

George Mason University, 2015

Dissertation Director: Dr. Angelos Stavrou

With the rapid proliferation of malware attacks on the Internet, malware detection and analysis play a critical role in crafting effective defenses. Advanced malware detection and analysis rely on virtualization and emulation technologies to introspect the malware in an isolated environment and analyze malicious activities by instrumenting code execution. Virtual Machine Introspection (VMI) systems have been widely adopted for malware detection and analysis. VMI systems use hypervisor technology to create an isolated execution environment for system introspection and to expose malicious activity. However, recent malware can detect the presence of virtualization or corrupt the hypervisor state and thus avoid detection and debugging.

In this thesis, I developed several systems using hardware isolated execution environments for attack detection, malware debugging, and sensitive operations. My research approach combines 1) the isolated execution concept with 2) hardware-assisted technologies. It leverages System Management Mode (SMM), a CPU mode in the x86 architecture, to transparently detect and debug armored malware and perform sensitive workloads. This research uses SMM to secure systems with a minimal Trust Computing Base (TCB) and low performance overhead. In addition, I develop a BIOS-assisted isolation environment

that is capable of running a secure commodity OS.

To demonstrate the effectiveness of my research, several prototypes of using SMM as the isolated execution environment are implemented. First, I use SMM to introspect all layers of system software, including applications, OSes, hypervisors, and firmware. Secondly, my research leverages SMM to transparently debug armored malware and achieve a higher level of transparency than state-of-the-art systems. Lastly, this thesis uses SMM to securely perform password-logins without trusting the operating system and prevents ring 0 keyloggers.

Chapter 1: Introduction

The proliferation of malware has increased dramatically and caused serious damage for Internet users in the past few years. McAfee reported that the presence of malware has been greatly increasing during the first quarter in 2014, seeing more than 30 million new malware samples [1]. In the last year alone, Kaspersky Lab products detected almost 3 billion malware attacks on user computers, and 1.8 million malicious programs were found in these attacks [2]. Symantec blocked an average of 568,000 web attacks per day in 2013, a 23% increase over the previous year [3]. Nowadays, computer systems rely on a large amount of applications and software to operate, and these software inevitably create many vulnerabilities that can be easily exploited by attackers. As such, malware detection and analysis are critical to understanding new infection techniques and maintaining a strong defense.

Traditionally, malware detection is provided by installing anti-malware tools (e.g., anti-virus) within the operating system. However, all defensive techniques that run as processes in the operating system are inherently vulnerable to malicious code executing at the same level. Therefore, when a rootkit compromises the OS, most if not all of the common protection suites become ineffective, misleading the user to believe that the system is protected while the malware operates freely in the background.

To address this problem, security researchers use virtualization technology to analyze system states for malware detection and debugging. It creates an isolated execution environment that cannot be affected by the advanced attacks (e.g., rootkits), and the malware would have no place to hide. For instance, Virtual Machine Introspection (VMI) [4] executes all programs inside a guest Virtual Machine (VM), translating their semantic state information to malware detection tools that run outside the VM (e.g., host machine). This

approach isolates and protects the malware detection software from the potentially vulnerable guest so that stealthy malware cannot interfere or corrupt the protection mechanisms. However, virtualization-based systems have some practical limitations.

First, virtualization-based systems depend on the integrity of the hypervisor, which has a sizable Trusted Computing Base (TCB). For instance, the latest Xen 4.2 contains approximately 208,000 lines of code. Although this size is dwarfed by the code size of a typical operating system, the attack surface posed by the hypervisor remains significant. The National Vulnerability Database [5] shows that there are 100 vulnerabilities in Xen and 90 vulnerabilities in VMWare ESX.

Secondly, virtualization-based systems are not able to detect and analyze rootkits with the same or higher privilege level. Hypervisor is referred to as ring -1, and it is capable of detecting rootkits with lower privileges (e.g., OS rootkits with ring 0 privilege). However, VM escape attacks [6, 7], hypervisor rootkits [8], and firmware rootkits [9, 10] are widely deployed.

Thirdly, malware writers can easily escape this detection mechanism by using a variety of anti-debugging, anti-virtualization, and anti-emulation techniques [11–17]. Malware can easily detect the presence of a VM or emulator and alter its behaviors to hide itself. Indeed, malware running inside of the VM can read a virtual device name or the IDT or LDT registers to detect the presence of a VM [18]. In such cases, attaining behavioral transparency from the perspective of the malware is urgent yet difficult to achieve.

Lastly, and most importantly, traditional VMI techniques incur a high overhead on system performance, making them unpopular among end-users. Some of the more theoretical solutions can incur such a high latency that they are deemed unfit for use in current computing systems. For instance, existing VMI methods may take on the order of seconds to pause a VM guest to scan its memory. I could potentially improve the performance through the use of heuristics or other approximations, but this leads to false positives and false negatives, adversely affecting the end solutions.

1.1 Problem Statement

Traditionally, malware detection is provided by installing anti-virus tools, and malware debugging is performed by user-level applications such as IDAPro. These defensive mechanisms that run as processes within the OS are vulnerable to malware executing at the same privilege-level. Furthermore, a rootkit can compromise the OS and avoid being detected. Recently, security researchers have used virtualization technology to create an isolated execution environment for securing systems. For instance, Virtual Machine Introspection (VMI) has been widely adopted for malware detection and analysis. However, existing virtualization-based approaches have four major limitations for malware detection and debugging, which are summarized as follows: 1) dependence on hypervisors with a large amount of TCB; 2) incapability of detecting hypervisor and firmware rootkits with the same or higher privilege-level; 3) visibility to malware with anti-debugging, anti-virtualization, and anti-emulation techniques; and 4) suffering a high overhead on system performance. Moreover, existing trusted execution environments (e.g., Flicker [19]) depend on Dynamic Root of Trust for Measurement (DRTM), which introduces a large overhead in Trusted Platform Module (TPM) operations.

1.2 Thesis Statement

This thesis addresses the limitations of existing isolated execution environments for system security. My research uses System Management Mode (SMM), a special CPU mode in the x86 architecture, to perform security operations. The proposed execution environment can introspect all layers of system software, transparently debug malware, and securely execute sensitive workloads with a minimal TCB and low performance overhead. This thesis also develop a BIOS-assisted isolation environment that is capable of running a secure commodity OS.

Chapter 2: Background

In this chapter, I introduce basic concepts and pertinent components. I first explain a typical computer architecture. Then, I describe the basic input-output system (BIOS), which serves as my trust base. I also introduce Coreboot, which is a particular open-source BIOS that facilitated my implementation.

2.1 Computer Hardware Architecture

Figure 2.1 shows the hardware architecture of a typical computer. The Central Processing Unit (CPU) connects to the Northbridge via the Front-Side Bus. The Northbridge has Memory Management Unit (MMU) and IOMMU, collectively called the memory controller hub. The Northbridge also connects to the memory, graphics card, and Southbridge. The Southbridge, also called the I/O controller hub, connects a variety of I/O devices including USB, SATA, and Super I/O, among others. Note that the BIOS is also connected to the Southbridge.

2.2 BIOS and Coreboot

BIOS is an indispensable component for all computers. The main function of the BIOS is to initialize the hardware devices, including the processor, main memory, chipsets, hard disk, and other necessary IO devices. BIOS code is normally stored on a non-volatile ROM chip on the motherboard. In recent years, a new generation of BIOS, referred to as unified extensible firmware interface (UEFI), has become increasingly popular in the market. UEFI is a specification that defines the new software interface between OS and firmware. One goal with UEFI is to ease the development by switching to the protected mode in

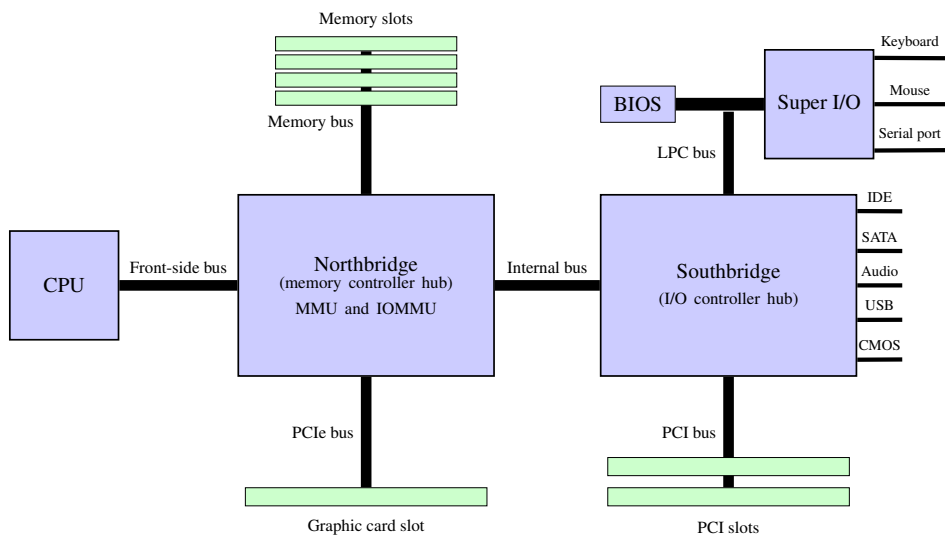


Figure 2.1: Typical Hardware Layout of a Computer

an early stage and writing most of the code in C language. A portion of the Intel UEFI frame (named Tiano Core) is open source; however, the main function of the UEFI (to initialize the hardware devices) is still closed source. Coreboot [20] (formerly known as LinuxBIOS) is an open-source project aimed at replacing the proprietary BIOS (firmware) in most modern computers. It performs a small amount of hardware initialization and then executes a so-called payload. Similar to the UEFI-based BIOS, Coreboot also switches to protected mode in a very early stage and is written mostly in the C language. We choose to use Coreboot rather than UEFI because Coreboot does all of the hardware initializations, whereas we would need to implement UEFI firmware from scratch, including obtaining all of the data sheets for our motherboard and other devices

Chapter 3: Related Work

In this chapter, I first discuss memory-based attacks and existing detection mechanisms. I also present related works on bridging the semantic gap in virtual machine introspection (VMI) systems. Next, I survey related works on malware debugging and analysis. Then, I explain related works on trusted execution environments. Lastly, I discuss related works that use SMM to build new attacks and new protection mechanisms.

3.1 Memory-based Attacks and Detection

Memory-based attack detection is an active research area. Due to the extensive usage of web browsers on various web applications in recent years, more and more heap-based memory corruption attacks have surfaced [21–23], as attackers can easily allocate malicious objects using scripting languages embedded in a web page. In 2009, heap spraying exploits were identified in Adobe Reader using JavaScript embedded in malicious PDF documents [24].

Researchers have proposed a number of effective defensive mechanisms [25, 26] against heap-based memory attacks. DieHarder [27] analyzed several memory allocators and showed they were vulnerable to attacks but also presented a new memory allocator against heap-based attacks. Nozzle [28], a runtime heap spray detector, examined individual objects in the heap, interpreted them as code, and performed static code analysis to detect malicious intent. Instead of detecting the attacks at the operating system level, [29] can detect drive-by-download attacks by emulation to automatically identify malicious JavaScript code.

A number of mechanisms and systems have been built to enforce kernel integrity and detect potential rootkits. SecVisor [30] is a tiny hypervisor that leverages new hardware extensions to enforce lifetime kernel integrity. However, the deployment of SecVisor requires modification of the kernel. Instead, I do not need to change any code in the operating

system, although it does require changing the BIOS. Flicker [19] and TrustVisor [31] employ Dynamic Root of Trust for Measurement (DRTM) to provide a trust environment for running security code. One particular usage is to run a rootkit detector for OS integrity checking. I can achieve a similar goal by using only SMM.

3.2 Bridging the Semantic Gap in VMI Systems

The semantic gap problem has fueled a large amount of research [32–34]. Recently, virtualization has been employed in malware detection and debugging. Security researchers have embraced virtual machine monitors (VMMs) as a new mechanism to guarantee deep isolation of untrusted system software components. “Out-of-the-box” defense mechanisms can resist tampering at the cost of a native, semantic view of the host that is enjoyed by the “in-the-box” approach. I must solve the same semantic gap problem since it has no context information when the system enters SMM.

VMWatcher [32] is a stealthy malware detection system that uses semantic view reconstruction. Essentially, it pauses a VM guest to scan the memory of that guest and then reconstructs semantic information of the data structures. Both Virtuoso [33] and VMST [34] are techniques that can automatically bridge the semantic gap in VM guests. Note that malware with ring 0 privilege can manipulate the kernel data structures to confuse the reconstruction process, and current semantic gap solutions suffer from this limitation [35]. As with VMI systems, SMM-based systems do not consider the attacks that mutate kernel structures.

3.3 Malware Debugging and Analysis

VAMPiRE [36] is a software breakpoint framework that runs within the operating system. Since it has the same privilege level as the operating system kernel, it can only debug Ring3 malware. Rootkits can gain kernel privileges to circumvent VAMPiRE.

Ether [37] is a malware analysis framework based on hardware virtualization extensions

(e.g., Intel VT). It runs outside of the guest operating systems by relying on underlying hardware features. BitBlaze [38] and Anubis [39] are QEMU-based malware analysis systems. They focus on understanding malware behavior instead of achieving better transparency. V2E [40] is a new malware analysis platform that combines both hardware virtualization and software emulation. Spider [41] uses Extended Page Table to implement invisible breakpoints and hardware virtualization to hide its side effects. Compared to my system, Ether, BitBlaze, Anubis, V2E, and Spider all rely on easily detected emulation or virtualization technology [11, 15, 16, 18].

BareBox [42] is a malware analysis framework based on a bare-metal machine without any virtualization or emulation technologies. However, it only targets the analysis of user-mode malware. Willems et al. [43] used branch tracing to record all of the branches taken by a program execution. As pointed out in the paper, the data obtainable by branch tracing is rather coarse, and this approach still suffers from a CPU register attack against branch tracing settings. BareCloud [44] is a recent armored malware detection system; it executes malware on a bare-metal system and compares disk- and network-activities of the malware with other emulation and virtualization-based analysis systems for evasive malware detection.

Virt-ICE [45] is a remote debugging framework similar to this work. It leverages virtualization technology to debug malware in a VM and communicates with a debugging client over a TCP connection. As it debugs the system outside of the VM, it is capable of analyzing rootkits and other ring 0 malware with debugging transparency. However, since it uses a VM, malware may refuse to unpack itself in the VM.

There is a vast array of popular debugging tools. For instance, IDA Pro [46] and OllyDbg [47] are commonly used debuggers running within the operating system that focus on Ring3 malware. DynamoRIO [48] is a process virtualization system implemented using software code cache techniques. It executes on top of the OS and allows users to build customized dynamic instrumentation tools. WinDbg [49] uses a remote machine to connect to the target machine using serial or network communications. However, these options

require special booting configuration or software running within the operating system, which is easily detected by malware.

3.4 Trusted Execution Environments

Flicker [19] and TrustVisor [31] employ a hardware support called Dynamic Root of Trust for Measurement (DRTM) with a small trusted computing base to create a secure environment. Flicker creates an on-demand secure environment using DRTM, while TrustVisor uses DRTM to securely initialize a light-weight hypervisor that uses hardware virtualization (VT-x/SVM) to protect the applications running in the secure environments. The two systems use the TPM to provide remote attestations and to securely store data when they are not executing. Compared to Flicker and TrustVisor, my thesis does not require DRTM and introduces a lower performance overhead.

Lockdown [50] is a system that uses a hardware switch and LEDs to provide a trusted path to a small hypervisor, which ensures virtual resource isolation between two OSes. Lockdown relies on the light-weight hypervisor to ensure that trusted applications can communicate only with trusted sites and thus can prevent malicious sites from corrupting the applications. To switch, it uses an ACPI-based mechanism (S4) to hibernate one OS and then wake up another one. Unfortunately, it requires more than 30 seconds to switch because hibernating requires writing the whole main memory content to the hard disk and reading it back later on.

Bumpy [51] is a Flicker-based system for securing sensitive network input. It handles inputs in a special code module that is executed in an isolated environment using the Flicker. Bumpy can protect a user's sensitive web inputs even with a compromised OS or web browser. Cloud Terminal [52] is a micro-hypervisor and provides secure access to sensitive applications from an untrusted OS. It moves most application logic to a remote server called the Cloud Rendering Engine and only runs a light-weight Secure Thin Terminal on the end host, so end-users can securely execute sensitive applications. It also uses the Flicker to setup the micro-hypervisor. Borders and Prakash proposed a Trusted Input

Table 3.1: Summary of SMM Attacks and Solutions

SMM Attacks	Solutions
Unlocked SMRAM [54–56]	Set D.LCK bit
SMRAM reclaiming [57]	Lock remapping and TOLUD registers
Cache poisoning [58, 59]	SMRR
Graphics aperture [60]	Lock TOLUD
TSEG location [60]	Lock TSEG base
Call/fetch outside of SMRAM [60, 61]	No call/fetch outside of SMRAM

Proxy (TIP) [53] to secure network inputs. The TIP runs as a module in a separate VM that proxies network connections of the primary VM. It uses the placeholder approach to substitute the actual sensitive data. TIP does not require modification of the web browser and server, and they are transparent to users. As stated in the limitation section of this paper, TIP relies on a virtual machine monitor. It also introduces a large trusted code base and significant slowdown for I/O intensive applications.

3.5 SMM-based Systems

System Management Mode (SMM) has been used as a basic building block for several defensive mechanisms. HyperGuard [57] suggests using SMM to monitor hypervisor integrity by taking snapshots of a VM guest and checking it in SMM. HyperSentry [62] used an out-of-band channel, specifically the Intelligent Platform Management Interface, to trigger SMM in checking the integrity of base code operating on critical data.

Several attacks based on SMM have been proposed, too. In 2004, Loic Dulfot [54] developed the first SMM-based attack to bypass protection mechanisms in OpenBSD. Before 2006, computers did not lock their SMRAM in the BIOS [55], and researchers used this flaw to implement SMM-based rootkits [55, 56]. Modern computers lock the SMRAM in the BIOS so that SMRAM is inaccessible from any other CPU modes after booting. Wojtczuk and Rutkowska demonstrated bypassing the SMRAM lock through memory reclaiming [57] or cache poisoning [58]. The memory reclaiming attack can be addressed by locking the remapping registers and Top of Low Usable DRAM (TOLUD) register. The cache poisoning

attack forced the CPU to execute instructions from the cache instead of SMRAM by manipulating the Memory Type Range Register (MTRR). Duflot also independently discovered this architectural vulnerability [59], but it has been fixed by Intel adding SMRR [63]. Furthermore, Duflot et al. [60] listed some design issues of SMM, but those issues can be fixed by correct configurations in BIOS and careful implementation of the SMI handler. Table 3.1 shows a summary of attacks against SMM and their corresponding solutions. Wojtczuk and Kallenberg [64] recently presented an SMM attack by manipulating UEFI boot script that allows attackers to bypass SMM lock and modify the SMI handler with ring 0 privilege. The UEFI boot script is a data structure interpreted by UEFI firmware during S3 resume. When the boot script executes, system registers like BIOS_NTL (SPI flash write protection) or TSEG (SMM protection from DMA) are not set so that attackers can force an S3 sleep to take control of SMM. Fortunately, as stated in the paper [64], the BIOS update around the end of 2014 fixed this vulnerability. In this thesis, I assume that SMM can be trusted.

Chapter 4: Hardware Isolated Execution Environments

4.1 System Management Mode

System Management Mode (SMM) is a mode of execution similar to Real and Protected modes available on x86 architectures. It provides a transparent mechanism for implementing platform-specific system control functions such as power management. It is implemented by the Basic Input/Output System (BIOS).

SMM is triggered by asserting the System Management Interrupt (SMI) pin on the CPU. This can be done in a variety of ways, which include writing to a hardware port or generating Message Signaled Interrupts with a PCI device. At the boundary of the next instruction, the CPU saves its state to a special region of memory called System Management RAM (SMRAM); it then atomically executes the SMI handler that is also stored in SMRAM. SMRAM cannot be addressed by the other modes of execution; the requests for addresses in SMRAM are instead forwarded to video memory by default. This caveat therefore allows SMRAM to be used as secure storage. The SMI handler is loaded into SMRAM by the BIOS at boot time, but once loaded, it cannot be changed outside of SMM. The SMI handler has unrestricted access to the physical address space and can run any instructions requiring any privilege level¹. The `RSM` instruction makes the CPU exit from SMM and resume execution in the previous mode.

In terms of transparency, the Intel manual specifies the following mechanisms to make SMM transparent to the application programs and operating systems [63]: 1) the only way to enter SMM is by means of an SMI; 2) the processor executes SMM code in a separate address space (SMRAM) that can be made inaccessible from the other operating modes; 3) upon entering SMM, the processor saves the context of the interrupted program or task; 4)

¹For this reason, SMM is often referred to as *ring -2*.

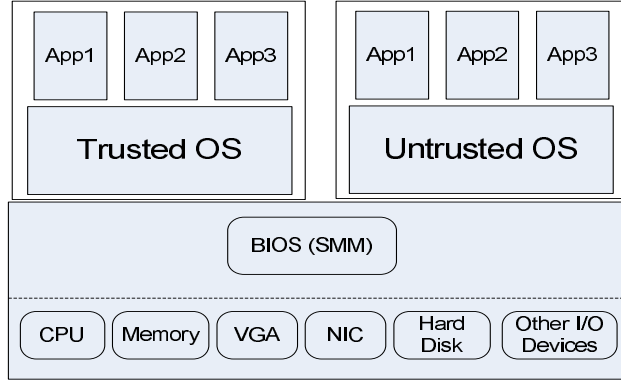


Figure 4.1: Architecture of SecureSwitch System

all interrupts normally handled by the operating system are disabled upon entry into SMM; and 5) the `RSM` instruction can be executed only in SMM.

My research will use SMM as an isolated execution environment to implement security operations including malware detection, transparent malware debugging, and online login.

4.2 SecureSwitch: BIOS-Assisted Isolation Environment

I co-developed a novel BIOS-assisted mechanism for secure instantiation and management of trusted execution environments [65]. A key design characteristic of this system is usability, the ability to quickly and securely switch between operating environments without requiring any specialized hardware or code modifications. The overall architecture of the SecureSwitch system is depicted in Figure 4.1, in which two OSes are loaded into the RAM at the same time. Commercial OSes that support ACPI S3 can be installed and executed without any changes. Instead of relying on a hypervisor, we modify the BIOS to control the loading, switching, and isolation between the two OSes.

The Secure Switching operation consists of two stages: *OS loading stage* and *OS switching stage*. In the OS loading stage, the BIOS loads two OSes into separated physical memory space. The trusted OS should be loaded first, followed by the untrusted OS. In the OS switching stage, the system can suspend one OS and then wake up another. In

particular, it must guarantee a trusted path against the spoofing trusted OS attack when the system switches from the untrusted OS to the trusted OS.

The system must guarantee a thorough isolation between the two OSes. Usually one OS is not aware of the other co-existing OS in the memory. Even if the untrusted OS has been compromised and can detect the coexisting trusted OS on the same computer, it still cannot access any data or execute any code on the trusted OS.

The system must guarantee a strong isolation between the two OSes to protect the confidentiality and integrity of the information on the trusted OS. According to the von Neumann architecture, we must enforce the resource isolation on major computer components, including CPU, memory, and I/O devices.

CPU Isolation: When one OS is running directly on a physical machine, it has full control of the CPU. Therefore, the CPU contexts of the trusted OS should be completely isolated from that of the untrusted OS. In particular, no data information should be left in CPU caches or registers after one OS has been switched out.

CPU isolation can be enforced in three steps: saving the current CPU context, clearing the CPU context, and loading the new CPU context. For example, when one OS is switching off, the cache is flushed back to the main memory. When one OS is switching in, the cache is empty. The content of CPU registers should also be saved separately for each OS and isolated from the other OS.

Memory Isolation: It is critical to completely separate the RAM between the two OSes so that the untrusted OS cannot access the memory allocated to the trusted OS. A hypervisor can control and restrict the RAM access requests from the OSes. Without a hypervisor, this system includes a hardware solution to achieve memory isolation. The BIOS allocates non-overlapping physical memory spaces for two OSes and enforces constrained memory access for each OS with a specific hardware configuration (e.g., DQS and DIMM Mask) that can only be set by the BIOS. The OS cannot change the hardware settings to enable access to the other OS's physical memory.

I/O Device Isolation: Typical I/O devices include hard disk, keyboard, mouse, network card (NIC), graphics card (VGA), etc. The running OS has full control of these I/O devices. For devices with their own *volatile memory* (e.g., NIC, VGA), we must guarantee that the untrusted OS cannot obtain any remaining information within the volatile memory (e.g., pixel data in the VGA buffer) after the trusted OS has been suspended. When a stateful trusted OS is switched out, the device buffer should be saved in the RAM or hard disk and then flushed. When a stateless trusted OS is switched out, the device buffer is simply flushed.

For I/O devices with *non-volatile memory* (e.g., USB, hard disk), the system must guarantee that the untrusted OS cannot obtain any sensitive data from the I/O devices used by the trusted OS. One possible solution is to encrypt/decrypt the hard disk when the trusted OS is suspended/awoken. However, this method will increase the OS switching time due to costly encryption/decryption operations. Another solution is to use two hard disks for two OSes separately and use BIOS (SMM) to ensure the isolation. When targeting browser-based applications that do not need to keep a local state, it is secure to save the temporary sensitive data in a RAM disk, which can maintain its content during OS sleep but gets cleaned when the system reboots.

Chapter 5: Using Hardware Isolated Execution Environments for Malware Detection

My research uses hardware isolated execution environments for malware detection. This thesis leverage SMM to build several prototypes for these purposes, and they are able to intropsect all layers of the system including application, OS, hypervisor, and firmware levels. Next, I explain these prototypes in detail.

5.1 Spectre: Application- and OS-level Malware Detector

5.1.1 Introduction

Virtual Machine Introspection (VMI) systems have been widely adopted for malware detection and analysis. VMI systems use hypervisor technology for system introspection and to expose malicious activity. However, recent malware can detect the presence of virtualization or corrupt the hypervisor state thus avoiding detection. I developed Spectre [66], a hardware-assisted dependability framework that leverages System Management Mode (SMM) to inspect the state of a system. Contrary to VMI, the trusted code base is limited to BIOS and the SMM implementations. Spectre is capable of transparently and quickly examining all layers of running system code including a hypervisor, the OS, and user-level applications. We demonstrate several use cases of Spectre including heap spray, heap overflow, and rootkit detection using real-world attacks on Windows and Linux platforms. In our experiments, full inspection with Spectre is 100 times faster than similar VMI systems because there is no performance overhead due to virtualization. Next, I explain the threat model, architecture, implementation, and evaluation of Spectre.

5.1.2 Threat Model and Assumptions

Most malware will alter memory at some point, causing the system to enter a state not intended in the original design. We call them *memory-based* attacks. For example, a typical heap spray attack will place a large number of NOP instructions in dynamic memory, a heap overflow attack will change heap application data or metadata in memory, and rootkits will alter kernel code or data. Since Spectre is capable of examining all layers of running system code and data (e.g., OS, user-level applications) in the memory, it can successfully detect those memory-based attacks including heap spray, heap overflow, and rootkits when accommodating corresponding memory checking modules.

Spectre uses SMM to detect malware in the operating system. The attack is assumed to have unlimited computing resources and can exploit zero-day vulnerabilities of desktop applications. We have a similar threat model to VMI systems as in [32–34], but since we do not rely on the operating system or hypervisor to accomplish the inspection task, we do not need to trust the hypervisor or the OS. We assume SMM is locked and will remain intact after boot, and the attacker cannot change the SMI handler or flash the BIOS and reboot. Cache poisoning techniques that change the SMI handler [58] are out of the scope of this paper. We assume that the target machine is equipped with trusted boot hardware, such as a BIOS with Core Root of Trust for Measurement (CRTM) and a Trusted Platform Module (TPM) [67] to ensure the integrity of the SMI handler upon booting the system. We assume that the attacker does not have physical access to the machine. We also assume that the hardware can be trusted to function normally; malicious hardware (e.g., hardware trojans) is out of scope.

5.1.3 System Architecture

Figure 5.1 illustrates the operation of the proposed Spectre system, which consists of two machines. The *target machine* is the machine to be protected, and the *monitor machine* is responsible for receiving status messages from the target machine and triggering alerts. The whole inspection process consists of four major stages. First, the target machine

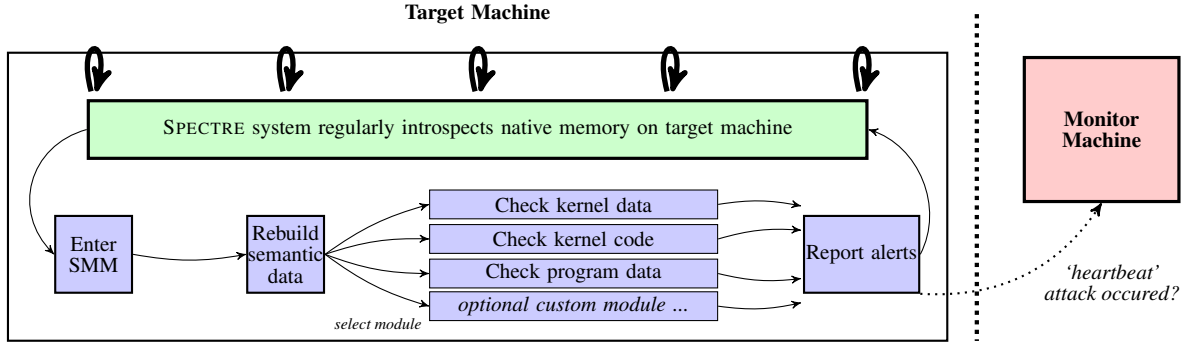


Figure 5.1: Operation of Spectre

enters SMM by triggering a system management interrupt (SMI) regularly and reliably. Second, after entering SMM, the target machine rebuilds accurate semantic information about the operating system in a trusted environment without relying on the (potentially altered) operating system. Third, the target machine executes monitoring modules that evaluate the integrity of the kernel or user-space processes. Finally, a “heartbeat” message is sent securely to the monitor machine. When a suspicious behavior is detected, an alert is transmitted as part of the heartbeat message.

SMM gives us several useful properties. First, it has unrestricted access to the whole physical memory in the system, so it is difficult for stealthy, malicious code to hide itself. Second, the SMI handler is loaded only once when the computer is powered up and locked thereafter. Thus, even if the malicious code can rewrite the BIOS or SMI handler, it will not be able to execute that code until rebooting. However, I can use TPM to prevent this attack by checking the integrity of the BIOS and SMM code before booting up. Third, the SMI handler can quickly inspect memory because it executes atomically and benefits tremendously from locality optimizations. Lastly, SMM provides a region of secure memory (SMRAM) in which I store data each time the SMI handler runs. This secure memory allows us to inspect system memory without having to trust the underlying operating system for storing relevant data. This protection is ensured transparently by the memory management unit (MMU), which redirects accesses to SMRAM addresses to a portion of video memory.

Periodic Triggering of System Management Mode

I periodically assert a system management interrupt (SMI) on the target machine so that it can enter SMM. There are two ways to trigger an SMI: software-based and hardware-based. Software can cause a SMI via I/O access to a particular port specified by the chipset. Software-based mechanisms are convenient and easy to implement, but they are not transparent to the operating system. Therefore, if the OS becomes compromised, malicious code can interfere with software and prevent it from accessing those special ports.

Alternatively, many hardware devices are also capable of triggering an SMI, including PCI devices, keyboards, and hardware timers. My system utilizes a hardware timer built into the chipset, which is capable of generating an SMI at a regular and configurable interval. I will set the timer configuration parameters in the BIOS before the OS loads, so I can trust the timer after booting. Nonetheless, advanced malware may be able to change these configuration settings after compromising the OS. This would effectively result in a denial of service. However, a monitor machine can trivially detect denials of service — it expects “heartbeat” messages to be sent at regular intervals. If these heartbeats cease, then the monitor machine can detect a denial-of-service attack. Additionally, I can prevent masquerading attacks by using a key exchange in the BIOS before the OS loads.

Rebuilding Semantic Information

Since SMM has unrestricted access to all physical memory and registers, the target machine can introspect all its physical memory once it enters SMM. However, only physical address space is visible in SMM, so I must reconstruct the semantics of various operating system structures in order to evaluate data and code integrity of the kernel and user-space programs.

When the SMI handler is first triggered, the CPU context that is saved contains virtual addresses relevant to the running thread. Moreover, in both Windows and Linux, many kernel structures reference virtual addresses. Thus, I should first be able to translate virtual addresses to physical addresses so that I can access important data from within the SMI handler. Fortunately, this process is identical in both Windows and Linux.

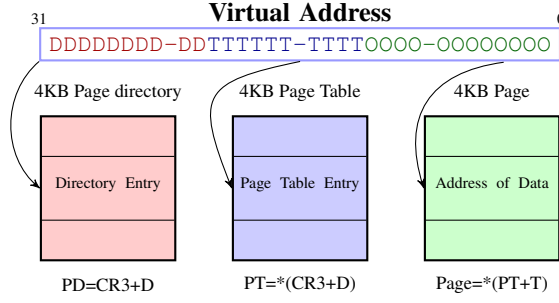


Figure 5.2: Translating Addresses from Virtual to Physical

Both operating systems reserve a large section of virtual address space for kernel mode operations. Addresses above a constant, `PAGE_OFFSET`, are considered kernel space. `PAGE_OFFSET` is `0xC0000000` for Linux, and `0x80000000` for Windows. In either case, finding the physical address in the kernel space simply consists of subtracting `PAGE_OFFSET` from the virtual address.

For user-space virtual addresses (VA), both operating systems use a two-level paging scheme on my test bed. Figure 5.2 shows the process of translating addresses from virtual to physical. There is a page directory containing pointers to particular page tables, which in turn point to specific pages. Each 32-bit virtual address is split into three regions: a 10-bit page directory offset, a 10-bit page table offset, and a 12-bit page offset. The CR3 register points to the base of the page directory. From there, the first 10 bits of the virtual address are used to find an entry in the directory, which points to the base of a page table. The next 10 bits of the virtual address are used as an offset into the page table, yielding a pointer to the page where the required data is stored. Physical Address Extension (PAE) essentially adds a fourth level of translation, which could easily be integrated into my system.

Memory Checking Modules

This research is designed to easily accommodate various existing defensive technologies. I will demonstrate this capability with several *modules* that detect an array of attacks,

including heap spray, heap overflow, and rootkits. Other checking modules can be extended into the SMI handler of my system.

My heap spray detection module regularly scans the heaps of vulnerable processes in memory. When a heap spray attack occurs, it will fill the heap with a NOP sled. When detecting a large region of NOP bytes, I will conclude that a heap spray attack has occurred.

This research will detect heap overflow attacks by evaluating the integrity of heap-related structures in the operating system. Typically, heap overflow attacks will alter entries in the *free list* maintained by the operating system [23]. This structure helps the OS track, from which blocks of the heap have been freed by the program for reallocation. A heap overflow attack will overflow the boundary of a heap buffer and rewrite data contained in an adjacent free block. This behavior will cause inconsistencies in the free list for which I can easily scan.

Once installed, rootkits pose a serious threat to a system's health. Nonetheless, in order to execute any code, they must alter the system memory in some detectable way, such as corrupting the list of processes. Ultimately, I detect rootkits by evaluating the integrity of kernel structures.

Reporting Alerts

When detecting an attack, the SMI handler alerts the monitor machine over a serial or Ethernet cable. I must ensure that communicating with the monitor machine is secure. There are two requirements to establish a trusted connection between the target machine and the monitor machine: One is a shared secret key between the target and the monitor machines, and the other is a trusted network interface on the target machine. The target machine can establish a shared secret key with the monitor machine in the BIOS before booting the OS. Since I trust the BIOS at startup, I can store the key in the trusted SMRAM. This key is then rendered inaccessible from other execution modes; only my SMI handler has access to it. This allows us to ensure that an attack cannot masquerade as my system to the monitor machine.

Since I cannot rely on the target machine’s untrusted operating system to relay the alert message to the monitor machine, my approach involves writing driver code within the SMI handler for my network card. I will use two separate network interfaces in my testbed: one for normal network usage and one exclusively for use by my SMI handler. This approach makes my system more transparent to the operating system. With no driver installed in the OS, software is unaware of its presence. My system can remain undetected while operating naturally at the expense of a PCI slot. However, while a compromised OS can scan the PCI device and write a new driver to operate the network card, it still cannot fake the network packet without the shared secret key. Thus, reporting alerts of my system are secure.

Moreover, my system can detect denial-of-service attacks that may occur if my system becomes compromised. Since the monitor machine will expect to receive “heartbeat” updates from the target machine at regular intervals, it is trivial to detect aberrations in packet delivery time.

5.1.4 Implementation

Spectre supports both Windows and Linux OS environments. In our testbed, the target machine has a ASUS-M2V_MX SE motherboard with AMD K8 northbridge and VIA VT8237R southbridge, 2.2GHz AMD Sempron LE-1250 CPU, and 2GB Kingston DDR2 RAM. We use the integrated network card for normal network traffic and an Intel e1000 Gigabit network card for SMM packet transmission. Additionally, the monitor machine consists of a simple Linux machine. It runs an instance of minicom for communication via the serial port and a simple socket program to receive network packets from the target machine. Its specifications are inconsequential to the performance of the system. We use an open source BIOS, Coreboot with a SeaBIOS payload. For a Linux environment, we use CentOS 5.5 with kernel 2.6.24 and Debian with kernel 2.6.32. For a Windows environment, we use Windows XP SP3. Each environment is 32 bit; however, our system is also capable of running in a 64-bit environment with slight changes in the paging system.

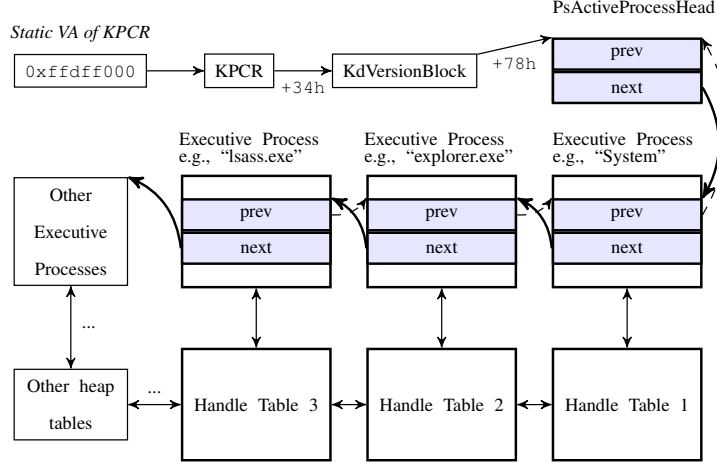


Figure 5.3: Finding the List of Processes in Windows

Periodically Triggering SMM

We use a hardware timer, General Purpose 0 (GP0), to periodically assert a system management interrupt (SMI) [68]. The timer is configured via control registers in the southbridge, which we set in Coreboot before the OS loads. This timer is configured with a starting value and unit of time. When the specified unit of time elapses, the timer value decrements by 1. Upon reaching 0, the timer will assert an SMI and then reset its value, restarting the process again. For example, when we assign the timer a value of 5 and a time unit of 1 second, it will trigger a SMI every 5 seconds.

Though a software-based SMI triggering mechanism would be easier to use, it is not usable for two reasons. First, software-based triggering would require extending our trust base to the operating system—malware that compromises the OS is able to stop the triggering software. Secondly, since it is software, the exact timing is left to the mercy of the OS scheduler. If the software trigger does not get scheduled due to high multiprogramming in the OS, then we may unintentionally suffer a denial of service. Thus, we choose the hardware-based approach that is much more reliable and transparent.

Rebuilding Semantic Information

Spectre performs inspection operations within SMM, which is agnostic to the particular operating system. However, in order to introspect the integrity of the OS, we have to fill the semantic gap on the data structures, which are different for Windows and Linux environments. We elaborate on this below.

Microsoft Windows maintains a complex hierarchy of data structures responsible for processes and threads. In particular, each CPU is associated with a Kernel Processor Control Region (KPCR) [69]. This data is always present at a static virtual address, 0xffdf000, in memory. Thus, if we can translate that virtual address to a physical address, we can easily access it from within the SMI handler. Fortunately, the CR3 register stores the physical address of the page directory of the currently executing process. This allows us to find the physical address corresponding to any given virtual address used by that process, including the KPCR. While each process has a unique CR3 value, SMM saves the CR3 register when switching from protected mode. Therefore, we can simply read the CR3 value from SMRAM to find the KPCR regardless of what the OS was doing before the SMI occurred.

At offset 0x34 of the KPCR, there is a pointer to another structure, the `KdVersionBlock`, which contains certain global variables pertinent to the current version of the kernel. Within the `KdVersionBlock`, the address of `PsActiveProcessHead` is stored at offset 0x78. `PsActiveProcessHead` is a pointer to the start of a circularly- and doubly-linked list of pointers to executive process structures, which we then use to find the heap of each process. Additionally, the executive process structure provides forward and backward links to other executive process structures. Figure 5.3 provides a visual representation of the structures we must traverse to find `PsActiveProcessHead` and the executive process structures to which it links.

Unfortunately, some rootkits are capable of altering this linked list to hide themselves through a technique called Direct Kernel Object Manipulation (DKOM) process hiding. This means we must consider alternative ways of enumerating processes. We consider a

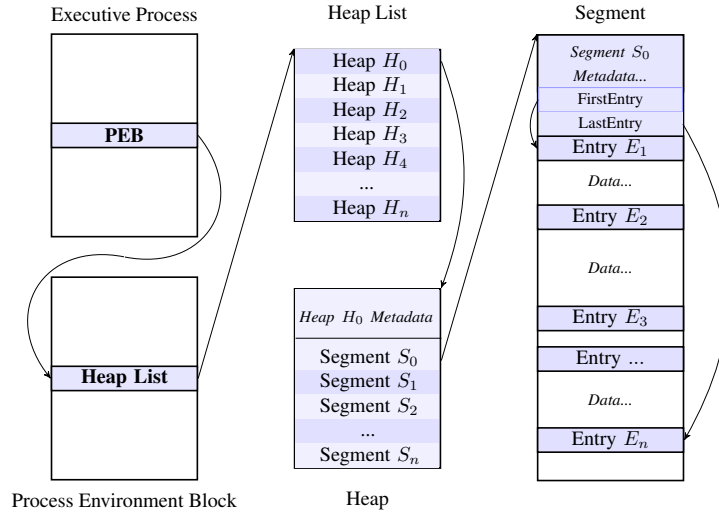


Figure 5.4: Finding Heap Data in Windows

method used by a program called KProcCheck [70].

First, we can use the method above to find the first executive process running on the system (called the `PsInitialSystemProcess`). This executive process is located at a fixed address, so we only need to find it once when first starting the system. Even if a rootkit removes this process from the linked list, we can still retrieve it from within the SMI handler.

Next, within the executive process structure, there is a `HANDLE_TABLE` structure containing information about that process’s files, devices, ports, and similar handle objects. This structure contains a `HandleTableList` consisting of backward and forward links to handle tables of other processes. This means we can enumerate each handle table for every process running on the system, regardless of whether a given process has been hidden. Additionally, the `HANDLE_TABLE` structure also contains a pointer to the executive process to which it belongs. Thus, even if a rootkit uses DKOM hiding, we can still find the executive process it tries to hide using this method.

Using the methods described above, we can enumerate all of the processes running on the system. Each executive process structure exists in kernel space of a particular process. It contains the name of the binary on the filesystem (e.g., “firefox.exe”). This allows us

to find and analyze a specific process, regardless of which process is running when SMM is triggered. While rootkits would be able to change the name of the process, we would be able to detect it via simple integrity checking. We can store the name of the image in SMRAM and check if that same executive process changes names during the next run.

Each executive process contains a pointer to a Process Environment Block (PEB), which is a user-space structure that stores the locations of heap structures belonging to that process. Each process has at least one default heap and can optionally create additional private heaps as needed. Pointers to each heap are stored in the process's PEB. Each heap structure contains additional pointers to a maximum of 64 heap segments, each of which stores a sequence of heap entries. The entries contain 8 bytes of metadata followed by the actual heap data. The entries in a segment are stored contiguously in virtual memory. Figure 5.4 illustrates the hierarchy of data structures we must traverse to enumerate entries in the heap.

Linux offers a much simpler process management scheme. There is a circularly- and doubly-linked list of `task_struct` structures, each of which contains information about processes running in the system. We can enumerate all of the processes by finding a single `task_struct` and walking through the list.

We leverage the kernel-exported symbol information to find a starting `task_struct` and then enumerate all of the processes. Firstly, we find the virtual address of the `init_task` pointer from the `System.map` file in the `/boot` directory. The `System.map` file is produced once when the kernel is compiled; it stores all of the symbol information about the kernel. `init_task` is a static address that points to the `task_struct` of a specific process, `swapper`. Within this `task_struct`, we can find forward and backward links that form a circularly- and doubly-linked list of tasks at offset 0x178 in our kernel. Additionally, offset 0x29b contains the name of the process, which helps identify specific processes. Figure 5.5 illustrates these steps to find the list of tasks in Linux. Similarly, we use the `modules` symbol in `System.map` to find the list of kernel modules.

Since the `task_struct` list is used by the scheduler, Linux rootkits cannot hide by

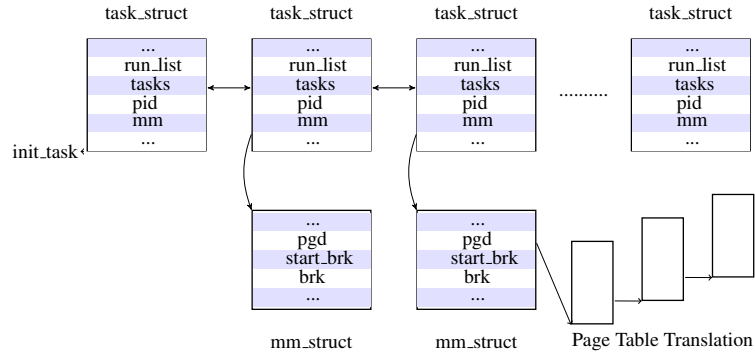


Figure 5.5: Finding the List of Tasks in Linux

altering this list—otherwise, they might impact their own execution. Instead, they often hide by altering the `/proc` directory.

Once we have the pointer to a task, all information related to the process address space is included in an object called the memory descriptor, `mm`. The `mm` field stores the pointer to a memory structure, called `mm_struct`, for the process. The `mm_struct` structure contains `start_brk` and `brk` fields, which correspond to the starting and ending addresses of the heap.

In contrast to Windows, the Linux environment simply allocates heap space one page at a time (via the `sbrk` system call). Typically, applications use heap allocators built into libraries like glibc, and thus malware typically exploits vulnerabilities in a particular heap allocator. For example, glibc uses a similar free list structure as in Windows—16 byte total metadata in free entries with forward and backward links to other free entries of the same size. Thus, the glibc allocator has free list vulnerabilities similar to those in Windows.

Running a Detection Module

Once we glean relevant semantic information from the operating system, we can start executing a module for system inspection. Our system is flexible to easily accommodate various existing defensive technologies. We demonstrate this capability with several modules that

can detect various memory-based attacks, including heap spray attacks, heap overflow attacks, and rootkits. Note that we acknowledge the simplicity of these detection algorithms that should not be considered as major contributions of this paper; our goal is to show the flexibility of our system framework to accommodate various checking modules. Other checking modules can be extended into the SMI handler of our system.

Once we have access to heap data, detecting a heap spray is the same for both Windows and Linux environments. We scan the heap for the presence of a potential NOP sled. Unfortunately, the x86 NOP instruction, `0x90`, is not the only technique used to achieve NOP-like behavior. Other common instructions include `or al, 0x0c` and `xor eax, eax`. In fact, many repeated sequences of bytes exhibit the behavior of a NOP sled, provided they do not affect the registers required for the shellcode to execute. Therefore, we heuristically check for the presence of a NOP sled by searching for contiguous, repeated sequences of bytes in the heap of a process.

Essentially, we wrote a regular expression engine in the SMI handler that recognizes the following pattern: $[\sim(0x00|0xFF)]\{n,\}$. This pattern will recognize a sequence of at least n or more repeated bytes other than `0x00` or `0xFF`. Naturally, changing the value of n will affect the false positive rate.

In Windows, an application can have multiple heaps, and each heap has a free list array with 128 elements called the `FreeList`. We can find this array at offset `0x178` from the heap base. Each `FreeList` is a list of free chunks chained by a doubly-linked list. Each free chunk has 16 bytes meta data including sizes of and pointers to the previous and current free chunks. In Linux, heap management is provided by a library (e.g., `glibc`), but the free blocks are chained by doubly-linked lists and use the same 16-byte header structure. The attacks exploiting the `FreeList` depend on the specific heap implementation, but the malicious code must change pointers to hijack execution. Our system transverses all entries in each heap's free list to see if there are any broken points. We did not implement a heap overflow detection module for Linux because heap free blocks are maintained by the `glibc` library. This adds another layer of the semantic gap problem for reconstructing heap

structures. We have considered it as a future work.

Detecting rootkits depends upon monitoring the integrity of 1) kernel code, and 2) kernel data. To check the integrity of the kernel code, we simply compute a hash of the static kernel code within the SMI handler. Alternatively, we send the static kernel code to a remote server for integrity attestation. Remote checking may be favorable in environments where consumption of network bandwidth is less expensive than local hash computation. Since the SMI handler essentially pauses the native system, we want to avoid overly long computation in the SMI handler to avoid incurring too much overhead.

The more challenging aspect is maintaining integrity of dynamic data structures in the kernel. Previous research has proposed many defensive techniques against rootkits [19, 32, 33]. In order to demonstrate this capability in our system, we wrote a simple rootkit detection module of listing all running processes (`pslist`) and kernel modules (`lsmod`) for both Microsoft Windows and Linux platforms. We leverage the security caveats of SMM to bring accurate semantic information from the operating system to a trusted ‘external’ viewpoint. We can discover rootkits by comparing these external views with the internal views of the operating system process states.

Communication with the Monitor Server

The last stage of our system requires communicating with an external server. We accomplish this task by writing driver code for our particular network card in the SMI handler. It consists of manually configuring registers on the device and interacting with the PCI bus.

In brief, we implemented a simple MAC-layer protocol for communicating with the external server. It sends a 214 byte packet in the SMI handler consisting of a 14-byte header and a fixed 200-byte payload. The payload is encrypted with a simple XOR with a key we store in SMRAM, which is first retrieved before the OS loads. The payload contains a sequencing number which simply increments by 1 each time the SMI handler runs. The rest of the payload provides enough spaces for detection modules to convey specific information to the monitor machine.

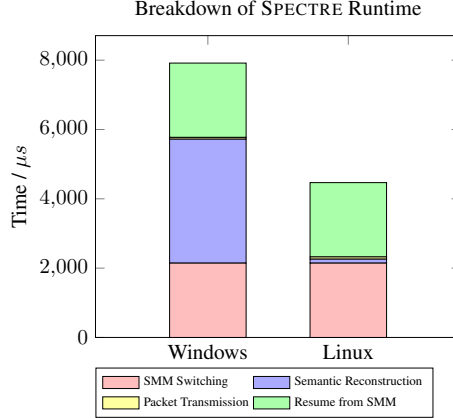


Figure 5.6: Breakdown of SMI Handler Runtime

5.1.5 Evaluation

Code Size

First, we considered the size of the code required for our system to run. In total, there are 470 lines of new C code in the SMI handler, including all three memory checking modules. Each module consisted of less than 100 lines of C code, and the total network transmission code was 110 lines. After compiling the Coreboot code, the binary size of our SMI handler was only 780 bytes, which reduced the trusted computing base of our system.

Breakdown of SMI Handler Runtime

Next, it is important to quantify how much time is required to execute each step of our system. For this experiment, we have broken Spectre into the following logical operations: 1) Switching to SMM; 2) Reconstructing OS semantics; 3) Running a detection module; 4) Reporting status via NIC; and 5) Switching from SMM to resume the OS. All of the above except for step 3 should take constant amounts of time. The running time of a detection module will depend upon the type of attack and the complexity of the detection technique. For example, the time taken to traverse a linked list of processes will depend upon how many processes are running (i.e., the length of the list) when the module begins executing.

However, the rest of the steps execute a fixed set of instructions, and thus we expect them to have somewhat constant running times. In this section, we wanted to understand the ‘fixed cost’ associated with using our system. In other words, *how much time does Spectre need to bring useful semantic information to the developer?* Thus, we considered each of the times associated with steps 1, 2, 4, and 5. Step 3 (running detection modules) is discussed later.

We measured the time taken by each step by measuring the TSC register, which stores how many CPU cycles have elapsed since powering on. We disabled technologies in the BIOS that affected CPU clock speed so that a difference in the TSC register represented a constant unit of time, computed with the equation,

$$T = (R_1 - R_0)(\frac{1}{C}),$$

where T is measured time, R_t is the value of the TSC register at time t , and C is the clock speed on the CPU. We recorded the TSC register at several points during our system’s execution, such as the beginning and end of the SMI handler.

Figure 5.6 shows the observed times taken for each step in each operating system. We can see from the graph that switching to and resuming from SMM take a significant amount of time. This is attributed to the power management operations that SMM must perform before our SMI handler can execute. Similarly, the time to resume from SMM is explained by several factors. Upon resuming from SMM, the hardware must also reconfigure itself to allow subsequent SMIs to occur, which requires many I/O operations and thus leads to a considerable running time.

Note the significant difference in the time taken for reconstructing the semantics of each operating system. Reconstructing Windows kernel semantics is much longer than in Linux (by two orders of magnitude). This is mainly due to the page table translation steps required in Windows since so much of the data about processes is stored in userspace. In Linux, however, most of the required data is stored in kernel space, and therefore finding the physical addresses reduces to simple subtraction of the `PAGE_OFFSET` constant.

Heap Spray Detection Module

We implemented a heap spray detection module as described previously. We tested Firefox, Adobe Acrobat Reader, and the Adobe Flash plugin in both Windows and Linux, but since MSIE is not available for Linux, we only tested it in Windows.

We chose four heap spray attacks available as Metasploit modules. Using Metasploit eased the experimentation because it allowed rapid deployment of each attack. Each attack has a corresponding Common Vulnerabilities and Exposures entry. The attacks we used are:

1. Firefox 3.5 CVE-2009-2478
2. Internet Explorer 6, 7, 8 CVE-2010-3971
3. Adobe Acrobat 9, 10.1 CVE-2011-2462
4. Adobe Flash Player < 10.2 CVE-2011-6069

These attacks all exploit a vulnerability in an application that causes it to start executing code in the heap. They are all written in scripting languages. The first three attacks use JavaScript, and the last uses ActionScript. They cause the host program to start executing the malicious script, which causes it to allocate large amounts of memory. Then, the attack hijacks control through another means (use-after-free, stack overflow, etc.) to start executing the sprayed memory. We ran this experiment in Windows but not Linux because the dynamic memory system in Windows is much more complex, and thus provides a ‘worst-case’ performance figure. Table 5.1 shows the average results for 25 trials of each type of heap spray attack. The results shows that Spectre can detect these attacks in less than 32 ms.

Heap Overflow Detection Module

We tested our system against CVE-2012-0276, a real heap overflow attack affecting an image viewer in Windows called XnView. The vulnerability exists in XnView versions 1.98 and

Table 5.1: Heap Spray Attack Detection Time (n=25)

	Detection Time	STD
Firefox	31.168 ms	0.272 ms
Internet Explorer	27.917 ms	0.154 ms
Adobe Acrobat Reader	25.839 ms	0.302 ms
Adobe Flash	29.455 ms	0.603 ms

earlier. In XnView, insufficient validation while decompressing certain TIFF files enables a heap-based buffer overflow. The malicious image overflows a heap entry and then it rewrites metadata of nearby free chunks in the heap. Then, it simply waits for these blocks to be reused. When the operating system unlinks one of these free blocks from the FreeList, execution jumps to the shellcode.

We detected this attack by checking the integrity of the FreeList, and we found that it takes 32 ms to detect this attack including 24 ms spent in the detection module and the fixed 8 ms associated with entering and exiting SMM.

Rootkit Detection Module

We used real, publicly available rootkits to test out system on both Windows and Linux platforms. On Windows platforms, we devised an effective defense mechanism against the Fu rootkit [71]. Fu Rootkits allow the intruder to hide information from user-mode applications and even from kernel modules. Fu hides information by directly manipulating data structures in the kernel. In particular, it removes an entry from the `PsActiveProcessHeader` list. However, we are able to find such hidden processes by finding and traversing the `HANDLE_TABLE` list.

We successfully detected the Fu rootkit using this method. On the target machine, Fu hides the `ssh.exe` process. We detected the hidden process by enumerating the handle tables in the SMI handler. This technique took only 8ms.

On the Linux platform, we tested a newly available rootkit, KBeast (Kernel Beast), on kernel 2.6.32. KBeast is an advanced armored Linux rootkit that hides its loadable kernel

module, hides files and directories, hides processes, hides sockets and connections, performs keystroke logging, and has anti-kill functionality [72]. It is currently undetectable by the latest rootkit detectors including *chkrootkit* [73] and *rkhunter* [74]. KBeast leverages the `sys_write` system call to fake output of system commands like `ps`, `ps tree`, `top`, and `ls of` to hide itself. Again, since SMM has an external view of the system states, our system reconstructs the semantics of data structures from physical memory to detect malicious behavior like process hiding.

We were able to detect KBeast in about 5ms using our system. Using the `ps` within the OS missed a network daemon process for malicious remote access. However, Spectre successfully discovered the hidden process by traversing the process list in the system.

System Overhead

The last and most important experiment tested how much overhead our system introduces to the target machine. We used freely available benchmarking software for both Microsoft Windows and Linux environments. This helped us account for the impact on overall system performance caused by our system’s periodic operation. For this experiment, we ran the benchmarking software without our system in place. Next, we ran the same benchmark with Spectre enabled at several different time intervals ranging from 0.0625 to 5.0 seconds using the General Purpose 0 (GP0) hardware timer on the southbridge to periodically trigger a SMI. We then calculated the overhead as a ratio between the scores with and without the system in place. In this experiment, the heap spray detection module targeted the heap of the Adobe Acrobat Reader application in both Windows and Linux. The heap overflow detection module targeted the heap of the XnView process in Windows. We did not consider a Linux-based heap overflow detection module because it requires reconstructing another layer of semantic information from the particular heap allocator used by a process. Lastly, the rootkit detection module listed all of the running processes in memory to find hidden processes.

In Windows, we used PassMark PerformanceTest to measure the benchmark on our

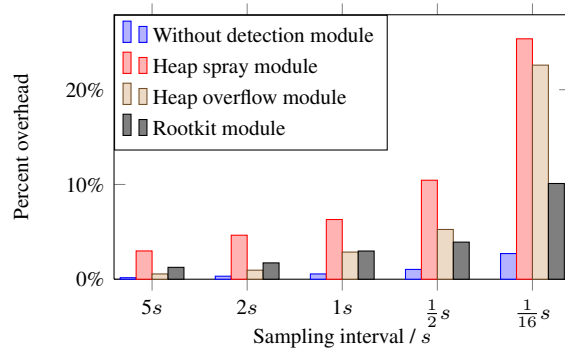


Figure 5.7: Overhead Introduced in Microsoft Windows

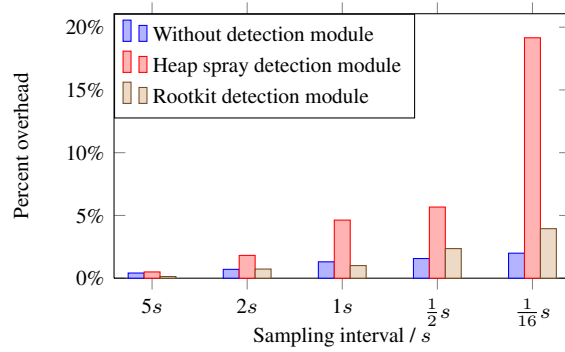


Figure 5.8: Overhead Introduced in Linux

test system. We specifically ran the CPU, disk, and memory tests in PassMark to see the implications on raw performance. Figure 5.7 shows the results of this experiment. These results indicate the relatively low overhead introduced at all sampling intervals. From Figure 5.7, we can see that the heap spray and heap overflow modules have slightly larger overhead than the rootkit detection module. This is because the heap-based modules must scan heap data, which takes roughly 30ms. Rootkit detection, on the other hand, simply scans the list of running tasks in memory. This task takes only 8ms in the SMI handler.

We used a similar methodology to test the overhead in Linux. We used the UnixBench suite to test performance while the system ran. This was less geared toward CPU and

memory performance, instead focusing on specific Unix-like operations, like system call and shell piping performance. The results are presented Figure 5.8. In general, Spectre introduces low overhead in Linux. Even at the lowest sampling interval of $\frac{1}{16}s$ (62.5ms), it causes only 20% overhead in the heap spray detection module, and only 5% overhead in the rootkit detection module.

Comparison with VMI Systems

Spectre provides a new framework for transparent system introspection and stealthy malware detection. Compared to well-known virtual machine introspection based architectures [4], the BIOS in Spectre serves a role similar to the hypervisor in VMI systems. Theoretically, Spectre can achieve the same level of protection as VMI if 1) we are able to implement and execute the same detection algorithms in SMRAM, and 2) we are able to reconstruct all of the necessary kernel- and user-space data structures that serve as the input to the detection algorithms. In this paper, we show several ways to include different detection modules and recover the necessary semantic data in Spectre.

Spectre improves upon VMI systems in three ways. First, Spectre is a hardware-assisted introspection tool which relies only on the BIOS—it does not need to trust the large-size hypervisor. Thus, Spectre has a much smaller TCB. Second, Spectre can achieve better transparency than VMI systems. Nowadays, armored malware [11, 14, 15, 18] can easily detect the presence of a VM, but Spectre can remain transparent while monitoring these malware. Third, Spectre achieves better performance because it does not need to deal with nested page table translation, and SMM switching is faster than VM switching. Table 5.2 shows the runtime comparison between Spectre and Virtuoso [33]. The program `pslist` shows all of the running process information in the OS, and the program `lsmod` shows all of the loaded kernel modules. The results show that Spectre can run these tools 100 times faster than those in Virtuoso. Recently, hardware virtualization extensions (e.g., Intel VT, AMD-V) have been adopted to VMI systems to speed up the introspection process.

VMI systems can operate based on trap conditions, allowing asynchronous, event-based

Table 5.2: Runtime Comparison of Introspection Programs Between Spectre and Virtuoso

		Spectre (ms)	Virtuoso (ms)
Windows	pslist	6.6	450.2
	lsmod	7.6	698.1
Linux	pslist	4.3	6494.1
	lsmod	4.4	2437.0

tools. The current Spectre prototype can only execute periodically, but it is a straightforward engineering challenge to implement similar functionality by using performance counters to trigger SMIs. We can assert SMIs when certain conditions are met in the CPU performance counters. For instance, when the instruction cache miss counter overflows, we can assert an SMI.

5.2 HyperCheck: Hypervisor-level Malware Detector

5.2.1 Introduction

The advent of cloud computing and inexpensive multi-core desktop architectures has led to the widespread adoption of virtualization technologies. Furthermore, security researchers embraced virtual machine monitors (VMMs) as a new mechanism to guarantee deep isolation of untrusted software components, which coupled with their popularity promoted VMMs as a prime target for exploitation. In this thesis, I present HyperCheck [75], a hardware-assisted tampering detection framework designed to protect the integrity of hypervisors and operating systems. It leverages System Management Mode (SMM), a CPU mode in x86 architecture, to transparently and securely acquire and transmit the full state of a protected machine to a remote server. I have implemented two prototypes based on our framework design *HyperCheck-I* and *HyperCheck-II*, which vary in their security assumptions and OS code dependence. In the experiments, I am able to identify rootkits that target the integrity of both hypervisors and operating systems. I show that HyperCheck can defend against attacks that attempt to evade our system. In terms of performance, we

measured that HyperCheck can communicate the entire static code of Xen hypervisor and CPU register states in less than 90 million CPU cycles, or 90 ms on a 1 GHz CPU. Next, I explain the threat model, architecture, implementation, and evaluation of Hypercheck.

5.2.2 Threat Model

Attacker’s Capabilities

The adversary is able to exploit vulnerabilities in any software running in the machine after booting. The software includes the VMM and all of its privileged components. For instance, the attacker can compromise a guest domain and escape to the privileged domain. When using PCI pass-through on Intel VT-d chipsets that do not have interrupt remapping, Xen 4.1 and 4.0 allow guest OS to gain host OS privileges by using DMA to generate malicious MSIs [76]. In Xen 3.0.3, pygrub [77] allows local users with elevated privileges in the guest domain (Domain U) to execute arbitrary commands in Domain 0 via a crafted `grub.conf` file [78]. In addition, the attacker can modify the hypervisor code or data using any known or zero-day attacks. For instance, the DMA attack [79] hijacks a device driver to perform unauthorized DMA accesses to the hypervisor’s code and data.

HyperCheck aims to detect OS rootkits or hypervisor rootkits. One kind of rootkit only modifies the memory and/or registers and runs in the kernel level. For instance, the `idt-hook` rootkit [80] modifies the interrupt descriptor table (IDT) in the memory and then gains control of the complete system. A stealthier version of the `idt-hook` rootkit (which we can call it copy-and-change attack) could keep the original IDT unchanged by copying it to a new location and altering it. Next, the attacker could change the IDTR register to point to the new location. Thus, a malicious interrupt handler would be executed when an interrupt occurs [81]. Our system could detect rootkits in an OS running on bare metal and rootkits in a native hypervisor.

General Assumptions

First of all, we assume BIOS is trusted. Since SMM code is loaded into SMRAM from the BIOS, we assume the SMRAM is properly set up by the BIOS while booting. To secure the BIOS code [82, 83], we can use a signed-BIOS mechanism to prevent any modification of the BIOS code, but this method requires that the BIOS updating process is securely implemented and trusted. An alternative way to secure the BIOS is to use Static Root of Trust Measurement (SRTM) to perform a trusted boot, and it requires that the Core Root of Trust Measurement (CRTM) is trusted and secure. The SMRAM is locked after booting into the OS. Once it is locked, we assume it cannot be subverted by the attacker (an assumption supported by current hardware). Furthermore, we assume attackers do not have physical access to our system.

Currently, our system cannot protect against attacks that modify dynamic data, such as modification of dynamically generated function pointers and return-oriented programming attacks. In these attacks, the control flow is redirected to a memory location controlled by the attackers. HyperCheck can leverage existing solutions (e.g., Address Space Layout Randomization (ASLR) [84, 85]) to prevent or mitigate such attacks; however, it is not the focus of this paper.

5.2.3 System Architecture

HyperCheck is composed of three key components: the physical memory acquisition module, the analysis module, and the CPU register checking module. Both the physical memory acquisition module and the CPU register checking module are on the target machine, and the analysis model is on the monitor machine. The memory acquisition module reads the memory contents of the protected machine and sends it to the analysis module, which then checks the memory contents for any malicious alterations. The CPU register checking module reads the CPU registers and validates their values. The overall architecture of HyperCheck is shown in Figure 5.9.

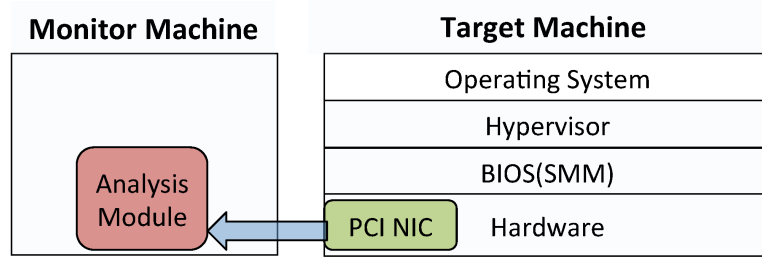


Figure 5.9: Architecture of HyperCheck

Acquiring Physical Memory

In HyperCheck, I will choose the hardware-based method to read the physical memory. There are several options for hardware components, such as PCI devices, FireWire bus devices, or a customized chipset. I will use a PCI network card because it is the most popular and commonly used hardware device. Note that existing commercial Ethernet cards need to install device drivers, and these drivers normally run in the OS or the driver domain, which is vulnerable to the attacks and may be compromised in my threat model. To avoid this problem, HyperCheck moves these device drivers into the SMI handler, which is inaccessible to the attackers after the SMRAM is locked. In addition, to prevent a malicious NIC from spoofing the NIC driver in SMM, I will use a secret key to authenticate the transmitted packets. The key can be obtained from the monitor machine while the target machine is booting up and then stored in the SMRAM. The key is used as a random seed to selectively hash a small portion of the data to avoid data replay attacks.

Another class of attacks is denial-of-service (DoS) attacks. This attack aims to stop or disable the device. For instance, according to the ACPI [86] specification, every PCI device supports the D3 state. This means that an ACPI-compatible device can be suspended by attackers who control the OS. Since neither the hypervisor nor the OS are trusted components in my framework, one possible attack is to selectively stop the NIC without stopping any other hardware. To prevent ACPI DoS attacks, I need an out-of-band mechanism to verify that the PCI card is not disabled. The monitor machine receiving the state snapshots

plays this role.

Since paging is not enabled in SMM, HyperCheck uses the CR3 register to translate the virtual memory addresses used by the OS kernel to the physical memory addresses used by the SMM. Since the acquisition module relies on physical addresses to read the memory contents, HyperCheck needs to find the physical address of the protected hypervisor and privileged domain. One method is to use the system.map file, which HyperCheck uses to obtain the virtual addresses of monitoring symbols. However, I believe there are many other ways to obtain these addresses. For instance, the system call table address can be found by using the interrupt vector of the INT 0x80 system call [87]. From the symbol files, HyperCheck first reads the virtual addresses of the target memory and then utilizes CR3 register to find the physical addresses corresponding to the virtual ones. Another possible way to get the physical addresses without using page table translation is to scan the entire physical memory and use pattern matching to find all potential targets. However, this method is not efficient because hypervisors and OS kernels have a small memory footprint.

Furthermore, HyperCheck should be able to check the integrity of any software above the BIOS. Although I focus on the Xen hypervisor in this paper, HyperCheck also can be used to check KVM or other hypervisors. Some operating systems use Address Space Layout Randomization (ASLR) in kernel booting (e.g., Windows 7 [84]), and it adds a fixed offset when setting up virtual address space. For example, Kernel Processor Control Region (KPCR) is located at a fixed virtual address 0xffdff000 in Windows XP and Windows 2000. In Windows 7, KPCR structure is no longer at a fixed address. However, researchers have demonstrated that the KPCR structure can be acquired by conditional searching of physical memory [88]. After obtaining the KPCR structure, I am able to bridge the semantic gap in the physical memory and identify the targeting memory contents.

Analyzing Memory Content

In practice, there is a semantic gap between the physical memory addresses in SMM that I will monitor and the virtual memory addresses used by the hypervisor or the OS. To verify

the memory contents, the analysis module must be aware of the semantics of the memory layout, which depends on the specific hypervisor or the OS I monitor. The current analysis module depends on three properties of the kernel (OS or hypervisor) memory: linearity, stability, and perpetuity.

The linearity property means the kernel virtual memory is linearly mapped to physical memory and the offset is fixed. For instance, on x86 architecture, the virtual memory of Xen hypervisor is linearly mapped into the physical memory. In order to translate the virtual address to the physical address in Xen, I only need to subtract the virtual address from an offset. In addition, Domain 0 of Xen is also linearly mapped to the physical memory. The offset for Domain 0 is machine-dependent but remains the same on any given machine. Moreover, other OS kernels, such as Windows [89], Linux [90], and OpenBSD [54], also have this property when they are directly running on bare metal.

The stability property means that the contents of monitoring memory must be static. If the contents are changing, there might be a time window between when the memory changes and when my acquisition module reads them. This may result in inconsistency for analysis and verification. As a result, HyperCheck does not check on dynamic kernel data (e.g., kernel stack).

The perpetuity property means the memory used by hypervisors will not be swapped out to the hard disk. If the memory is swapped out, then I cannot identify or match any content by only reading the physical memory. I would have to read the swap files on the hard disk. For instance, Windows kernel code can be swapped to a disk. For this case, I have two solutions to read these swap pages in HyperCheck system. One method is to port a small disk driver in SMM to enable disk access. Then, I can use page table information to locate these pages on the disk and send them to the monitor machine for integrity checking. The other solution is simply to wait for the swapped pages to swap back into memory. Since HyperCheck enters SMM periodically, I can check the page table information to see if the pages have been swapped in. After these pages are present in memory, I will send them out in SMM. Additionally, I can force these pages to be swapped back into memory

by accessing them to generate page faults.

The HyperCheck relies on linearity, stability, and perpetuity to work correctly. Besides the Xen hypervisor, most OSes have these three properties, too.

Reading and Verifying CPU Registers

Since the PCI NIC card cannot read the CPU registers, I must use another method to read them. Fortunately, SMM can read and verify the CPU registers. When the CPU switches to SMM, it saves the register context in the SMRAM. The processor fetches the first instruction of the SMI handler at the address $[SMBASE + 0x8000]$ and stores the CPU states in the area from $[SMBASE + 0xFE00]$ to $[SMBASE + 0xFFFF]$ [63]. The default value of SMBASE is 0x30000. HyperCheck verifies the registers in SMM and reports the results via the Ethernet card to the monitor machine. HyperCheck focuses on monitoring two registers: IDTR and CR3. IDTR should never change after system initialization. CR3 is used by SMM code for memory address translation of the hypervisor kernel code and data. The offsets between physical addresses and virtual ones should never change as I discussed in Section 5.2.3.

5.2.4 Implementation

We implement two prototypes for HyperCheck on two physical machines: HyperCheck-I uses an original closed source BIOS, and HyperCheck-II uses an open source BIOS called Coreboot [20]. We first develop HyperCheck-I for quick prototyping and debugging in our previous conference publication. After that, we implement HyperCheck-II as an improved version of our previous prototype in terms of security and scalability.

HyperCheck-I follows the HyperCheck framework but uses two physical machines: one as the target machine and the other one as the monitor machine. On the target machine, we install Xen 3.1 natively and use Intel e1000 Ethernet card as the acquisition module. We modify the default SMM code in the original Dell BIOS on the target machine to transfer system states to the monitor machine. Since we use original BIOS with closed source code,

we need to apply reverse engineering methods to change the default SMI handler code [56] on the target machine. However, HyperCheck-I comes with two drawbacks. First, it needs to rely on an unlocked SMRAM to inject the customized SMI handler code and most machines today have locked their SMRAM. The other drawback of HyperCheck-I design is the high development complexity. Due to the time-consuming reverse engineering requirement and the usage of assembly language, it is difficult to add new functions into the BIOS. For instance, we have to use a kernel module to prepare the network transmit descriptors, instead of implementing all the functions in the BIOS. Therefore, we implement another HyperCheck prototype called HyperCheck-II using Coreboot, an open source BIOS.

HyperCheck-II also uses one physical machine as the target machine and another physical machine as the monitor machine. Coreboot can provide an unlocked SMRAM for us to add customized SMI handler code. HyperCheck-II locks the SMRAM in the Coreboot after booting. Since we can directly modify the BIOS code on the target machine, we can easily program the SMM code in the BIOS instead of performing reverse engineering of the BIOS in HyperCheck-I. In addition, we write C code in HyperCheck-II rather than assembly code in HyperCheck-I.

Memory Acquisition Module

HyperCheck uses a dedicated PCI network card to transfer the memory contents. In our prototype, we have two network interfaces on the target machine. We use an Intel e1000 network card to transfer the system states and the integrated network card for the normal traffic. When we implement the acquisition module, the main task is to port the e1000 network card driver into SMM to scan the memory and send the memory out to the monitor machine. Since HyperCheck-I does not have the source code of the BIOS, we use a similar method mentioned in [56] to modify the default SMM code in the BIOS. It writes the SMM code in 16-bit assembly code, uses a user-level program to open the SMRAM, and then copies the assembly code to the SMRAM. While HyperCheck-II uses the open source Coreboot as the BIOS, we have full control over the BIOS code. Thus, we can write C code

to port the e1000 NIC driver into the SMI handler of HyperCheck-II.

Both HyperCheck-I and HyperCheck-II split the e1000 NIC driver into two parts: initialization and data transferring. The initialization part is complex and similar to the Linux NIC driver. The data transferring part is much simpler than the NIC initialization part. Therefore, we modify the existing Linux e1000 NIC driver to only initialize the network card and move the packet transferring part into the SMI handler. In HyperCheck-I, we compile the assembly code of data transferring into an ELF object file, use a small loader to parse the ELF object file, and then load the code into SMRAM. In HyperCheck-II, we write the data transferring code in Coreboot directly, compile the BIOS code to a new ROM image, and flash the image into the BIOS chip of the target machine.

After porting the e1000 NIC driver into the SMM, we modify the driver to scan the memory and send the contents to the monitor machine. HyperCheck uses two transmission descriptors per packet, one for the packet header and the other for the packet data. The content of the header should be predefined. In our prototypes, there are 14 bytes in the header, which includes the source MAC address, destination MAC address, and two bytes of protocol type. Since the NIC has been initialized by the OS, the driver in SMM only needs to prepare the TX descriptor ring, and then write the index of last descriptor in the ring to the Transmit Descriptor Tail (TDT) register. The NIC would automatically send all of the packets in the TX descriptor ring to the monitor machine using DMA. The NIC driver also needs to prepare a header structure and point the header TX descriptors to this header. For the payload, the data descriptors directly point to the address of the memory that needs to be sent out.

To prevent replay attacks, a secret key is transferred from the monitor machine to the target machine during the booting of target machine. The key is used to create a random seed to selectively hash the data. If we hash the entire data stream, the performance impact may be high. To reduce the overhead, we use the secret key as a seed to generate one big random number used for one-time pad encryption and another set of serial random numbers. The serial random numbers are used as the indices of the positions of the memory. Then,

the contents at these positions are XORed with the big random number before starting NIC DMA. After the transmission is done, the memory contents received at the monitor machine are XORed again to restore the original value.

The NIC driver also checks the loop-back setting of the device before sending the packet. To further guarantee the data integrity, the NIC driver stays in the SMM until all of the packets have been written to the internal FIFO of the NIC. Then, it adds extra 16 KB data to the end to flush the internal 16 KB FIFO buffer of the NIC. Thus, the attacker cannot use loop-back mode to get the secret key or peek into the internal NIC buffer through debugging registers of the NIC.

Analysis Module

We use a direct Ethernet cable to connect the monitor machine and the target machine, and we assume that the monitor machine is trusted. Therefore, the target machine does not need to authenticate the monitor machine. If we connect the two machines through Internet, further authentication mechanism will be needed. On the monitor machine, we run `tcpdump` to capture the packets from the acquisition module and send the output of `tcpdump` to the analysis module. The analysis module is written in a Perl script that reads the input and checks for any alteration. First, the analysis module recovers the memory contents using the same secret key. Then, it compares two consecutive memory snapshots bit by bit. If they are different, the analysis module outputs an alert on the console. The administrator can decide whether it is a normal update of the hypervisor or an intrusion. Note that during the booting time of the system, the contents of the control data and code may change.

The analysis module checks the integrity of the static code and control data of Xen. The static code is Xen hypervisor code; the control data includes the IDT table, the hypercall table, and the exception table of Xen. To find the physical addresses of these control tables, we use `Xen.map` symbol file. First, we find the virtual addresses of `idt_table`, `hypercall_table` and `exception_table`. The physical addresses of these symbols are

equal to the virtual address minus fixed offset 0xff000000 on x86-32 bit architecture with PAE enabled. The address of Xen hypervisor code is from `_stext` to `_etext`. HyperCheck can also monitor the control data and static code of Domain 0. It includes the system call table and the code part of Domain 0 (Cent OS 5.3 uses a modified Linux 2.6.18 kernel). The kernel of Domain 0 is also linearly mapped to the physical memory. We use a kernel module running in Domain 0 to compute the exact offset. On our target machine, the offset is 0x83000000. Note that there is no IDT table for Domain 0, since there is only one such table in the hypervisor. We also use these parameters in the acquisition module to improve the scan efficiency.

CPU Register Checking Module

HyperCheck monitors IDTR and CR3 registers in CPU register checking module. The contents of IDTR should never change after system boots up. The SMM code can read this register by `lidt` instruction. HyperCheck uses CR3 to translate the virtual addresses to physical addresses. Essentially, it walks through all the page tables as a hardware Memory Management Unit (*MMU*) does. Note that offset between the virtual address and the physical address of hypervisor kernel code and data should never change due to the static mapping. For example, it is 0xff000000 for Xen 32 bit with PAE enabled. If any physical address is not equal to the virtual address minus the offset, it indicates a potential attack. The SMM code reports the checking result via the Ethernet card to the monitor machine.

From HyperCheck-I to HyperCheck-II

Since we cannot directly change the closed source BIOS in HyperCheck-I, the development and debugging complexity hinders the system extension with other functionalities and the verification of system security. Therefore, we are motivated to implement another HyperCheck prototype using Coreboot, an open source BIOS.

Similar to HyperCheck-I, HyperCheck-II reserves a small portion of memory by adding the boot parameter `mem=2000M` to the Xen hypervisor or Linux kernel. Since the total

memory size is 2048 MB, it saves 48 MB of memory to store the TX descriptor ring.

HyperCheck-II does not rely on any kernel modules but the trusted BIOS. After the system triggers SMI, it enters SMM and executes the SMI handler, which scans the memory, obtains the location of the memory, prepares the TX descriptors, and writes them to the TX descriptor ring in the reserved memory. Next, the NIC card reads the TX descriptor ring and sends out the data. After NIC finishes sending the data, the system exits SMM.

HyperCheck-II is more secure than HyperCheck-I. HyperCheck-I prepares the TX descriptor ring using a kernel module that may be compromised by attackers. Instead, HyperCheck-II can overcome this security flaw by preparing the TX descriptors in the SMI handler. Even if the attacker owns the OS, it still cannot manipulate the operations in the SMI handler that is securely loaded into the locked SMRAM. In addition, HyperCheck-I requires an unlocked SMRAM to load our customized SMM code; Coreboot can add a customized SMI handler and lock SMRAM before the system boots up. Additionally, HyperCheck-II is robust against evasion attacks, as discussed in the next section.

Evasion Attack Defense

Most polling-based integrity checking systems (e.g., [57, 62, 75]) are vulnerable to evasion attacks. If the polling time is detectable, an attacker can clean the attacking trace before the polling and then reload the malicious code. As discussed in our previous work [91], there are three types of evasion attacks for polling-based systems. The first type (Type-I) is to directly intercept the triggering events by detouring SMI invocation. The software-based SMI triggering writes to a specific port, and the attacker can locate the code (OUT %AL, %0XB2) in memory to detour the program execution before triggering SMI. The hardware-based SMI triggering method can be rerouted to normal interrupt and reissue another SMI triggering. The second type (Type-II) of evasion attack is to indirectly derive periodicity of the polling time by using an SMI detector. After knowing the polling time, an attacker can launch the evasion attack between two checks. The last type (Type-III) of evasion attack is to avoid random polling by running a small malicious task after exiting from SMM. This

type of evasion attack assumes the attacker knows when the SMM exits.

HyperCheck-I may suffer from evasion attacks; however, HyperCheck-II can prevent all three types of evasion attacks. We use Message Signaled Interrupts (MSI) in a PCI network card to invoke SMI. The delivery mode of MSI register in PCI configuration space can be set as the SMI mode. However, if the hypervisor or OS has been compromised, the attacker can modify MSI register to trigger a normal interrupt controlled by the attacker and then reissue an SMI (Type-I evasion attack). To trigger the same SMI, the attacker needs to know the MAC address of the dedicated NIC card and use another machine to send an authenticated packet to reissue the same SMI. Since the secret key is saved in SMRAM, the attacker cannot send the authenticated packet without knowing the secret key. Furthermore, the attacker may use other methods to trigger SMI, but we can prevent it by checking the reason for the triggered SMI, which has also been implemented in HyperSentry [62].

In order to defend against Type-II evasion attack, HyperCheck-II randomly triggers SMI using a dedicated commercial network card. We read `/dev/random` on the monitor machine as the pseudo-random generator seed and set a random delay between two SMI triggering packets. When an authenticated packet reaches the NIC interface, an SMI is generated by a Message Signaled Interrupt.

A Type-III evasion attack runs a small malicious task after SMM exits to avoid the random polling. However, this type of evasion attack needs to know when SMM exits. In HyperCheck-II, we include a random delay function in SMI handler, so the SMI handler will take various amounts of time between two checks. Thus, the attacker cannot accurately predict when SMM exits.

5.2.5 Evaluation

We evaluate the HyperCheck system on two different testbeds for HyperCheck-I and HyperCheck-II. The monitor machine is the same for both HyperCheck-I and HyperCheck-II. It is a Dell Precision 690 with 8 GB RAM and one 3.0 GHz Intel Xeon CPU with two cores. The host operating system is 64-bit CentOS 5.3. The target machine in HyperCheck-I is implemented

Table 5.3: Symbols for Xen hypervisor, Domain 0, Linux and Windows

System	Symbol	Usage
Xen	idt_table	Interrupt Descriptor Table
	hypercall_table	Hypercall Table
	exception_table	Exception Table
	._stext	Beginning of Hypervisor Code
	._etext	End of Hypervisor Code
Dom0	sys_call_table	Dom0's System Call Table
	._text	Beginning of Dom0's Kernel Code
	._etext	End of Dom0's Kernel Code
Linux	idt_table	Kernel's Interrupt Descriptor Table
	sys_call_table	Kernel's System Call Table
	._text	Beginning of Kernel Code
	._etext	End of Kernel Code
Windows	PCR→idt	Kernel's Interrupt Descriptor Table
	KiServiceTable	Kernel's System Call Table

on a Dell Optiplex GX 260 with one 2.0 GHz Intel Pentium 4 CPU and 512 MB memory. Xen 3.1 and Linux 2.6.18 is installed on the physical machine and the Domain 0 is CentOS 5.4. The Dell BIOS version A09 is closed source. The target machine in HyperCheck-II uses an ASUS M2V-MX_SE motherboard with 2.2 GHz AMD Sempron LE-1250 CPU and 2 GB memory. We install CentOS 5.5 as the operating system. We replace the original BIOS with the open source Coreboot V4.

Code Size

In the HyperCheck-I implementation, we inject about 100 lines of assembly code into the original BIOS. Since HyperCheck-II uses the open source Coreboot, we add only 200 lines of C code into Coreboot source tree. The code base of Coreboot V4 is 232,315 lines of code, and its payload Seabios has 21,576 lines of code, which are measured using SLOCCount [92].

Verifying the Static Property

We verify an important system assumption that the control data and respective code are statically mapped into the physical memory. We use a monitoring module designed to detect

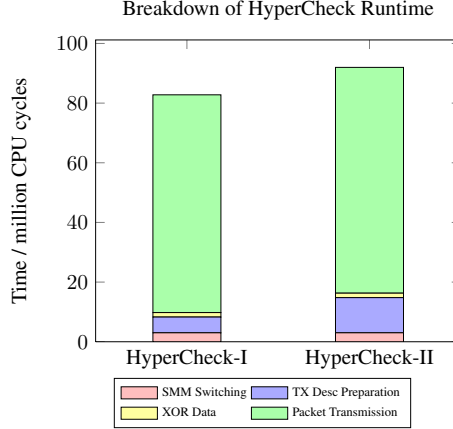


Figure 5.10: Breakdown of HyperCheck Runtime

legitimate control data and code modifications throughout the experiments. It enables us to test our approach against data changes and self-modifying code in the Xen hypervisor and Domain 0. We also test the static properties of Linux 2.6 and Windows XP 32-bit kernels. In all these tests, the hypervisor and the OSes are booted into a minimal state. The symbols used in the experiments are shown in Table 5.3. During system booting time, we find that the control data and the code may change. For example, the physical memory of IDT is all 0s when the system first boots up, but after several seconds, it becomes non-zero and static. The reason is that the IDT table is initialized later in the booting process.

Integrity Attack Detection

To verify HyperCheck’s capability of detecting attacks against the hypervisor, we implement DMA attacks [79] on the Xen hypervisor. Firstly, we port the HDD DMA attacks to modify the Xen hypervisor and Domain 0. In this experiment, there are four attacks against the Xen hypervisor (modifying IDT table, hypercall table, exception table, and Xen code) and two attacks against Domain 0 (modifying system call table and Domain 0 code). In another experiment, we modify the PCnet network card to perform the DMA attack from the hardware directly. The modified PCnet NIC is used to attack Linux and Windows

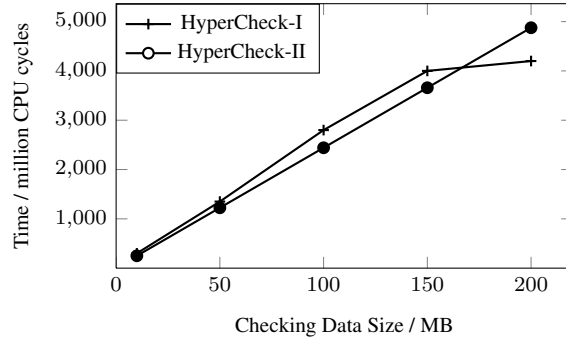


Figure 5.11: Network Time Delay for Variable Data Size in HyperCheck

operating systems. This experiment includes three attacks against Linux 2.6.18 kernel (modifying IDT table, system call table and kernel code) and two attacks to Windows XP SP2 kernel (modifying IDT table and system call table). In our experiments, HyperCheck-I and HyperCheck-II correctly detect all these attacks and report the memory content changes on the target machine.

Breakdown of HyperCheck

To quantify how much time is required to execute each step in the system, we breakdown the HyperCheck into four logical operations: 1) SMM context switch; 2) TX descriptors preparation; 3) XOR data; and 4) packet transmission. To measure the time for each operation, we use `rdtsc` instruction to print out the TSC counter value. This experiment is conducted on both HyperCheck-I and HyperCheck-II. The sending data size is about 2.8 MB including Xen code and Domain 0 code; we also add an extra 16 KB data at the end to flush the NIC internal buffer. In addition, we use 7 KB as the packet size because it introduces the lowest network delay; more details of network delay can be found in Section 5.2.5.

Figure 5.10 shows the observed times of each breakdown operation in HyperCheck-I and HyperCheck-II. We can see that the majority is packet transmission time. Additionally, HyperCheck-II spends more CPU cycles for preparing TX descriptors because the same amount of code running in SMM takes more time than in normal protected mode. This

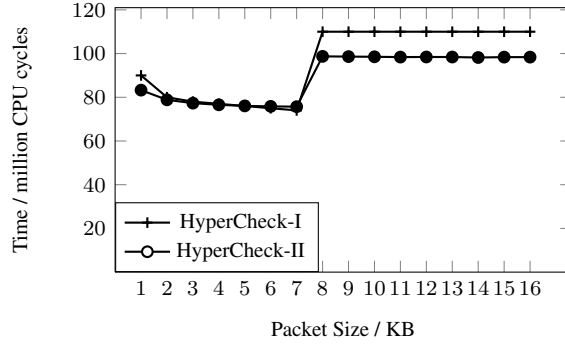


Figure 5.12: Network Time Delay for Variable Packet Size in HyperCheck

is mainly due to the fact that 1) SMM operates in 32-bit mode while normal OS runs in 64-bit protected mode, and 2) SMM physical memory needs to be uncachable to avoid cache poisoning attacks [58, 59].

The size of different hypervisors and OSes may vary (e.g., Linux running with KVM). HyperCheck is scalable to measure other hypervisors and OSes, but it should expect more performance overhead when measuring larger code base systems. We measure the time delay for sending different sizes of data in both HyperCheck-I and -II where the packet size is 7 KB. The results are shown in Figure 5.11. We can see that the time increases almost linearly along with the size of memory in both prototypes.

Network Packet Size Analysis

To optimize the network time delay for our system, we measure the packet transmission time by varying the packet size for sending a fixed amount of memory. The memory size is about 2.8 MB including Xen code and Domain 0 code. We range the packet size from 1 KB to 16 KB on both HyperCheck-I and HyperCheck-II. Figure 5.12 shows the results. When the packet size is less than 7 KB, the transmission time is about constant. However, when the packet size increases to 8 KB, the overhead increases dramatically and remains constant after that. The reason is that the internal NIC transfer FIFO buffer size is 16 KB

Table 5.4: Time Measurements for Variable Packet Sizes in HyperCheck-II

Packet size (KB)	3	4	5	6	7	8	9	10	11
Sending time on target (million CPU cycles)	77	77	76	76	76	99	99	99	98
Receiving time on monitor (million CPU cycles)	87	87	86	86	86	114	114	114	114
Processing time on monitor (million CPU cycles)	21	22	21	21	21	20	21	21	21

for our network card. Therefore, when the packet size becomes 8 KB or larger, the buffer cannot hold two packets at the same time, and this introduces the delay.

Table 5.4 shows the time measurements on both the target machine and the monitor machine for variable packet size ranging from 3 KB to 11 KB in HyperCheck-II. The total amount of data transferred is 2,897 KB, including Xen code, Domain 0 code, and 16 KB for flushing the internal NIC buffer. The sending time is measured on the target machine in HyperCheck-II; the receiving time and processing time are measured on the monitor machine. The receiving time represents the time period between the first packet arrives and the time when the last one arrives, and it is measured by *tcpdump*. To process the data, we use a customized program on the monitor machine to compare the Xen code and Domain 0 code byte by byte, and it takes about 21 million CPU cycles. To optimize the time delay on the monitor machine, we can process packets while receiving packets. In this case, the total time delay on the monitor machine will be bounded by the receiving time because receiving packets takes more time than processing packets.

System Overhead

We also measure the overall system overhead incurred by different sampling intervals of HyperCheck-II. In this experiment, we run the UnixBench [93] suite without our system in place. Next, we run the benchmark with HyperCheck-II enabled at several different time intervals ranging from 0.0625 to 5 seconds using Global Standby Timer (GP0) on the southbridge to periodically trigger an SMI. We then calculate the overhead as a ratio with and without the system in place. In this experiment, we transfer Xen and domain 0 code (2,881 KB) and use 7 KB as the packet size. Figure 5.13 shows the result of overhead. In

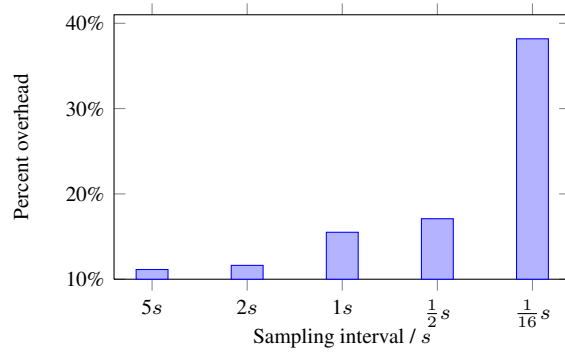


Figure 5.13: Overhead Introduced in HyperCheck-II with Different Sampling Intervals

Table 5.5: Comparison on Time Overhead

Code base (Size:MB)	HyperCheck	HyperGuard	Flicker
Linux (2.0)	31 ms	203 ms	1022 ms
Xen+Dom0 (2.7)	40 ms	274 ms	>1022 ms
Window XP (1.8)	28 ms	183 ms	>972 ms
Hyper-V (2.4)	36 ms	244 ms	>1022 ms
VMWare ESXi (2.2)	33 ms	223 ms	>1022 ms

general, HyperCheck-II introduces a low overhead. It causes 2% overhead when triggering SMI every 5 seconds and 11% overhead with a 1 second sampling interval.

Comparison with Other Methods

HyperGuard [57] suggests using SMM to read the memory and hash it on the target machine. Flicker [19] is a TPM-based approach that can be used to monitor the integrity of the kernels. We compare our method with them, and Table 5.5 shows the results. We can see that the overhead of HyperCheck is one order of magnitude lower than HyperGuard and TPM-based method. In HyperGuard, it hashes the entire data to check its integrity, while HyperCheck only hashes a random portion of the data and then sends the entire data out using an Ethernet card. For the TPM-based method, the most expensive operation is the TPM quote, which takes 972 ms. Although HyperCheck needs TPM in SRTM process to

Table 5.6: Comparison on Capability and Overhead

	Memory	Registers	Overhead
HyperCheck	x	x	Low
HyperGuard	x	x	High
Copilot	x		Low
Flicker	x	x	High

secure the BIOS, it does not require TPM at the runtime. Once the SMM is securely setup, HyperCheck leverages SMM to perform its integrity checking, while Flicker requires TPM operation for each check. Additionally, an overall comparison between HyperCheck and other methods is shown in Table 5.6. In summary, HyperCheck can monitor both memory and registers with a lower overhead.

5.3 IOCheck: Firmware-level Malware Detector

5.3.1 Introduction

As hardware devices have become more complex, firmware functionality has expanded, exposing new vulnerabilities to attackers. The National Vulnerabilities Database (NVD [5]) shows that 183 firmware vulnerabilities have been found since 2011. The Common Vulnerabilities and Exposures (CVE) list from Mitre shows 537 entries that match the keyword ‘firmware,’ and 94 new firmware vulnerabilities were found in 2013 [94]. A recent study shows that 40,000 servers are remotely exploitable due to vulnerable management firmware [95]. Attackers can exploit these vulnerabilities in firmware [9] or tools for updating firmware [10].

After compromising the firmware of an I/O device (e.g., NIC card), attackers alter memory via DMA [9, 96, 97] or compromise surrounding I/O devices [98, 99]. Fortunately, the Input Output Memory Management Unit (IOMMU) mechanism can protect the host memory from DMA attacks. It maps each I/O device to a specific area in the host memory so that any invalid access fails. Intel Virtualization Technology for Directed I/O (VT-d) is

one example of IOMMU. AMD also has its own I/O virtualization technology called AMD-Vi. However, IOMMU cannot always be trusted as a countermeasure against DMA attacks, as it relies on a flawless configuration to operate correctly [100]. In particular, researchers have demonstrated several attacks against IOMMU [76, 101, 102].

Static Root of Trust for Measurement (SRTM) [103] with help from the Trust Platform Module (TPM) [67] can check the integrity of the firmware and I/O configurations while booting. It uses a fixed or immutable piece of trusted code, called the Core Root of Trust for Measurement (CRTM), contained in the BIOS at the start of the entire booting chain, and every piece of code in the chain is measured by the predecessor code before it is executed, including firmware. However, SRTM only secures the booting process and cannot provide runtime integrity checking.

Trust Computing Group introduced Dynamic Root of Trust for Measurement (DRTM) [104]. To implement this technology, Intel developed Trusted eXecution Technology (TXT) [105], providing a trusted way to load and execute system software (e.g., OS or VMM). TXT uses a new CPU instruction, **SENTER**, to control the secure environment. Intel TXT does not make any assumptions about the system state, and it provides a dynamic root of trust for Late Launch. Thus, TXT can be used to check the runtime integrity of I/O configurations and firmware. AMD has a similar technology called Secure Virtual Machine, and it uses the **SKINIT** instruction to enter the secure environment. However, both TXT and SVM introduce a significant overhead on the late Launch Operation (e.g., the **SKINIT** instruction in [19]).

In this thesis, I present IOCheck [106], a framework to enhance the security of I/O devices at runtime. It leverages System Management Mode (SMM), a CPU mode in the x86 architecture, to quickly check the integrity of I/O configurations and firmware. IOCheck identifies the target I/O devices on the motherboard and checks the integrity of their corresponding configurations and firmware. In contrast to existing firmware integrity checking systems [107, 108], our approach is based on SMM instead of Protected Mode (PM). While PM-based approaches assume the booting process is secure and the OS is trusted, our

approach only assumes a secure BIOS boot to set up SMM, which is easily achieved via SRTM.

The superiority of SMM over PM is two-fold. First, we can reduce the Trusted Computing Base (TCB) of the analysis platform. Similar to Viper [108] and NAVIS [107], IOCheck is a runtime integrity checking system. Viper and NAVIS assume the OS is trusted and use software in PM to check the integrity, while IOCheck uses SMM without relying on the OS, resulting in a much smaller TCB. IOCheck is also immune to attacks against the OS, facilitating a stronger threat model than the checking systems running in the OS. Second, we achieve a much higher performance compared to the DRTM approaches [19] running in PM. DRTM does not rely on any system code; it can provide a dynamic root of trust for integrity checking. IOCheck can achieve the same security goal because SMM is a trusted and isolated execution environment. However, IOCheck is able to achieve a much higher performance over Intel TXT or AMD SVM approaches. Based upon experimental results, SMM switching time takes microseconds, while the switching operation of the DRTM approach [19] takes milliseconds.

We implement a prototype of our system using different methods to enter SMM. First, we develop a random polling-based integrity checking system that checks the integrity of I/O devices, which can mitigate transient attacks [91, 109]. To further defend against transient attacks, we also implement an event-driven system that checks the integrity of a network card’s management firmware.

We conduct extensive experiments to evaluate IOCheck on both Microsoft Windows and Linux systems. The experimental results show that the SMM code takes about 10 milliseconds to check PCI configuration space and firmware of NIC and VGA. Through testing IOCheck with popular benchmarks, IOCheck incurs about a 2% overhead when we set the random polling instruction interval between $[1, 0xffffffff]^1$. We also compare IOCheck with the DRTM approach; our results indicate that our system’s switching time is three orders of magnitude faster than DRTM. Furthermore, the switching time of IOCheck is

¹It takes about .5s to run 0xffffffff instructions. Table 5.11 explains this further.

constant while the switching operation in DRTM depends on the size of the loaded secure code.

5.3.2 Threat Model and Assumptions

Threat Model

We consider two attack scenarios. First, we consider an attacker who gains control of a host through a software vulnerability and then attempts to remain resident in a stealthy manner. We assume such an attacker installs firmware rootkits (specifically, a backdoor [110]) after infecting the OS so that the malicious code remains even if the user reinstalls the OS.

In the second scenario, we assume the firmware itself can be remotely exploited due to vulnerabilities. For instance, Duflot et al. [9] demonstrate an attack that remotely compromises a Broadcom NIC with crafted UDP packets. Additionally, Bonkoski et al. [95] show a buffer overflow vulnerability in management firmware that affected thousands of servers.

Assumptions

An attacker is able to tamper with the firmware by exploiting zero-day vulnerabilities. Since IOCheck does not rely on the operating system, we assume the attacker has ring 0 privilege. Thus, attackers are granted more capabilities in our work than those OS-based systems [107, 108]. We assume the system is equipped with SRTM, in which CRTM is trusted so that it can perform a self-measurement of the BIOS. Once the SMM code is securely loaded into the SMRAM, we lock the SMRAM in the BIOS. We assume the SMM is secure after locking SMRAM. Moreover, we assume the attacker does not have physical access to our system.

5.3.3 System Architecture

IOCheck is a framework that enhances the security of I/O devices at runtime, and it checks the static configurations and code of I/O devices. Figure 5.14 shows the architecture of

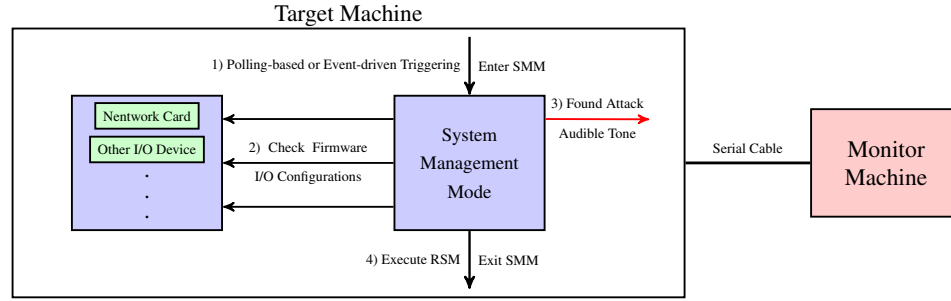


Figure 5.14: Architecture of IOCheck

IOCheck. The target machine on the left connects to the remote machine via a serial cable. In the target machine, the box on the left lists all of the I/O devices on a motherboard; the box on the right represents the System Management Mode that checks the integrity of I/O configurations and firmware. The framework takes the following four steps for each check: 1) the target machine switches into SMM using a polling-based or event-driven triggering approach; 2) the SMI handler checks the integrity of target I/O devices; 3) if a potential attack is found, the system plays an audible tone on the target machine, and SMM sends a message to the remote machine via the serial cable; and 4) the target machine executes RSM instruction to exit SMM. Next, I detail the design of these four steps.

Triggering an SMI

Triggering an SMI is the only way to enter SMM [63]. In general, there are software- and hardware-based methods to trigger an SMI. In software, I can write to an ACPI port to raise an SMI. For example, Intel chipsets use port `0x2b` as specified by the southbridge datasheet. My test bed with VIA VT8237r as the southbridge uses `0x52f` as the SMI trigger port [68]. In terms of hardware-based methods, there are many hardware devices that can be used to raise an SMI, including keyboards, network cards, and hardware timers.

The algorithm for triggering SMIs plays an important role in the system design. In general, there are polling-based and event-driven approaches for generating SMIs. The

polling-based approach aims to poll and check the state of a target system at regular intervals. When I use this approach to check the integrity of a target system, it compares the newly retrieved state with the pristine state to see if any malicious change has happened. For instance, I can periodically trigger an SMI and check the firmware and I/O configurations on the target machine. However, polling at regular intervals in the system is susceptible to transient [109] and evasion attacks [91].

Transient attacks are a class of attacks that do not produce persistent changes within a victim's system. Periodic polling-based integrity checking systems suffer from this kind of attack. This is because they require the presence of inconsistent system states to infer an intrusion. The transient attack can avoid detection by removing all evidence of an attack before the integrity checks begin, while resuming the malicious code after the checks finish. There are two approaches to mitigating the transient attack in polling-based systems. One is to minimize the time window of the polling to reduce the likelihood that an attacker can finish cleaning its traces. Alternatively, I can randomize the polling so that the attacker is less likely to learn a particular polling pattern. These two methods can be achieved by using performance counters to generate an SMI.

Moreover, I can use an event-driven triggering method to further mitigate transient attacks. The polling-based systems are likely to miss events between two checks, while the event-driven triggering approach cannot. For instance, supposing that my system monitors a static memory region to see if any malicious modification happens, the event-driven approach can trigger an SMI for every memory change. Thus, I am able to monitor all of the memory updates, including malicious ones.

Checking I/O Configurations and Firmware

After the target machine switches into SMM, I will check the I/O configuration and firmware. Next, I describe the I/O devices that my system can check.

Before the system boots up, the BIOS initializes all of the hardware devices on the motherboard and sets corresponding configurations to them. These devices rely on the

configurations to operate correctly. Here I use the PCI configuration space and IOMMU configuration as examples.

PCI Configuration Space: Each PCI or PCI Express controller has a configuration space. Device drivers read these configurations to determine what resources (e.g., memory mapped location) have been assigned by the BIOS to the devices. Note that the PCI configurations should be static after the BIOS initialization. However, an attacker with ring 0 privilege can modify the PCI configuration space. For example, the attacker can relocate the device memory by changing the Base Address Register in the PCI configuration space. Additionally, PCI/PCIe devices that support Message Signaled Interrupts (MSI) contain registers in the PCI configuration space to configure MSI signalling. Wojtczuk and Rutkowska demonstrate that the attacker in the driver domain of a VM can generate malicious MSIs to compromise a Xen hypervisor [76]. Note that IOCheck assumes that the PCI configuration remains the same after the BIOS initialization and does not consider “Plug-and-Play” PCI/PCIe devices.

IOMMU Configurations: IOMMU restricts memory access of I/O devices. For example, it can prevent a Direct Memory Access (DMA) attack from a compromised I/O device. IOMMU is composed of a set of DMA Remapping Hardware Units (DRHU). They are responsible for translating addresses from I/O devices to physical addresses in the host memory. The DRHU first identifies a DMA request by BDF-ID (Bus, Device, Function number). Then, it uses BDF-ID to locate the page tables associated with the requested I/O controller. Finally, it translates the DMA Virtual Address (DVA) to a Host Physical Address (HPA), which is similar to the MMU translation.

Although IOMMU gives us effective protection from DMA attacks, it relies on correct configurations to operate appropriately. Several ways have been demonstrated to bypass IOMMU [76, 101]. I can mitigate these attacks by checking the integrity of the critical configurations of IOMMU at runtime.

For example, the DMA Remapping (DMAR) Advanced Configuration and Power Interface (ACPI) table should never change after booting. The DMAR ACPI table describes

Table 5.7: IOMMU Configurations

Register/Table Name	Description
Root-entry table address register	Define the base address of the root-entry table (first-level table identified by bus number)
Domain mapping tables	Include root-entry table and context-entry tables (second-level tables identified by device and function numbers)
Page tables	Define memory regions and access permissions of I/O controllers (third-level tables)
DMA remapping ACPI table	Define the number of DMA remapping hardware units and I/O controllers assigned to each of them

the number of DRHUs present in the system and I/O controllers associated with each of them. It is set by the BIOS before the system boots up. In addition, the base address of the configuration tables for DMA remapping unit should be static. I can check the integrity of these static configurations to ensure that IOMMU operates correctly. Table 5.7 shows the static configuration of IOMMU.

IOCheck will check the firmware of I/O devices including the network card, graphics card, keyboard, and mouse. Next, I will use a NIC and the BIOS as examples.

Network Interface Controller: Modern network cards continue to become more and more complex. NICs usually include a separate on-chip processor and memory to support various functions. Typically, a NIC loads its firmware from Electric Erasable Programmable Read-Only Memory (EEPROM) to flash memory and then executes the code on the on-chip processor. To check the integrity of NIC's firmware at runtime, IOCheck stores a hash value of the original firmware image in SMRAM while the system executes the BIOS code. After the operating system boots up, IOCheck reads the NIC's firmware code from the flash memory and calculates the hash value of the current image. If the computed hash value does not match the stored value, an attack against NIC may have occurred. For some network cards [111], I can monitor the Program Counter of the on-chip CPU through the NIC's debugging registers, which can restrict the instruction pointer to the code section of the memory region. For instance, if the instruction pointer points to a memory region that stores heap or stack data, then a code injection and control flow hijacking may have

happened.

Monitoring the integrity of the static code and instruction pointer can prevent an attacker from injecting malicious code into firmware; however, it cannot detect advanced attacks, such as Return Oriented Programming attacks, which technically do not inject any code. To detect these attacks, I can implement a shadow stack to protect the control flow integrity of the NIC firmware. Duflot et al. implemented a similar work in NAVIS [107]. I will study the control flow integrity of the firmware as my future work.

Basic Input/Output System: As mentioned before, SRTM can check the integrity of the BIOS at booting time, which helps us securely load the SMM code from the BIOS to the SMRAM. After the system boots up, attackers with ring 0 privilege might modify the BIOS using various tools (e.g., flashrom [112]). However, they are not able to access locked SMRAM. Thus, I can use the SMM code to check the runtime integrity of the BIOS. The checking method is similar to other firmware verification techniques. IOCheck stores a hash value of initial code and checks if any alterations occur while the system is running.

Although the modified BIOS with malicious code cannot be executed until the system resets and SRTM detects this BIOS attack before booting, I can detect this attack earlier than SRTM, which provides runtime detection and serves as a complementary defense. Earlier detection of such attacks can also limit the damage they wreak against the system. Note that I assume CRTM in the BIOS is immutable and trusted, but attackers can modify any other BIOS code (e.g., ACPI tables). Otherwise, I cannot perform SRTM correctly.

Reporting an Alert and Exiting SMM

The last stage of IOCheck is to report any alerts to a human operator. I will accomplish this task by playing an audible tone to notify a user that a potential attack may happen. To distinguish the type of attack (e.g., firmware attack vs. configuration attack), I will use different tone frequencies for a variety of I/O attacks. In addition, I will use a serial cable connecting the target machine to the remote machine. If a potential attack has been found, SMM prints the attacking information on the remote machine through the serial port.

Note that the reporting stage happens in SMM. Even if an attack disables the PC speaker or serial console in PM, I can enable it in SMM and guarantee an audible tone and a serial message to be delivered when an attack is detected. After the reporting stage, the SMI handler simply executes the `RSM`² instruction to exit from SMM.

5.3.4 Implementation

We implement a prototype of IOCheck system using two physical machines. The target machine uses an ASUS M2V-MX_SE motherboard with an AMD K8 Northbridge and a VIA VT8237r Southbridge. It has a 2.2 GHz AMD LE-1250 CPU and 2 GB Kingston DDR2 RAM. We use a PCIe-based Intel 82574L Gigabit Ethernet Controller and a PCI-based Jaton VIDEO-498PCI-DLP Nvidia GeForce 9500GT as the testing devices. To program SMM, we use open-source BIOS, Coreboot. Since IOCheck is OS-agnostic, we install Microsoft Windows 7 and CentOS 5.5 on the target machine. The external machine is a Dell Inspiron 15R laptop with Ubuntu 12.04 LTS. It uses a 2.4 GHz Intel Core i5-2430M CPU and 6 GB DDR3 RAM.

Triggering an SMI

We implement a random polling-based triggering algorithm to check integrity of I/O configurations and firmware by using performance counters to generate SMIs. The performance monitoring registers count hardware events such as instruction retirement, L1 cache miss, or branch misprediction. The x86 machines provide four of these counters from which we can select a specific hardware event to count [113]. To generate an SMI, we first configure one of the performance counters to store its maximum value. Next, we select a desired event (e.g., a retired instruction or cache miss) to count so that the next occurrence of that event will overflow the counter. Finally, we configure the Local Advanced Programmable Interrupt Controller (APIC) to deliver an SMI when an overflow occurs. Thus, we are able to trigger an SMI for the desired event. The performance counting event is configured by

²The `RSM` instruction can only be used while in SMM [63].

the `PerfEvtSel` register, and the performance counter is set by the `PerfCtr` register [113].

To randomly generate SMIs, we first generate a pseudo-random number, r , ranging from 1 to m , where m is a user-configurable maximum value. For example, a user could set m as `0xffff` ($2^{16} - 1$), so the random number resides in the set `[1,0xffff]`. Next, we set the performance counter to its maximum value (`0xffffffff`) minus this random number ($2^{48} - 1 - r$). We also set the desired event in `PerfEvtSel` and start to count the event. Thus, an SMI will be raised after r occurrences of the desired event. We use a linear-congruential algorithm to generate the pseudo-random number, r , in SMM. We use the parameters of the linear-congruential algorithm from Numerical Recipes [114]. We use the TSC value as the initial seed and save the current random number in SMRAM as the next round's seed.

To further mitigate transient attacks, we consider event-driven-based triggering approaches. We implement an event-driven-based version of `IOCheck` for checking the integrity of a NIC's management firmware, and the detailed implementation is described as follows. When a management packet arrives at the PHY interface of the NIC, the manageability firmware starts to execute. We use Message Signalled Interrupts (MSI) to trigger an SMI when a manageability packet arrives at the network card. First, we configure the network card to deliver an MSI to the I/O APIC with the delivery mode specified as SMI. When the I/O APIC receives this interrupt, it automatically asserts the SMI pin, and an SMI is generated. Next, we use the SMM code to check the integrity of the management firmware. Note that the act of this triggering is generated via a hardware interrupt in the NIC, and the management firmware code is decoupled from this. Thus, we trigger an SMI for every manageability packet before the firmware has an opportunity to process it.

Checking I/O Configurations and Firmware

We use a popular commercial network card, an Intel 82574L Gigabit PCIe Ethernet Controller, as our target I/O device. First, we check the PCIe configuration space of the network

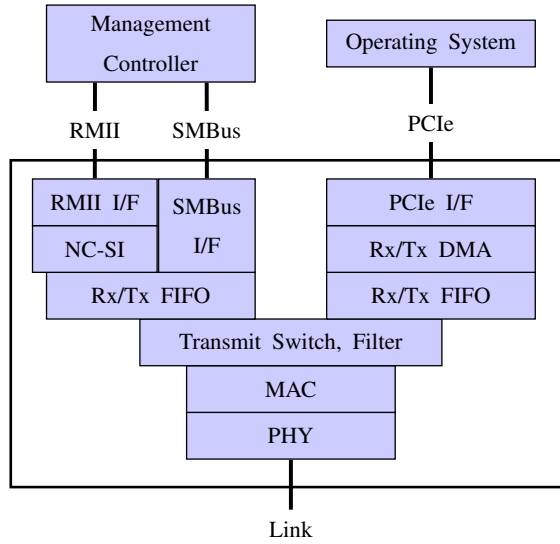


Figure 5.15: Architecture Block Diagram of Intel 82574L

card. The NIC on our testbed is at bus 3, device 0, and function 0. To read the configuration space, we use standard PCI reads to dump the contents. We use a standard hash function MD5 [115] to hash these 256 bytes of the configuration and compare the hash value with the original one generated during booting.

Network management is an increasingly important requirement in today’s networked computer environments, especially on servers. It routes manageability network traffic to a Management Controller (MC). One example of MC is the Baseboard Management Controller (BMC) in Intelligent Platform Management Interface (IPMI). The management firmware inevitably contains vulnerabilities that could be easily exploited by attackers. Bonkoski et al. [95] identified more than 400 thousand IPMI-enabled servers running on publicly accessible IP addresses that are remotely exploitable due to textbook vulnerabilities in the management firmware. The 82574L NIC [116] provides two different and mutually exclusive bus interfaces for manageability traffic. One is the Intel proprietary System Management Bus (SMBus) interface, and the other is the Network Controller - Sideband Interface (NC-SI). Each manageability interface, it has its own firmware code

Table 5.8: PCI Expansion ROM Header Format for x86

Offset	Length	Value	Description
0h	1	55h	ROM signature, byte 1
1h	1	AAH	ROM signature, byte 2
2h	1	xx	Initialization size
3h	3	xx	Entry point for INIT function
6h-17h	12h	xx	Reserved
18h-19h	2	xx	Pointer to PCI data structure

that implements the functions. Figure 5.15 shows a high-level architectural block diagram of the 82574L NIC [116].

The management firmware of these two interfaces is stored in a Non-Volatile Memory (NVM). The NVM is I/O mapped memory in the NIC, and we use the EEPROM Read Register (EERD 0x14) to read it. EERD is a 32-bit register used to cause the NIC to read individual words in the EEPROM. To read a word, we write a 1b to the Start Read field. The NIC reads the word from the EEPROM and places it in the Read Data field and then sets the Read Done field to 1b. We poll the Read Done bit to make sure that the data has been stored in the Read Data field. All of the configuration and status registers of 82574L NIC, including EERD, are memory-mapped when the system boots up. To access EERD, we use normal memory read-and-write operations. The memory address of EERD is INTEL_82574L_BASE plus EERD offset.

Jaton VIDEO-498PCI-DLP GeForce 9500GT is a PCI-based video card. It is at bus 7, device 0, and function 0 on our testbed. Similar to the checking approach of NIC, we first check the PCI configuration space of the VGA device. Then, we check the integrity of the VGA expansion ROM. The VGA expansion ROM is memory-mapped, and the four-byte register at offset 0x30 in the PCI configuration space specifies the base address of the expansion ROM. Note that bit 0 in the register enables the accesses to the expansion ROM. PCI expansion ROMs may contain multiple images for different architectures. Each image must contain a ROM header and PCI data structure, which specify image information such as code type and size. Table 5.8 shows the formats of ROM header and Table 5.9 shows

Table 5.9: PCI Data Structure Format

Offset	Length	Description
0h	4	Signature, the string "PCIR"
4h	2	Vendor identification
6h	2	Device identification
8h	2	Reserved
Ah	2	PCI data structure length
Ch	1	PCI data structure revision
Dh	3	Class code
10h	2	Image length
12h	2	Revision level of code/data
14h	1	Code type
15h	1	Indicator
16	2	Reserved

the PCI data structure. Note that we only check the image for x86 architecture since our testbed is on Intel x86.

We first use the base address of expansion ROM to locate the header of the first image. Next, we read the pointer to PCI data structure at offset 0x18 to 0x19. Then, we identify the code type at offset 0x14 in the PCI data structure. If this image is for Intel x86 architecture, we check the integrity of this image by comparing the hash values. Otherwise, we repeat the steps above for the next image.

Reporting an Alert and Exiting SMM

To play a tone, we program the Intel 8253 Programmable Interval Timer (PIT) in the SMI handler to generate tones. The 8253 PIT performs timing and counting functions, and it exists in all x86 machines. In modern machines, it is included as part of the motherboard's Southbridge. This timer has three counters (Counters 0, 1, and 2), and we use the third counter (Counter 2) to generate tones via the PC speaker. In addition, we can generate different kinds of tones by adjusting the output frequency. In the prototype of IOCheck, a continuous tone would be played by the PC speaker if an attack against NIC has been found. If an attack against VGA has been found, an intermittent tone would be played.

We use a serial cable to print status messages and debug corresponding I/O devices in

SMM. The `printk` function in Coreboot prints the status messages to the serial port on the target machine. When the target machine executes the BIOS code during booting, the external machine sends a 16-byte random number to the target machine through the serial cable. Then, the BIOS stores the random number as a secret in the SMRAM. Later, the status messages are sent with the secret for authentication. We run a `minicom` instance on the external machine and verify if the secret is correct. If a status message is not received in an expected time window or the secret is wrong, we conclude that an attack has occurred.

5.3.5 Evaluation

Code Size

In total, there are 310 lines of new C code in the SMI handler. The MD5 hash function has 140 lines of C code [115], and the rest of the code implements the firmware and PCI configuration space checking. After compiling the Coreboot, the binary size of the SMI handler is only 1,409 bytes, which introduces a minimal TCB to our system. The 1,409-byte code encompasses all functions and instructions required to check the integrity of the NIC and VGA firmware and their PCI configuration spaces. The code size will increase if we check more I/O devices. Additionally, other static code exists in Coreboot related to enabling SMM to run on a particular chipset. For example, a `printk` function is built into the SMM code to enable raw communication over a serial port.

Attack Detection

We conduct four attacks against our system on both Windows and Linux platforms. Two of them are I/O configuration attacks, which relocate the device memory by manipulating the PCI configuration space of NIC and VGA. The other two attacks modify the management firmware of the NIC and VGA option ROM. The Base Address Registers (BARs) in the PCI configuration space are used to map the device’s register space. They reside from offset 0x10 to 0x27 in the PCI configuration space. For example, the memory location BAR0 specifies the base address of the internal NIC registers. An attacker can relocate

Table 5.10: Breakdown of SMI Handler Runtime (Time: μs)

Operations	Mean	STD	95% CI
SMM switching	3.92	0.08	[3.27,3.32]
Check NIC's PCIe configuration	1169.39	2.01	[1168.81,1169.98]
Check NIC's firmware	1268.12	5.12	[1266.63,1269.60]
Check VGA's PCI configuration	1243.60	2.61	[1242.51,1244.66]
Check VGA's expansion ROM	4609.30	1.30	[4608.92,4609.68]
Send a message	2082.95	3.00	[2082.08,2083.82]
Configure the next SMI	1.22	0.06	[1.20,1.24]
SMM resume	4.58	0.10	[4.55,4.61]
Total	10,383.07		

these memory-mapped registers for malicious purposes by manipulating the BAR0 register. To conduct the experiments, we first enable IOCheck to check the PCI configuration space. Next, we modify the memory location specified by the BAR0 register on Windows and Linux platforms. We write a kernel module to modify the BAR0 register in Linux and use the RWEverything [117] tool to configure it in Windows. We also modify the management firmware of NIC and the VGA option ROM. The management firmware is stored as a Non-Volatile memory, and it is I/O mapped memory; the VGA option ROM is memory-mapped. These attacks are also conducted on both Windows and Linux platforms.

After we modify NIC's PCIe configuration or the firmware, IOCheck automatically plays a continuous tone to alert users and the `minicom` instance on the external machine shows an attack against NIC has been found. After the modification of VGA's PCI configuration or option ROM, an intermittent tone is played by the PC speaker.

Breakdown of SMI Handler Runtime

To quantify how much time each individual step is required to run, we break down the SMI handler into eight operations. They are 1) switch into SMM; 2) check the PCIe configuration of NIC; 3) check the firmware of NIC; 4) check the PCI configuration of VGA; 5) check the option ROM of VGA; 6) send a status message; 7) configure the next SMI; and 8) resume Protected Mode. For each operation, we measure the average time taken in SMM. We use

Table 5.11: Random Polling Overhead Introduced on Microsoft Windows and Linux

Random Polling Intervals			Benchmark Runtime(s)		System Slowdown	
	Instructions	Time (μs)	Windows	Linux	Windows	Linux
1	[1,0xffffffff]	(0,~650,752]	0.285	0.393	0.014	0.011
2	[1,0xffffffff]	(0,~40,672]	0.297	0.398	0.057	0.023
3	[1,0xffffffff]	(0,~2,542]	0.609	0.463	1.167	0.190
4	[1,0xffffffff]	(0,~158]	4.359	1.480	14.512	2.805
5	[1,0xffff]	(0,~10]	91.984	18.382	~326	~46

the Time Stamp Counter (TSC) register to calculate the time. The TSC register stores the number of CPU cycles elapsed since powering on. First, we record the TSC values at the beginning and end of each operation, respectively. Next, we use the CPU frequency to divide the difference in the TSC register to calculate how much time this operation.

We repeat this experiment 40 times. Table 6.5 shows the average time taken for each operation. We can see that the SMM switching and resuming take only 4 and 5 microseconds, respectively. Checking 256 bytes of the PCIe/PCI configuration space register takes about 1 ms. The 82574L NIC has 70 bytes of SMBus Advanced Pass Through (APT) management firmware and 138 bytes of NC-SI management firmware. The size of the x86 expansion ROM image is 1 KB in the testing VGA. Checking NIC’s firmware takes about 1 ms, while checking VGA’s option ROM takes about 5 ms. Naturally, the size of the firmware affects the time of the checking operation. We send a status message (e.g., I/O devices are OK) in each run of the SMI handler, which is about 2 ms. The time it takes to generate a random number and configure performance counters for the next SMI is only 1.22 ms. Thus, the total time spent in SMM is about 10 milliseconds. Additionally, we calculate the standard deviation and 95% confidence interval for the runtime of each operation.

System Overhead

To measure system overhead introduced by this approach, we use the SuperPI [118] program to benchmark our system on Windows and Linux. We first run the benchmark without IOCheck enabled. Then, we run it with different random-polling intervals. Table 5.11 shows

the experimental results. The first column shows the random polling intervals used in the experiment. For example, $(0, 0xffff]$ means a random number, r , is generated in that interval. We use retired instructions as the counting event in the performance counter. Thus, after running r sequential instructions, an SMI will be asserted. The second column also indicates the time elapsed. Since the CPU (AMD K8) on our testbed is 3-way super-scalar [119], we assume that the average number of instructions-per-cycle (IPC) is 3, and the equation for this transformation is $T = \frac{I}{(C * IPC)}$, where T is the real time, I is the number of instructions, and C is the clock speed on the CPU.

We can see from Table 5.11 that the overhead will increase if we reduce the random-polling interval, while small intervals have a higher probability of quickly detecting attacks. Intervals in rows 1 and 2 introduce less than 6% overhead, so intervals similar to or between them are suitable for normal users in practice. Other intervals in the table have large overhead, making them unsuitable in practice. These results demonstrate the feasibility and scalability of our approach.

Comparison with the DRTM Approach

IOCheck provides a new framework for checking firmware and I/O devices at runtime. Compared to the well-known DRTM approach (e.g., Flicker [19]), SMM in IOCheck serves a similar role as the trusted execution environment in DRTM. However, IOCheck achieves a better performance in comparison. AMD uses the `SKINIT` instruction to perform DRTM, and Intel implements DRTM using a CPU instruction called `SENTER`. The SMM switching operation in IOCheck plays the same role as `SKINIT` or `SENTER` instructions in the DRTM approach. As stated in the Table II of Flicker [19], the time required to execute the `SKINIT` instruction depends on the size of the Secure Loader Block (SLB). It shows a linear growth in runtime as the size of the SLB increases. From Table 5.12, we can see that the `SKINIT` instruction takes about 12 milliseconds for 4 KB of SLB. However, SMM switching only takes about 4 ms, which is about three orders of magnitude faster than the `SKINIT` instruction. Furthermore, SMM switching time is independent from the size of the SMI handler. This

Table 5.12: Comparison between SMM-based and DRTM-based Approaches

	IOCheck	Flicker [19]
Operation	SMM switching	SKINIT instruction
Size of secure code	Any	4 KB
Time	3.92 μs	12 ms
Trust BIOS boot	Yes	No

is because IOCheck does not need to measure the secure code every time before executing it, and we lock the secure code in SMRAM.

Note that IOCheck trusts the BIOS boot while Flicker does not. IOCheck requires a secure BIOS boot to ensure the SMM code is securely loaded into SMRAM. However, the DRTM approach (e.g., Intel TXT) also requires that the SMM code is trusted. Wojtczuk and Rutkowska demonstrate several attacks [102,120,121] against Intel TXT by using SMM if the SMM-Transfer Monitor is not present. From this point of view, both systems must trust the SMM code.

Chapter 6: Using Hardware Isolated Execution Environments for Malware Debugging

6.1 MalT: Towards Transparent Debugging

6.1.1 Introduction

Traditional malware analysis employs virtualization [37, 41, 122] and emulation [39, 40, 45] technologies to dissect malware behavior at runtime. This approach runs the malware in a Virtual Machine (VM) or emulator and uses an analysis program to introspect the malware from the outside so that the malware cannot infect the analysis program. Unfortunately, malware writers can easily escape this analysis mechanism by using a variety of anti-debugging, anti-virtualization, and anti-emulation techniques [11, 13–16, 123]. Malware can easily detect the presence of a VM or emulator and alter its behavior to hide itself. Chen et al. [11] executed 6,900 malware samples and found that more than 40% of them reduced malicious behavior under a VM or with a debugger attached. Branco et al. [123] showed that 88% and 81% of 4 million analyzed malware samples had anti-reverse engineering and anti-virtualization techniques, respectively. Furthermore, Garfinkel et al. [12] concluded that virtualization transparency is fundamentally infeasible and impractical. To address this problem, security researchers have proposed analyzing malware on bare metal [42, 43]. This approach makes anti-VM malware expose its malicious behavior, and it does not require any virtualization or emulation technology. However, malware analysis on bare metal runs an analysis program within the Operating System (OS), and ring 0 malware can easily detect its presence. Thus, stealthy malware detection and analysis still remains an open research problem.

I present MalT [124], a novel approach that progresses toward stealthy debugging by

leveraging System Management Mode (SMM) to transparently debug software on bare-metal. Our system is motivated by the intuition that malware debugging needs to be transparent, and it should not leave artifacts introduced by the debugging functions. SMM is a special-purpose CPU mode in all x86 platforms. The main benefit of SMM is to provide a distinct and easily isolated processor environment that is transparent to the OS or running applications. With the help of SMM, we are able to achieve a high level of transparency, which enables a strong threat model for malware debugging. We briefly describe its basic workflow as follows. We run malware on one physical target machine and employ SMM to communicate with the debugging client on another physical machine. While SMM executes, Protected Mode is essentially paused. The OS and hypervisor, therefore, are unaware of code executing in SMM. Because we run debugging code in SMM, we expose far fewer artifacts to the malware, enabling a more transparent execution environment for the debugging code than existing approaches.

The debugging client communicates with the target server using a GDB-like protocol with serial messages. We implement the basic debugging commands (e.g., breakpoints and memory/register examination) in the current prototype of MalT. Furthermore, we implement four techniques to provide step-by-step debugging: (1) instruction-level, (2) branch-level, (3) far control transfer level, and (4) near return transfer level. We also design a user-friendly interface for MalT to easily work with several popular debugging clients, such as IDAPro [46] and GDB.

MalT runs the debugging code in SMM without using a hypervisor. Thus, it has a smaller Trusted Code Base (TCB) than hypervisor-based debugging systems [37, 39, 40, 45], which significantly reduces the attack surface of MalT. Moreover, MalT is OS-agnostic and immune to hypervisor attacks (e.g., VM-escape attacks [6, 7]). Compared to existing bare-metal malware analysis [42, 43], SMM has the same privilege level as hardware. Thus, MalT is capable of debugging and analyzing kernel and hypervisor rookits as well [8, 125].

We develop a prototype of MalT on two physical machines connected by a serial cable. To demonstrate the efficiency and transparency of our approach, we test MalT with

popular packing, anti-debugging, anti-virtualization, and anti-emulation techniques. The experimental results show that MalT remains transparent against these techniques. Additionally, our experiments demonstrate that MalT is able to debug crashed kernels/hypervisors. MalT introduces a reasonable overhead: It takes about 12 microseconds on average to execute the debugging code without command communication. Moreover, we use popular benchmarks to measure the performance overhead for the four types of step-by-step execution on Windows and Linux platforms. The overhead ranges from 2 to 973 times slowdown on the target system, depending on the user’s selected instrumentation method.

6.1.2 Threat Model and Assumptions

Threat Model

MalT is intended to transparently analyze a variety of code that is capable of detecting or disabling typical malware analysis or detection tools. We consider two types of powerful malware in our threat model: armored malware and rootkits.

Armored malware or evasive malware [44] is a piece of code that employs anti-debugging techniques. Malicious code can be made to alter its behavior if it detects the presence of a debugger. There are many different detection techniques employed by current malware [13]. For example, `IsDebuggerPresent()` and `CheckRemoteDebuggerPresent()` are Windows API methods in the kernel32 library that return values based upon the presence of a debugger. Legitimate software developers can take advantage of such API calls to ease the debugging process in their own software. However, malware can use these methods to determine if it is being debugged to change or hide its malicious behavior from analysis.

Malware can also determine if it is running in a virtual machine or emulator [11, 14, 16]. For instance, Red Pill [18] can efficiently detect the presence of a VM. It executes a non-privileged (ring 3) instruction, `SIDT`, which reads the value stored in the Interrupt Descriptor Table (IDT) register. The base address of the IDT will be different in a VM than on a bare-metal machine because there is only one IDT register shared by both host-OS and guest-OS. Additionally, QEMU can be detected by accessing a reserved Model Specific Register

(MSR) [40]. This invalid access causes a General Protection (GP) exception on a bare-metal machine, but QEMU does not.

Rootkits are a type of stealthy malicious software. Specifically, they hide certain process information to avoid detection while maintaining continued privileged access to the system. There are a few types of rootkits ranging from user mode to firmware level. For example, kernel mode rootkits run in the operating system kernel (in ring 0) by modifying the kernel code or kernel data structures (e.g., Direct Kernel Object Modification). Hypervisor-level rootkits run in ring -1 and host the target operating system as a virtual machine. These rootkits intercept all of the operations including hardware calls in the target OS, as shown in Subvirt [8] and BluePill [125]. Since MalT runs in SMM with ring -2 privilege, it is capable of debugging user mode, kernel mode, and hypervisor-level rootkits. As no virtualization is used, MalT is immune to hypervisor attacks (e.g., VM escape [6, 7]). However, because firmware rootkits run in ring -2, MalT cannot detect these kind of rootkits.

Assumptions

As our trusted code (SMI handler) is stored in the BIOS, we assume the BIOS will not be compromised. We assume the Core Root of Trust for Measurement (CRTM) is trusted so that we can use Static Root of Trust for Measurement (SRTM) to perform the self-measurement of the BIOS and secure the boot process [103]. We also assume the firmware is trusted, although we can use SMM to check its integrity [106]. After booting, we lock the SMRAM to ensure the SMI handler code is trusted. We assume the debugging client and remote machine are trusted. Further, we consider an attacker that can have unlimited computational resources on our machine. We assume the attacker launches a single vulnerable application that can compromise the OS upon completing its first instruction. Lastly, we assume the attacker does not have physical access to the machines. Malicious hardware (e.g., hardware trojans) is also out of scope.

6.1.3 System Architecture

Figure 7.1 shows the architecture of the proposed MaT system. The remote client is equipped with a simple GDB-like debugger. The user inputs basic debugging commands (e.g., *list registers*), then the target machine executes the command and replies to the client as required. When a command is entered, the client sends a packet to the target server. The network packet contains the actual command. The target machine in SMM transmits a response packet containing the information requested by the command. Since the target machine executes the actual debugging command within the SMI handler, its operation remains transparent to the target application and underlying operating system.

As shown in the Figure 6.1, the debugging client first sends an SMI triggering packet to the debugging server. I will dedicate a PCI-based network card on the debugging server and make use of Message Signaled Interrupts (MSIs) to generate SMIs when the NIC receives packets. Secondly, once the debugging server enters into SMM, the debugging client starts to send debugging commands to the SMI handler on the server. Thirdly, the SMI handler transparently executes the corresponding commands (e.g., *list registers* and *set breakpoints*), and sends a response message back to the client.

The SMI handler on the debugging server inspects the debugged application at run time. If the debugged application hits a breakpoint, the SMI handler sends a "breakpoint hit" message to the debugging client and stays in SMM until further debugging commands are received. In addition, the proposed system can achieve step-by-step debugging via performance counters on the CPU. Next, I will detail each component of the system.

Remote Debugger: Client

The client can ideally implement a variety of popular debugging options. For example, I could use the SMI handler to implement the GDB protocol so that it would properly interface with a regular GDB client. Similarly, I might implement the necessary plug-in for IDAPro to correctly interact with my system. However, this would require implementing a full TCP network stack within the SMI handler.

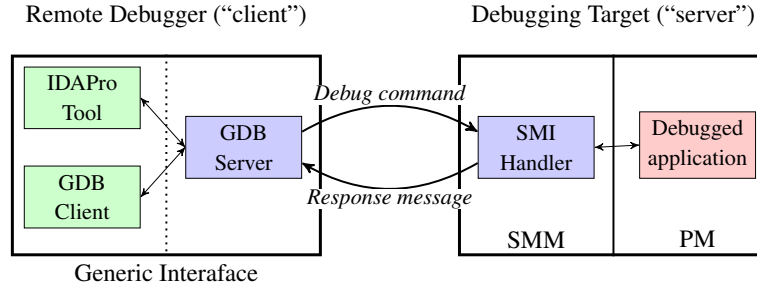


Figure 6.1: Architecture of MalT

Instead, I will implement a custom protocol with which to communicate between the remote client and the SMI handler. I will implement a small GDB-like client to simplify my implementation.

Debugging Target: Server

The debugging server consists of two parts: the SMI handler and the debugging target application. The SMI handler implements the critical debugging features (e.g., breakpoints and state reports), thus restricting the execution of debugging code to System Management Mode (SMM). The debugging target executes in Protected Mode as usual. I will cause SMIs frequently, and since the CPU state is saved within SMRAM each time the SMI handler executes, I can reconstruct useful information and perform typical debugging operations each time an SMI is triggered.

SMRAM contains architectural state information of the thread that was running when the SMI was triggered. Since the SMIs are produced regardless of the running thread, SMRAM will often contain states unrelated to the debugging target. In order to find the relevant state information, I must solve the semantic gap problem. By bridging the semantic gap within the SMI handler, I can ascertain state of the thread executing in Protected Mode. This is similar to Virtual Machine Introspection systems [32–34]. I would need to continue my analysis in the SMI handler only if the SMRAM state belongs to a thread I am interested in debugging. Otherwise, I can exit the SMI handler immediately.

Communicating Debugging Commands

In order to implement remote debugging in my system, I will define a simple network protocol used by the client and server hosts.

These commands are derived from basic GDB stubs, which are intended for debugging embedded software. The commands cover the basic debugging operations, which the client can expand upon. The small number of commands greatly simplifies the process of networking within the SMI handler. The remote client, after all, can glean an array of semantic and symbolic information from the command feedback and user interaction. For instance, there is no reason to store symbolic information on the target server; the user does not directly interact with the target machine while debugging software on it.

The overarching principle of the proposed system is that the communication between the client and the server should not affect normal traffic on the target machine. To simplify the network implementation, I will use two network interfaces on the target machine: one for normal traffic and one dedicated to debugging. I will install a PCI-based network card on the motherboard and port its driver into the SMI handler. Furthermore, I will use a crossover Ethernet cable to directly connect the client machine to the target machine. Since I directly attach the two machines, I statically assign the MAC addresses of the two machines to populate the packet header.

6.1.4 Implementation

The MalT system is composed of two main parts: 1) the debugging client used by the malware analyst and 2) the debugging server, which contains the SMI handler code and the target debugging application. In this section, we will describe how these two parts are implemented and used.

Debugging Client

The client machine consists of a simple command line application. The user can direct the debugger to perform useful tasks, such as setting breakpoints. For example, the user

writes simple commands such as `b 0xdeadbeef` to set a breakpoint at address `0xdeadbeef`. The specific commands are described in Table 6.1. We did not implement features such as symbols; such advanced features pose an engineering challenge that we will address in our future work. The client machine uses serial messages to communicate with the server.

Debugging Server

The target machine consists of a computer with a custom Coreboot-based BIOS. We changed the SMI handler in the Coreboot code to implement a simple debugging server. This custom SMI handler is responsible for all of the typical debugging functions found in other debuggers such as GDB. We implemented remote debugging functions via said serial protocol to achieve common debugging functions such as breakpoints, step-by-step execution, and state inspection and mutation.

Semantic Gap Reconstruction

As with Virtual Machine Introspection (VMI) systems [4], SMM-based systems encounter the well-known semantic gap problem. In brief, SMM cannot understand the semantics of raw memory. The CPU state saved by SMM only belongs to the thread that was running when the SMI was triggered. If we use step-by-step execution, there is a chance that another application is executing when the SMI occurs. Thus, we must be able to identify the target application so that we do not interfere with the execution of unrelated applications. This requires reconstructing OS semantics. Note that MalT has the same assumptions as traditional VMI systems [35].

In Windows, we start with the Kernel Processor Control Region (KPCR) structure associated with the CPU, which has a static linear address, `0xffdff000`. At offset `0x34` of KPCR, there is a pointer to another structure called `KdVersionBlock`, which contains a pointer to `PsActiveProcessHead`. The `PsActiveProcessHead` serves as the head of a doubly and circularly linked list of Executive Process (EProcess) structures. The EProcess structure is a process descriptor containing critical information for bridging the semantic gap in Windows

Table 6.1: Communication Protocol Commands

Message format	Description
R	A single byte, R is sent to request that all registers be read. This includes all the x86 registers. The order in which they are transmitted corresponds with the Windows trap frame. The response is a byte, r , followed by the registers $r_1 r_2 r_3 r_4 \dots r_n$.
mAAAAALLL	The byte m is sent to request a particular memory address for a given length. The address, A, is a 32-bit little-endian virtual address indicating the address to be read. The value L represents the number of bytes to be read.
Wr1r2r3...rn	The byte W is sent to request that the SMI handler write all of the registers. Each value r_i contains the value of a particular register. The response byte, + is sent to indicate that it has finished.
SAAAAALLLV...	The command, S, is sent when the debugger wants to write a particular address. A is the 32-bit, little-endian virtual address to write, L represents the length of the data to be written, and V is the memory to be written, byte-by-byte. The response is a byte, +, indicating that the operation has finished, or a - if it fails.
BAAAA	The B command indicates a new breakpoint at the 32-bit little-endian virtual address A. The response is + if successful, or - if it fails (e.g., trying to break at an already-broken address). If the SMI handler is triggered by a breakpoint (e.g., the program is in breakpoint debugging status), it will send a status packet with the single character, B, to indicate the program has reached a breakpoint and is ready for further debugging. The SMI handler will wait for commands from the client until the Continue command is received, whereupon it will exit from SMM.
C	The C command continues execution after a breakpoint. The SMI handler will send a packet with single character, +.
X	The X command clears all breakpoints and indicates the start of a new debugging session.
KAAAA	The K command removes the specified breakpoint if it was set previously. The 4-byte value A specifies the virtual address of the requested breakpoint. It responds with a single + byte if the breakpoint is removed successfully. If the breakpoint does not exist, it responds with a single -.
SI, SB, SF, SN	The SI command indicates stepping the system instruction by instruction. The SB command indicates stepping the system by taken branches. The SF command indicates stepping the system by control transfers including far call/jmp/ret. The SN command indicates stepping the system by near return instructions. The SMI handler replies with single character, +.

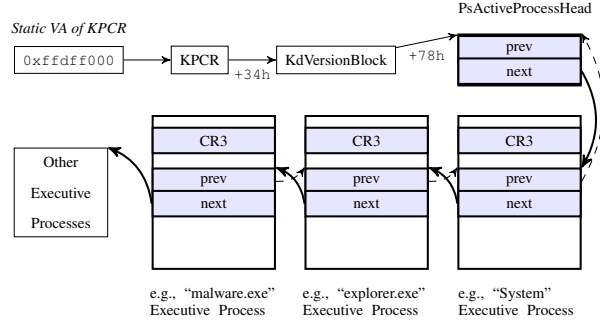


Figure 6.2: Finding a Target Application in Windows

NT kernels. Figure 6.2 illustrates this procedure.

In particular, the Executive Process contains the value of the CR3 register associated with the process. The value of the CR3 register contains the physical address of the base of the page table of that process. We use the name field in the `EPROCESS` or `task_struct` to identify the CR3 value of the target application when it executes first instruction. Since malware may change the name field, we only compare the saved CR3 with the current CR3 to identify the target process for further debugging. Alternatively, we can compare the EIP value with the target application's entry point. This method is simpler but less reliable since multiple applications may have the same entry point. Filling the semantic gap in Linux is a similar procedure, but there are fewer structures and thus fewer steps. Previous works [32,66] describe the method, which MalT uses to debug applications on the Linux platform. Note that malware with ring 0 privilege can manipulate the kernel data structures to confuse the reconstruction process, and current semantic gap solutions suffer from this limitation [35]. As with VMI systems, MalT does not consider the attacks that mutate kernel structures.

Triggering an SMI

The system depends upon reliable assertions of System Management Interrupts (SMIs). Because the debugging code is placed in the SMI handler, it will not work unless the CPU

can stealthily enter SMM.

In general, we can assert an SMI via software or hardware. The software method writes to an Advanced Configuration and Power Interface (ACPI) port to trigger an SMI, and we can use this method to implement software breakpoints. We can place an `out` instruction in the malware code so that when the malware’s control flow reaches that point, SMM begins executing, and the malware can be analyzed. The assembly instructions are:

```
mov $0x52f, %dx;  
out %ax, (%dx);
```

The first instruction moves the SMI software interrupt port number (0x2b on Intel, and 0x52f in our chipset [126]) into the `dx` register, and the second instruction writes the contents stored in `ax` to that SMI software interrupt port. (The value stored in `ax` is inconsequential). In total, these two instructions take six bytes: `66 BA 2F 05 66 EE`. While this method is straightforward, it is similar to traditional debuggers using `INT3` instructions to insert arbitrary breakpoints. The alternative methods described below are harder to detect by self-checking malware.

In MalT, we use two hardware-based methods to trigger SMIs. The first uses a serial port to trigger an SMI to start a debugging session. In order for the debugging client to interact with the debugging server and start a session, we reroute a serial interrupt to generate an SMI by configuring the redirection table in I/O Advanced Programmable Interrupt Controller (APIC). We use serial port COM1 on the debugging server, and its Interrupt Request (IRQ) number is 4. We configure the redirection table entry of IRQ 4 at offset 0x18 in I/O APIC and change the Delivery Mode (DM) to be SMI. Therefore, an SMI is generated when a serial message arrives. The debugging client sends a triggering message, causing the target machine to enter SMM. Once in SMM, the debugging client sends further debugging commands to which the target responds. In MalT, we use this method to trigger the first SMI and start a debugging session on the debugging server. The time of triggering the first SMI is right before each debugging session after reboot because MalT assumes that the first instruction of malware can compromise the system.

The second hardware-based method uses performance counters to trigger an SMI. This method leverages two architectural components of the CPU: performance monitoring counters and Local Advanced Programmable Interrupt Controller (LAPIC) [113]. First, we configure the Performance Counter Event Selection (PerfEvtSel0) register to select the counting event. There is an array of events from which to select; we use different events to implement various debugging functionalities. For example, we use the Retired Instructions Event (C0h) to single-step the whole system. Next, we set the corresponding performance counter (PerfCtr0) register to the maximum value. In this case, if the selected event happens, it overflows the performance counter. Lastly, we configure the Local Vector Table Entry (LVTE) in LAPIC to deliver SMIs when an overflow occurs. HyperSentry [62] and [127] use similar methods to switch from a guest VM to the hypervisor VMX root mode.

Breakpoints

Breakpoints are generally software- or hardware-based. Software breakpoints allow for unlimited breakpoints, but they must modify a program’s code, typically placing a single interrupt or trap instruction at the breakpoint. Self-checking malware can easily detect or interfere with such changes. On the other hand, hardware breakpoints do not modify code, but there can only be a limited number of hardware breakpoints as restricted by the CPU hardware. Stealthy breakpoint insertion is an open problem [36].

In MalT, we emulate the behavior of software breakpoints simply by modifying the target’s code to trigger SMIs. An SMI is triggered on our testbed by writing a value to the hardware port, 0x52f. In total, this takes six bytes. We thus save six bytes from the requested breakpoint address and replace them with the SMI triggering code. Thus, when execution reaches this point, the CPU enters SMM. We store the breakpoint in SMRAM, represented as 4 bytes for the address, 6 bytes for the original instruction, and one byte for a validity flag. Thus, each breakpoint occupies 11 bytes in SMRAM. When the application’s control reaches the breakpoint, it generates an SMI. In the SMI handler, we write the saved binary code back to the application text and revert the Extended Instruction Pointer (EIP)

register so that it will resume execution at that same instruction. Then, we wait in the SMI handler until the client sends a *continue* command. In order to remove an inserted breakpoint, the client can send a remove-breakpoint command and the SMI handler will disable that breakpoint by setting the enable flag to 0. However, this software breakpoint solution still makes changes to the application memory that are visible to malware—MalT does not use software breakpoints.

We implement a new hardware breakpoint technique in MalT. It relies on performance counters to generate SMIs. Essentially, we compare the EIP of the currently executing instruction with the stored breakpoint address during each cycle. We use 4 bytes to store the breakpoint address and 1 byte for a validity flag. In contrast to the software breakpoint method described above, we do not need to store instructions because we do not change any application memory. Thus, we need only 5 bytes to store such hardware breakpoints. For each Protected Mode instruction, the SMI handler takes the following steps: (1) Check if the target application is the running thread when the SMI is triggered; (2) check if the current EIP equals a stored breakpoint address; (3) start to count retired instructions in the performance counter, and set the corresponding performance counter to the maximum value; (4) configure LAPIC so that the performance counter overflow generates an SMI.

Breakpoint addresses are stored in SMRAM, and thus the number of active breakpoints we can have is limited by the size of SMRAM. In our system, we reserve a 512-byte region from `SMM_BASE+0xFC00` to `SMM_BASE+0xFE00`. Since each hardware breakpoint takes 5 bytes, we can store a total 102 breakpoints in this region. If necessary, we can expand the total region of SMRAM by taking advantage of a region called `TSeg`, which is configurable via the `SMM_MASK` register [113]. In contrast to the limited number of hardware breakpoints on the x86 platform, MalT is capable of storing more breakpoints in a more transparent manner.

Table 6.2: Stepping Methods in MalT

PMC Events	Description [113]
Retired instructions	Counts retired instructions, plus exceptions and interrupts
Retired taken branches	Includes all architectural control flow changes, plus exceptions and interrupts
Retired far control transfers	Includes far calls/jumps/returns, exceptions and interrupts
Retired near returns	Counts near return instructions (RET or RET lw) retired

Step-by-Step Execution Debugging

As discussed above, we break the execution of a program by using different performance counters. For instance, by monitoring the Retired Instruction event, we can achieve instruction-level stepping in the system. Table 6.2 summarizes the performance counters we used in our prototype. First, we assign the event to the PerfEvtSel0 register to indicate that the event of interest will be monitored. Next, we set the value of the counter to the maximum value (i.e., a 48-bit register is assigned $2^{48} - 2$). Thus, the next event to increase the value will cause an overflow, triggering an SMI. Note that the -2 term is used because the Retired Instruction event also counts interrupts. In our case, the SMI itself will cause the counter to increase as well, so we account for that change accordingly. The system becomes deadlocked if the value is not chosen correctly.

Vogl and Eckert [127] also proposed the use of performance counters for instruction-level monitoring. It delivers a Non-Maskable Interrupt (NMI) to force a VM Exit when a performance counter overflows. However, the work is implemented on a hypervisor. MalT leverages SMM and does not employ any virtualization, which provides a more transparent execution environment. Additionally, their work [127] incurs a time gap between the occurrence of a performance event and the NMI delivery, while MalT does not encounter this problem. Note that the SMI has priority over an NMI and a maskable interrupt as well. Among these four stepping methods, instruction-by-instruction stepping achieves fine-grained tracing, but at the cost of a significant performance overhead. Using the Retired Near Returns event causes low system overhead, but it only provides coarse-grained debugging.

6.1.5 Transparency Analysis

In terms of transparency, it heavily depends on its subjects. In this paper, we consider the transparency of four subjects. They are (1) virtualization, (2) emulation, (3) SMM, and (4) debuggers. Next, we discuss the transparency of these subjects one by one.

Virtualization: The transparency of virtualization is difficult to achieve. For instance, Red Pill [18] uses an unprivileged instruction `SIDT` to read the interrupt descriptor (IDT) register to determine the presence of a virtual machine. To work on a multi-processor system, Red Pill needs to use `SetThreadAffinityMask()` Windows API call to limit thread execution to one processor [15]. nEther [128] detects hardware virtualization using CPU design defects. Furthermore, there are many footprints introduced by virtualization such as well-known strings in memory [11], magic I/O ports [42], and invalid instruction behaviors [14]. Moreover, Garfinkel et al. [12] argued that building a transparent virtual machine is impractical.

Emulation: Researchers have used emulation to debug malware. QEMU simulates all the hardware devices including CPU, and malware runs on top of the emulated software. Because of the emulated environment, malware can detect it. For example, accessing a reserved or unimplemented MSR register causes a general protection exception, while QEMU does not raise an exception [16]. Table 6.3 shows more anti-emulation techniques. Although some of these defects could be fixed, determining perfect emulation is an undecidable problem [37].

SMM: SMM is a hardware feature existing in all x86 machines. Regarding its transparency, the Intel manual [63] specifies the following mechanisms that make SMM transparent to the application programs and operating systems: (1) the only way to enter SMM is by means of an SMI; (2) the processor executes SMM code in a separate address space (SMRAM) that is inaccessible from the other operating modes; (3) upon entering SMM, the processor saves the context of the interrupted program or task; (4) all interrupts normally

handled by the operating system are disabled upon entry into SMM; and (5) the **RSM** instruction can be executed only in SMM. Note that SMM steals CPU time from the running program, which is a side effect of SMM. For instance, malware can detect SMM based on the time delay. However, SMM is more transparent than virtualization and emulation.

Debuggers: An array of debuggers have been proposed for transparent debugging. These include in-guest [36, 46], emulation-based [38, 39], and virtualization-based [37, 41] approaches. MalT is an SMM-based system. As to the transparency, we only consider the artifacts introduced by debuggers themselves, not the environments (e.g., VMM or SMM). Ether [37] proposes five formal requirements for achieving transparency, including 1) high privilege, 2) no non-privileged side effects, 3) identical basic instruction execution semantics, 4) transparent exception handling, and 5) identical measurement of time. MalT satisfies the first requirement by running the analysis code in SMM with ring -2. We enumerate all of the side effects introduced by MalT in Section 6.1.5 and attempt to meet the second requirement in our system. Since MalT runs on bare metal, it immediately meets the third and fourth requirements. Lastly, MalT partially satisfies the fifth requirement by adjusting the local timers in the SMI handler. We further discuss the timing attacks below.

Side Effects Introduced by MalT

MalT aims to transparently analyze malware with minimum footprints. Here we enumerate the side effects introduced by MalT and show how we mitigate them. Note that achieving the highest level of transparency requires MalT to run in single-stepping mode.

CPU: We implement MalT in SMM, another CPU mode in the x86 architecture, which provides an isolated environment for executing code. After recognizing the SMI assertion, the processor saves almost the entirety of its state to SMRAM. As previously discussed, we rely on the performance monitoring registers and LAPIC to generate SMIs. Although these registers are inaccessible from user-level malware, attackers with ring 0 privilege can read and modify them. LAPIC registers in the CPU are memory-mapped, and its base address is normally at 0xFEE00000. In MalT, we relocate LAPIC registers to another physical

address by modifying the value in the 24-bit base address field of the `IA32_APIC_BASE` Model Specific Register (MSR) [63]. To find the LAPIC registers, attackers need to read `IA32_APIC_BASE` MSR first that we can intercept. Performance monitoring registers are also MSRs. `RDMSR`, `RDPMSR`, and `WRMSR` are the only instructions that can access the performance counters [113] or MSRs. To mitigate the footprints of these MSRs, we run MalT in instruction-by-instruction mode and adjust the return values seen by these instructions before resuming Protected Mode. If we find a `WRMSR` to modify the performance counters, the debugger client will be notified.

Memory and Cache: MalT uses an isolated memory region (SMRAM) from normal memory in Protected Mode. Any access to this memory in other CPU modes will be redirected to VGA memory. Note that this memory redirection occurs in all x86 machines, even without MalT; this is not unique to our system. Intel recently introduced System Management Range Registers (SMRR) [63] that limits cache references of addresses in SMRAM to code running in SMM. This is the vendor’s response to the cache poisoning attack [58]; MalT does not flush the cache when entering and exiting SMM to avoid cache-based side-channel detection.

IO Configurations and BIOS: MalT reroutes a serial interrupt to generate an SMI to initialize a debugging session, and the modified redirection table entry in I/O APIC can be read by malware with ring 0 privilege. We change the redirection table entry back to its original value to remove this footprint in the first generated SMI handler. Once SMM has control of the system, the SMIs are triggered by configuring performance counters. MalT uses a custom BIOS, Coreboot, to program the SMM code. An attacker with ring 0 privilege can check the hash value of the BIOS to detect the presence of our system. To avoid this fingerprint, we flash the BIOS with the original image before the debugging process using the tool Flashrom [112], and it takes about 28 seconds to flash the Coreboot with the original AMI BIOS. At that time, the SMI handler, including the MalT code, has been loaded into SMRAM and locked. Note that we also need to reflash the Coreboot image for the next system restart.

Timing: There are many timers and counters on the motherboard and chipsets, such as the Real Time Clock (RTC), the Programmable Interval Timer (8253/8254 chip), the High Precision Event Timer (HPET), the ACPI Power Management Timer, the APIC Timer, and the Time Stamp Counter (TSC). Malware can read a timer and calculate its running time. If the time exceeds a certain threshold, malware can conclude that a debugger is present. For the configurable timers, we record their values after switching into SMM. When SMM exits, we set the values back using the recorded values minus the SMM switching time. Thus, the malware is unaware of the time spent in the SMI handler. However, some of the timers and counters cannot be changed, even in SMM. To address this problem, we adjust the return values of these timers in instruction-level stepping mode. For example, the `RDTSC` instruction reads the TSC register and writes the value to the `EAX` and `EDX` registers. While debugging, we can check if the current instruction is `RDTSC` and adjust the values of `EAX` and `EDX` before leaving the SMI handler.

Unfortunately, MalT cannot defend against timing attacks involving an external timer. For instance, malware can send a packet to a remote server to get correct timing information (e.g., NTP service). In this case, malware can detect the presence of our system and alter its behavior accordingly. One potential solution to address this problem is to intercept the instruction that reaches out for timing information and prepare a fake time for the OS. Naturally, this would not be foolproof as an attacker could retrieve an encrypted time from a remote location. Such attacks are difficult to contend with because we cannot always know when a particular packet contains timing information. To the best of our knowledge, all existing debugging systems with any measurable performance slowdown suffer from this attack. As stated in Ether [37], defending against external timing attacks for malware analysis systems is Turing undecidable. However, external timing attacks require network communications and thus dramatically increase the probability that the malware will be flagged. We believe that this deterrent—malware will avoid using external timing attacks precisely because it wants to minimize its footprint on the victim’s computer, including using spin loops. We can also analyze portions of the malware separately and amortize the

analysis time.

Analysis of Anti-debugging, -VM, and -emulation Techniques

To analyze the transparency of MalT system, we employ anti-debugging, anti-virtualization, and anti-emulation techniques from [11, 13–16] to verify our system. Since MalT runs on a bare-metal machine, these anti-virtualization techniques will no longer work on it. Additionally, MalT does not change any code or the running environments of operating systems and applications so that normal anti-debugging techniques cannot work against it. For example, the debug flag in the PEB structure on Windows will not be set while MalT is running. Table 6.3 summarizes popular anti-debugging, anti-virtualization, and anti-emulation techniques, and we have verified that MalT can evade all these detection techniques.

Testing with Packers

Packing is used to obfuscate the binary code of a program. It is typically used to protect the executable from reverse-engineering. Nowadays, malware writers also use packing tools to obfuscate their malware. Packed malware is more difficult for security researchers to reverse-engineer the binary code. In addition, many packers contain anti-debugging and anti-VM features, further increasing the challenge of reverse-engineering packed malware.

To demonstrate the transparency of MalT, we use popular packing tools to pack the `Notepad.exe` application in a Windows environment and run this packed application in MalT with near return stepping mode, OllyDbg [47], DynamoRIO [130], and a Windows virtual machine, respectively. Ten packing tools are used, including UPX, Obsidium, AS-Pack, Armadillo, Themida, RLPack, PELock, VMProtect, eXPressor, and PECompact. All these packing tools enable the settings for anti-debugging and anti-VM functions if they have them. After running the packed `Notepad.exe`, if the Notepad window appears, we know that it has launched successfully. Table 6.4 lists the results. All the packing tools except UPX, ASPack, and RLPack can detect OllyDbg. Obsidium, Armadillo, Themida,

Table 6.3: Summary of Anti-debugging, Anti-VM, and Anti-emulation Techniques

Anti-debugging [13, 123]	
API Call	Kernel32!IsDebuggerPresent returns 1 if target process is being debugged ntdll!NtQueryInformationProcess: ProcessInformation field set to -1 if the process is being debugged kernel32!CheckRemoteDebuggerPresent returns 1 in debugger process NtSetInformationThread with ThreadInformationClass set to 0x11 will detach some debuggers kernel32!DebugActiveProcess to prevent other debuggers from attaching to a process
PEB Field	PEB!IsDebugged is set by the system when a process is debugged PEB!NtGlobalFlags is set if the process was created by a debugger
Detection	ForceFlag field in heap header (+0x10) can be used to detect some debuggers UnhandledExceptionFilter calls a user-defined filter function, but terminates in a debugging process TEB of a debugged process contains a NULL pointer if no debugger is attached; valid pointer if some debuggers are attached Ctrl-C raises an exception in a debugged process, but the signal handler is called without debugging Inserting a Rogue INT3 opcode can masquerade as breakpoints Trap flag register manipulation to thwart tracers If entryPoint RVA set to 0, the magic MZ value in PE files is erased ZwClose system call with invalid parameters can raise an exception in an attached debugger Direct context modification to confuse a debugger 0x2D interrupt causes debugged program to stop raising exceptions Some In-circuit Emulators (ICEs) can be detected by observing the behavior of the undocumented 0xF1 instruction Searching for 0xCC instructions in program memory to detect software breakpoints TLS-callback to perform checks
Anti-virtualization	
VMWare	Virtualized device identifiers contain well-known strings [11] checkvm software [129] can search for VMWare hooks in memory Well-known locations/strings associated with VMWare tools
Xen	Checking the VMX bit by executing CPUID with EAX as 1 [128] CPU errata: AH4 erratum [128]
Other	LDTR register [15] IDTR register (Red Pill [18]) Magic I/O port (0x5658, 'VX') [42] Invalid instruction behavior [14] Using memory deduplication to detect various hypervisors including VMware ESX server, Xen, and Linux KVM [17]
Anti-emulation	
Bochs	Visible debug port [11]
QEMU	cpuid returns less specific information [40] Accessing reserved MSR registers raises a General Protection (GP) exception in real hardware; QEMU does not [16] Attempting to execute an instruction longer than 15 bytes raises a GP exception in real hardware; QEMU does not [16] Undocumented icebp instruction hangs in QEMU [40], while real hardware raises an exception Unaligned memory references raise exceptions in real hardware; unsupported by QEMU [16] Bit 3 of FPU Control Word register is always 1 in real hardware, while QEMU contains a 0 [40]
Other	Using CPU bugs or errata to create CPU fingerprints via public chipset documentation [16]

Table 6.4: Running Packed `Notepad.exe` under Different Environments

Packing Tool	MalT	OllyDbg V1.10	DynamoRIO V4.2.0-3	VMware Fusion V6.0.2
UPX V3.08	OK	OK	OK	OK
Obsidium V1.4	OK	Access violation at 0x00000000	Segmentation fault	OK
ASPack V2.29	OK	OK	OK	OK
Armadillo V2.01	OK	Access violation at 0x42434847	Crash	Crash
Themida V2.2.3.0	OK	Privileged instruction exception	Exception at 0x10a65d7	Message: cannot run under a VM
RLPack V1.21	OK	OK	OK	OK
PELock V1.0694	OK	Display message and terminate	Segmentation fault	OK
VMProtect V2.13.5	OK	Message: a debugger was found	OK	Crash
eXPressor V1.8.0.1	OK	Message: unknown executable format	Segmentation fault	Crash
PECompact V3.02.2	OK	Access violation at 0x00000000	OK	OK

PELock, and eXPressor are able to detect DynamoRIO, and the VM can be detected by Armadillo, Themida, VMProtect, and eXpressor. In contrast, MalT remains transparent to all these packing tools as we expected.

Transparency of MalT

Section 6.1.5 show that existing anti-debugging, anti-VM, anti-emulation, and packing techniques cannot detect the presence of MalT. This is because the current techniques are not targeting MalT’s functions or code, so it is possible that future malware could detect MalT due to the ever-present cat-and-mouse game between attackers and defenders. As for ‘tomorrow’s malware,’ we enumerate and mitigate the side effects introduced by MalT in Section 6.1.5. Note that mitigating all footprints require stepping instructions granularity. As with other debugging systems, MalT cannot defend against external timing attacks.

Running Environment Used by MalT: MalT is built on SMM so that the transparency of MalT depends on the implications of SMM usage. Since SMM is not intended for debugging, the hardware devices and software on the system may not expect this usage, which may introduce side-channel footprints for attackers to detect MalT (e.g., performance slowdown and frequent switching). However, we believe using SMM is more transparent than using virtualization or emulation as done in previous systems due to its minimal TCB and attack surface.

Towards True Transparency: Debugging transparency is a challenging and recently active problem in the security community. Unlike previous solutions that use virtualization

or emulation, MalT isolates the execution in the CPU, which provides a novel idea of addressing the transparency problem. Although MalT is not fully transparent, we would like to draw the attention to the community of this hardware-based approach because the running environment of the debugger is more transparent than those of previous systems (i.e., virtualization and emulation). Moreover, we further argue hardware support for truly transparent debugging. For instance, there could be a dedicated and well-designed CPU mode for debugging, perhaps with performance counters that are inaccessible from other CPU modes; that provides a transparent switching method between CPU modes.

6.1.6 Evaluation

Testbed Specification and Code Size

We evaluated MalT on two physical machines. The target server used an ASUS M2V-MX_SE motherboard with an AMD K8 northbridge and a VIA VT8237r southbridge. It has a 2.2 GHz AMD LE-1250 CPU and 2 GB Kingston DDR2 RAM. The target machine used Windows XP SP3, CentOS 5.5 with kernel 2.6.24, and Xen 3.1.2 with CentOS 5.5 as domain 0. To simplify the installation, they are installed on three separate hard disks, and the SeaBIOS manages the booting. The debugging client was a Dell Inspiron 15R laptop with Ubuntu 12.04 LTS. It uses a 2.4 GHz Intel Core i5-2430M CPU and 6 GB DDR3 RAM. We used a USB-to-serial cable to connect two machines.

We used `cloc` [131] to compute the number of lines of source code. Coreboot and its SeaBIOS payload contained 248,421 lines. MalT added about 1,500 lines of C code in the SMI handler. After compiling the Coreboot code, the size of the image was 1MB, and the SMI handler contained 3,098 bytes. The debugger client contained 494 lines of C code.

Debugging with Kernels and Hypervisors

To demonstrate that MalT is capable of debugging kernels and hypervisors, we intentionally crash the OS kernels and domain 0 of a Xen hypervisor and then use MalT to debug them. For the Linux kernel and domain 0 of the Xen hypervisor, we simply run the command

`echo c > /proc/sysrq-trigger`, which performs a system crash by a NULL pointer dereference. To force a Blue Screen of Death (BSOD) in Windows, we create a new value named `CrashOnCtrlScroll` in the registry and set it equal to a `REG_DWORD` value of `0x01`. Then, the BSOD can be initiated by holding the Ctrl key and pressing the Scroll Lock key twice. After a system crashes, MalT can start a debugging session by sending an SMI triggering message. In our experiments, MalT is able to examine all of the CPU registers and the physical memory of the crashed systems.

Breakdown of Operations in MalT

In order to understand the performance of our debugging system, we measured the time elapsed during particular operations in the SMI handler. We used the Time Stamp Counter (TSC) to measure the number of CPU cycles elapsed during each operation; we multiplied the clock frequency by the delta in TSCs.

After a performance counter triggers an SMI, the system hardware automatically saves the current architectural state into SMRAM and begins executing the SMI handler. The first operation in the SMI handler is to identify the last running process in the CPU. If the last running process was not the target malware, we only need to configure the performance counter register for the next SMI and exit from SMM. Otherwise, we perform several checks. First, we check for newly received messages and whether a breakpoint has been reached. If there are no new commands and no breakpoints to evaluate, we reconfigure the performance counter registers for the next SMI. Table 6.5 shows a breakdown of the operations in the SMI handler if the last running process is the target malware in instruction-by-instruction stepping mode. This experiment shows the mean, standard deviation, and 95% confidence interval of 25 runs. The SMM switching time takes about 3.29 microseconds. Command checking and breakpoint checking take about 2.19 microseconds in total. Configuring performance monitoring registers and SMI status registers for subsequent SMI generation takes about 1.66 microseconds. Lastly, SMM resume takes 4.58 microseconds. Thus, MalT takes about 12 microseconds to execute an instruction without

Table 6.5: Breakdown of SMI Handler (Time: μs)

Operations	Mean	STD	95% CI
SMM switching	3.29	0.08	[3.27,3.32]
Command and BP checking	2.19	0.09	[2.15,2.22]
Next SMI configuration	1.66	0.06	[1.64,1.69]
SMM resume	4.58	0.10	[4.55,4.61]
Total	11.72		

Table 6.6: Stepping Overhead on Windows and Linux (Unit: Times of Slowdown)

Stepping Methods	Windows					Linux				
	π	<i>ls</i>	<i>ps</i>	<i>pwd</i>	<i>tar</i>	π	<i>ls</i>	<i>ps</i>	<i>pwd</i>	<i>tar</i>
Retired far control transfers	2	2	2	3	2	2	3	2	2	2
Retired near returns	30	21	22	28	29	26	41	28	10	15
Retired taken branches	565	476	527	384	245	192	595	483	134	159
Retired instructions	973	880	897	859	704	349	699	515	201	232

debugging commands communication.

Step-by-Step Debugging Overhead

In order to demonstrate the efficiency of our system, we measure the performance overhead of the four stepping methods on both Windows and Linux platforms. We use a popular benchmark program, SuperPI [118] version 1.8, on Windows and version 2.0 on Linux. SuperPI is a single-threaded benchmark that calculates the value of π to a specific number of digits and outputs the calculation time. This tightly written, arithmetic-intensive benchmark is suitable for evaluating CPU performance. Additionally, we use four popular Linux commands, `ls`, `ps`, `pwd`, and `tar` to measure the overhead. `ls` is executed with the root directory; `pwd` is executed under home directory; and `tar` is used to compress a hello-world program with 7 lines of C code. We install `Cygwin` on Windows to execute these commands. First, we run the programs and record their runtimes. Next we enable each of the four stepping methods separately and record the runtimes. SuperPI calculates 16 K digits of π , and we use shell scripts to calculate the runtimes of the Linux commands.

Table 6.6 shows the performance slowdown introduced by step-by-step debugging. The first column specifies four different stepping methods; the following five columns show the slowdown on Windows, which is calculated by dividing the current running time by the base running time; and the last five columns show the slowdown on Linux. It is evident that far control transfer (e.g., call instruction) stepping only introduces a 2x slowdown on Windows and Linux, and this facilitates coarse-grained tracing for malware debugging. As expected, fine-grained stepping methods introduce more overhead. The instruction-by-instruction debugging causes about 973x slowdown on Windows for running SuperPI, which demonstrates the worst-case performance degradation in our four debugging methods. This high runtime overhead is due to the 12-microsecond cost of every instruction (as shown in Table 6.5) in the instruction-stepping mode. One way to improve the performance is to reduce the time used for SMM switching and resume operations by cooperating with hardware vendors. Note that MalT is three times as fast as Ether [37, 40] in the single-stepping mode.

Despite a three order-of-magnitude slowdown on Windows, the debugging target machine is still usable and responsive to user interaction. In particular, the instruction-by-instruction debugging is intended for use by a human operator from the client machine, and we argue that the user would not notice this overhead while entering the debugging commands (e.g., **Read Register**) on the client machine. We believe that achieving high transparency at the cost of performance degradation is necessary for certain types of malware analysis. Note that the overhead in Windows is larger than that in Linux. This is because 1) the semantic gap problem is solved differently in each platform, and 2) the implementations of the benchmark programs are different.

Chapter 7: Using Hardware Isolated Execution Environments for Executing Sensitive Workloads

7.1 TrustLogin: Securing Password-Login

7.1.1 Introduction

Logging in is part of daily practice in the modern world. We use it to authenticate ourselves to applications for resource accesses. Consequently, login credentials are one of the top targets for attackers. For example, keylogger malware was found on UC Irvine health center computers in May 2014, and it is estimated that 1,813 students and 23 non-students were impacted [132]. Additionally, it is reported that attackers have stolen credit card information from customers who shopped at 63 Barnes & Noble stores using keyloggers [133]. A case study has shown that 10,775 unique bank account credentials were stolen by keyloggers in a seven-month period [134]. Protecting login credentials is a critical part of daily life.

Nowadays, operating systems are complex and rely on millions of lines of code to operate (e.g., the Linux kernel has about 17 million lines of code [135]). The large Trusted Computing Base (TCB) of these OSes inevitably creates vulnerabilities that could be exploited by attackers. The Common Vulnerabilities and Exposures (CVE) list shows that 240 vulnerabilities have been found for the Linux kernel [136]. An attacker can easily leverage these vulnerabilities to create rootkits and keyloggers.

On top of an untrusted OS, no matter how secure the network applications are, the sensitive data used by secure applications is at risk of leakage. For example, an attacker can install a stealthy keylogger after compromising the OS, so the banking login information entered in a web browser can be obtained by the attacker without a user awareness. Therefore, the protection of the user's sensitive data during network operations is crucial;

we need to prevent malicious behaviors of attackers on network applications.

In this chapter, I present TrustLogin [137], a framework to securely perform login operations on commodity operating systems. Even if the operating system and applications are compromised, an attacker is not able to reveal the login password from the host. TrustLogin leverages System Management Mode (SMM), a CPU mode that exists in x86 architecture, to transparently protect the login credentials from keyloggers. Since we assume the attackers have ring 0 privilege, all of the software including the operating system cannot be trusted. SMM is a separate CPU mode with isolated execution memory, and it is inaccessible from the OS, which satisfies the needs of our system.

When users enter their passwords, TrustLogin automatically switches into SMM and records the keystroke. It provides a randomly generated string to the OS kernel and then the network driver prepares the login packets. When the login packets arrive at the network card, TrustLogin switches into SMM again and replaces the placeholder with the real password. Under the protection of TrustLogin, rootkits (e.g, keyloggers) cannot steal the sensitive data even with ring 0 privilege. To combat spoofing attacks, we implement two novel techniques that ensure the trust path when switching to SMM. They use the LED lights on keyboard and the PC speaker to interact with users. More importantly, TrustLogin does not modify application- and OS-code, and it is transparent from client and server sides.

To demonstrate the effectiveness of our approach, we conduct two study cases to use TrustLogin with legacy and secure applications. We test TrustLogin with real-world keyloggers on both Windows and Linux platforms, and the experiment results show that TrustLogin is able to protect the login password against them. We also measure the performance overhead introduced by executing code in SMM. Our results show SMM switching only takes about 8 microseconds. TrustLogin takes 33 milliseconds to store and replace a keystroke and most of the time is consumed by the trusted path indication (i.e., playing a melody and showing a LED light sequence); it spends 30 microseconds on injecting the password back to a login packet for the tested application.

7.1.2 Threat Model and Assumptions

Threat Model

Keyloggers can be classified into two types: hardware- and software-based. Hardware-based keyloggers are small electronic devices that are used to capture the keystrokes. They are often built in the keyboard itself and have separate non-volatile memory to store the keystrokes. Hardware keyloggers do not require installation of any software or power source for their operations. For instance, there are some commercial hardware keyloggers available [138]. In this paper, we do not consider this type of keylogger, and we assume the keyboard is not malicious.

Software-based keyloggers are installed within the operating system, and most of the keyloggers in the real world are this type. There are two kinds of software keyloggers: user- and kernel-level. For instance, a user-level keylogger can use the `GetKeyboardState` API function to capture keystrokes in Windows. This kind of keylogger is efficient but also easily detected. Kernel-level keyloggers are implemented at the kernel level and require administrator privilege to install. For example, a keyboard filter driver can be used to stealthily capture keystrokes [139]. TrustLogin considers software-based keyloggers as the threat model, and it guarantees that keystrokes cannot be stolen if such a keylogger is present.

Assumptions

TrustLogin assumes that the attackers have unlimited computing resources and can exploit zero-day vulnerabilities of the host OS and desktop applications. We only consider attacks against the host machine; network attacks are out of the scope of this paper. We do not consider phishing attacks, which trick users to send their credentials to a remote host. We assume the hardware and firmware of the host machine are trusted, and the attacker cannot flash the BIOS or modify the firmware. We assume SMRAM is locked and remains intact after boot, and the attacker cannot change the SMI handler. We assume that the attacker does not have physical access to the machine. We do not consider Denial-of-Service (DoS)

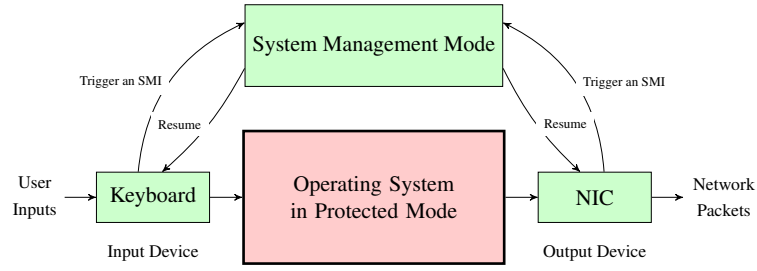


Figure 7.1: Architecture of TrustLogin

attacks against our system. Ring 0 malware can easily disable SMI triggering and stop the login process.

7.1.3 System Architecture

Figure 7.1 shows the architecture of TrustLogin. There are four rectangles in the figure; the green rectangles represent trusted components, including the keyboard, Network Interface Card (NIC), and System Management Mode (SMM). The red rectangle represents the operating system in Protected Mode, which may have been compromised by attackers. When a user inputs the sensitive information (e.g., password) from the keyboard, the keyboard automatically triggers an SMI for every key press. The SMI handler (which executes in SMM) records the keystrokes and inserts bogus place-holders in the keyboard buffer. After resuming Protected Mode, the OS only handles the place-holders. In other words, the attackers with ring 0 privilege can only retrieve the string of place-holders. When the login packet is about to transmit, TrustLogin triggers another SMI by using the network card. The SMI handler replaces the bogus place-holders with the original keystrokes in the network packet. We also make sure the packet leaves the network card within the SMI handler so that malware cannot read the packet. Next, we explain the system step by step.

Entering Secure Input Mode

In TrustLogin, we have two modes for the system. One is the secure input mode, and the other is the normal input mode. In the secure input mode, TrustLogin intercepts all of the keystrokes and protects them from the keyloggers or rootkits. When the user is about to enter the sensitive information (e.g., password), he or she needs to switch to the secure input mode. Past systems have used a variety of ways to notify the system. For instance, Bumpy [51] uses “@@” as a Secure Attention Sequence (SAS) to signal to the system that the user is about to enter sensitive inputs.

One requirement of switching into the Secure Input Mode is that the entering method should be rarely used by default. Ideally, it should be unique (e.g., a dedicated hardware switch [65]), but SAS-like “@@” sequence also works. The other requirement is usability. In TrustLogin, we simply use the key combination, **Ctrl+Alt+1**, to signal our system and enter the secure input mode. When TrustLogin reads an Enter key in the secure input mode, it stops intercepting keystrokes and switches to the normal input mode. Since users often end password inputs by pressing Enter, this is reasonable.

Intercepting Keystrokes

TrustLogin intercepts every keystroke and records them in the SMRAM in the secure input mode. Before introducing how keystrokes are intercepted in TrustLogin, we will explain how keystrokes are handled normally.

The input/output devices (e.g., keyboard) connect to the Southbridge (a.k.a. I/O controller Hub). Whenever a key is pressed or released, the keyboard notifies the I/O Advanced Programmable Interrupt Controller (APIC) in the Southbridge. I/O APIC looks up the I/O redirection table based on the Interrupt Request (IRQ), and then creates the corresponding interrupt message. The IRQ for the keyboard is 1, and the interrupt message includes the Delivery Mode (DM), Fixed, and the interrupt vector, **0x93**. The interrupt message goes through the PCI and system buses, and arrives at the local APIC in the CPU. Based on the DM and interrupt vector, the local APIC looks up the Interrupt Descriptor Table (IDT),

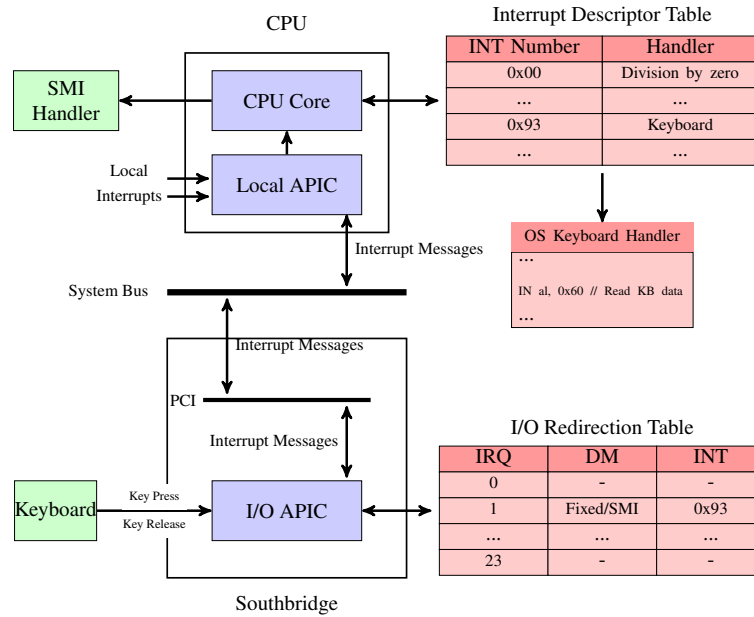


Figure 7.2: Keystroke Handling in TrustLogin

and then the CPU jumps to the base address of the OS keyboard interrupt handler. The OS keyboard interrupt handler starts to execute keyboard handling functions. Specifically, it reads the keyboard data registers by accessing port 0x60 and may display the key value on the display monitor. Figure 7.2 shows the keystroke handling process.

Note that there are two interrupts for each keystroke: key press and key release. When the key is pressed or released, the keyboard sends a message known as “scan code” to the keyboard controller output buffer that the OS handler read later. There are two different types of scan codes: “make codes” and “break codes.” A make code is sent when a key is pressed, and a break code is sent when a key is released. Every key has a unique make code and break code. There are three different sets of scan codes. Our keyboard uses the scan code set 1 [140]. For example, the make code of the A key is 0x1E, and its break code is 0x9E. If a user keeps holding the key, the keyboard would continue to send interrupts with the make code. When the user releases the key, the break code would be written into the output buffer of the keyboard controller.

To record the keystrokes, TrustLogin raises an SMI during the key press or release handling process and saves the keystrokes in the SMRAM. Additionally, we also save the scan code set mapping in the SMI handler to figure out which key is pressed or released. Next, we explain two approaches that we implement to trigger an SMI during the keystroke handling.

TrustLogin can use hardware I/O traps to generate an SMI [113]. The I/O trap feature allows SMI trapping on access to any I/O port using `IN` or `OUT` instruction. As mentioned, the OS interrupt handler needs to read the keyboard data register by accessing port `0x60`. If we configure the SMI I/O trap, an SMI would be triggered when the OS handler reads the keyboard data registers. In this way, we are able to intercept all keystrokes and save them in the SMRAM. When the OS keyboard interrupt handler executes `IN al, 0x60` instruction, the system automatically generates an SMI. However, this `IN` instruction would not be executed again when resuming the OS in Protected Mode. To address this problem, the SMI handler needs to read the key value from the keyboard data register and store it in the `EAX` as if no trap has been created. Additionally, we need to disable the SMI I/O trap in the SMI handler. Otherwise, an SMI will be buffered when accessing the I/O port. In that case, the SMI will be immediately triggered after exiting SMM and the system will halt. Note that Wecherowski demonstrated similar triggering approach in [141].

The other approach of triggering SMIs is to reroute keyboard interrupt by reconfiguring I/O APIC. As shown in Figure 7.2, The Delivery Mode (DM) of the I/O redirection table can be configured as “SMI” instead of “Fixed” normally. In other words, we are able to deliver an SMI to the CPU for every keyboard interrupt; that is, every key press causes our code to execute in SMM. Next, we store the keystroke to the SMRAM by reading the keyboard data register in the SMI handler.

We read the 1-byte scan code from the keyboard data register by reading I/O port `0x60`. After extracting the scan code, we map the scan code to the key value using the scan code set 1 table. Next, we store the key in the SMRAM. Since this approach reroutes the keyboard interrupt to an SMI, the original keyboard interrupt is not handled. To address

this problem, we configure the keyboard control register (i.e., IO port 0x64) to reissue the interrupt. We write the command code, 0xD2, to the control register. This special command means the next byte written to the keyboard data register (i.e., I/O port 0x60) will be as if it came from the keyboard [68]. We write a replaced scan code back to the data register after writing the command code. After exiting SMM, another interrupt is generated due to the new data in the keyboard data register. Additionally, we need to disable SMI triggering in I/O APIC when reissuing the interrupt in the SMI handler. This makes sure that the reissued interrupt is a normal keyboard interrupt with “fixed” as the DM. Otherwise, an immediate SMI will be generated after exiting SMM, which causes an infinite loop (deadlock). The method for accessing the I/O APIC or keyboard controller is specified in the Southbridge datasheet [68].

Embleton et al. also used a similar approach to generate SMIs in [55]. However, we did not see that the I/O read operation of the keyboard data register was destructive in the SMI handler; we were able to read the data register multiple times until a new value was written. Additionally, it uses interprocessor interrupts (IPI) to reissue the interrupt by configuring the Interrupt Command Register (ICR), while we simultaneously write to the keyboard control register to reissue the normal interrupt.

Universal Serial Bus (USB) is a popular external interface standard that enables communication between the computer and other peripherals. There are currently three versions of USB in use: USB 1.1, 2.0, and 3.0. A USB system has a host controller, and it sits between the USB device and the operating system. USB 1.1 uses Universal Host Controller Interface (UHCI) [142]; USB 2.0 uses Enhanced Host Controller Interface (EHCI) [143]; and the recent USB 3.0 uses eXtensible Host Controller Interface (XHCI) [144]. From the manuals of these standards, all support triggering SMIs. For instance, XHCI uses a 32-bit register to enable SMIs for every xHCI/USB event it needs to track, and we are able to trigger an SMI for every key press required by TrustLogin. This register is located at xHCI Extended Capabilities Pointer (XECp) + 0x04, and we can find XECp from the base address of the XHCI + 0x10. Similar registers that enable SMIs can also be found at EHCI

and UHCI. Moreover, Schiffman and Kaplana [145] demonstrated that USB keyboards can generate SMIs.

Generating Placeholders

To replace the original password, we generate a placeholder for each keystroke intercepted in the SMI handler. One of the simplest methods is to replace each keystroke with a constant character (e.g., character ‘p’). However, this method cannot pass the security checks that ensure the strength of the password. For instance, most of the password policies require that passwords contain at least one digit, one lowercase character, one uppercase character, and one special character. Although these checks are usually performed on the server side, they could be done on the client application. To address this problem, TrustLogin replaces a keystroke based on its type. TrustLogin substitutes the original keystroke with a random one of the same type.

We use a linear-congruential algorithm to generate a pseudo-random number n in the SMI handler. The parameters of the linear-congruential algorithm we used are from Numerical Recipes [114]. Next, we use $n \bmod k$, where k is the cardinality of the corresponding type (e.g., 26 each for lower- or uppercase characters) to generate a random character. In terms of the special characters, different applications or servers may have a different set of valid special characters. For instance, the American Express website does not allow special characters like ‘.’ in the password, while Bank of America and CitiBank do accept it. TrustLogin assumes the application allows six special characters as follows: dot, underscore, star, percent, question mark, and sharp. We can always update the set of special characters based on the application requirements. Next, we discuss how the network card intercepts packets and replaces the placeholders with the original password.

Intercepting Network Packets

TrustLogin starts to intercept the network packets when the Enter key is received in the secure input mode. This means the user has finished entering the password and the OS is

about to transmit the login credentials. We use a popular commercial PCI-based network card, Intel e1000 [146], to demonstrate this in TrustLogin.

Message Signaled Interrupts (MSIs) are an optional feature incorporated into PCI devices. They essentially allow a PCI device to generate an interrupt without having to make use of a physical interrupt pin on the connector. Introduced in PCI version 2.2, MSIs allow the device to send a variety of different interrupts to the CPU via the chipset. One such interrupt is the SMI. We can configure the MSI configuration registers (offset 0xF0 to 0xFF) in the PCI configuration space to enable SMI triggering.

When MSIs are enabled, the network card generates a message when any of the unmasked bits in the Interrupt Cause Read register (ICR) are set to 1 [146]. The ICR contains all interrupt conditions for the network card. Each time an interrupt occurs, the corresponding interrupt bit is set in the register. The interrupts are enabled through Interrupt Mask Set/Read Register (IMS). For instance, the first bit of IMS sets a mask for Transmit Descriptor Written Back (TDWB). When the hardware finishes transmitting a packet, it sets a status bit back to the transmit descriptor; this action could be an interrupt condition. In TrustLogin, we reroute this interrupt to an SMI by using MSI. This means we can trigger an SMI for each packet when it is transmitted. In the SMI handler, we then inspect all of the transmit descriptors in the transmit queue and search for the login packet. It is possible that the first packet that generates the SMI is the login packet. To address this edge case, we create a transmit descriptor in the SMI handler beforehand and make sure the first SMI from NIC is triggered by this transmit descriptor. This transmit descriptor is created when TrustLogin enables the NIC's SMI triggering by rerouting TDWB interrupts to an SMI. Note that the transmit queue may become empty before finding the login packet, so we may miss the first transmit descriptor that arrives at the empty transmit queue. Thus, we insert a transmit descriptor whenever the transmit queue becomes empty until identifying the login packet.

Figure 7.3 shows the format of the transmit descriptor structure [146]. Buffer Address points to the data in the host memory. The CMD (i.e., command) field specifies the RS

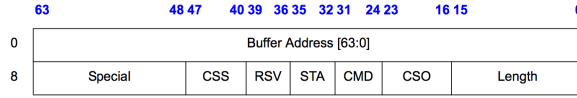


Figure 7.3: Transmit Descriptor Format

(i.e., report status) bit. With this bit set, the hardware writes a status bit back to the STA (i.e., status) field in the descriptor when a packet is transmitted. For the inserted packet, we set the RS bit and NULL to the Buffer Address so that it transfers no data. We also make sure all of the inspected packets have the RS bit set.

To replace the placeholders in the network packets, TrustLogin can simply search the sequence of the placeholders in the packets. We can use the Transmit Descriptor Base Address (TDBA) and Transmit Descriptor Tail (TDT) to find the addresses of the transmit descriptors. The transmit descriptor structure contains all of the information about the packet including the address of the payload. Note that the addresses here are physical addresses (i.e., no paging) because the Direct Memory Access (DMA) engine of the NIC only understands the physical addresses. One challenge of this method is that the network packets are encrypted (e.g., TLS). After the SMI handler finds and replaces the placeholders, it waits until the packet leaves the host to avoid further sensitive data leakage. Moreover, the attacker may use NIC's diagnostic registers to access transmitted packet connects. We empty the NIC's internal Packet Buffer Memory (PBM) by writing 16KB random data since the size of the internal buffer of our testing NIC is 16KB [146].

Ensuring Trusted Path

One challenge of TrustLogin is the reliability of triggering SMIs. As shown in Figure 7.2, the I/O redirection table in red is not trusted. An attacker with ring 0 privilege can modify the table to intercept an SMI, and then prepare a fake switching process so that the users think that he or she is in the SMM. In this case, the attacker can trick the user and get the password. This is a typical spoofing attack. There has been some research tackling this

problem [50–52, 65]. Bumpy [51] uses an external smartphone as the trusted monitor to acknowledge the switching. SecureSwitch [65] and Lockdown [50] use a dedicated switch to ensure the trust path. Cloud Terminal [52] uses a UI with strawberries on the screen as a shared secret to prevent spoofing attack. In TrustLogin, we implement two novel methods to prevent the spoofing attacks. One approach is to use the keyboard Light Emitting Diode (LED) lights, and the other is to use the PC speaker. Next, we explain the implementation details of these two approaches.

We use the LED lights on the keyboard to ensure the trust path. Usually, there are three LED lights on the keyboard, indicating Num, Caps, and Scroll locks. The users can set a shared secret LED light sequence to indicate that the system is in SMM. For instance, we can refer to scroll lock as 0, number lock as 1, and caps lock as 2. $\{[0 \text{ on}] \rightarrow [0 \text{ off}] \rightarrow [1 \text{ on}] \rightarrow [1 \text{ off}] \rightarrow [2 \text{ on}] \rightarrow [2 \text{ off}]\}$ is a LED light sequence. When the system switches into SMM, the SMI handler performs the shared secret LED light sequence so that the user knows the system is in SMM and not tricked by attackers.

To program the keyboard LED lights, we write a command byte, `0xED`, into the keyboard data register, and then write an LED state byte to the same I/O port. Bit 0 is for scroll lock; bit 1 is for number lock; bit 2 is for caps lock. Value 1 means on and 0 indicates off. Since every keystroke generates two interrupts (i.e., key press and release), TrustLogin only shows the LED light sequence when the key is released. We can easily identify a key release by checking the value of the scan code (greater than `0x80` [140]).

To help the user to identify the LED light sequence, we set a time delay between two lights. For instance, there should be a time delay between `[0 off]` and `[1 on]` for distinction. In TrustLogin, each light is on for 1 ms, and we set the same time delay when switching lights. The authors can identify that sequence based on their observations in the experiments. The user can adjust the time delay based on their preference.


```

    mov $0x30D40, %ecx    ;1 CPU cycle
DELAY:
    nop                   ;1 CPU cycle
    nop                   ;1 CPU cycle
    nop                   ;1 CPU cycle
    loop DELAY            ;8 CPU cycles

```

Listing 7.1: Assembly Code that Introduces 1 ms Delay on Our Testbed

Listing 7.1 shows the assembly code that introduces a 1 ms delay on our testbed. This delay function loads a counter, 0x30D40 or 200,000 in decimal, into **EAX**, and spinlocks until the counter is 0. The value, 200,000, is calculated from the time it takes to execute the loop instructions on our testbed. The testbed has an AMD Sempron LE-1250 2.2 GHz processor with AMD K8 chipset. The **MOV** and **NOP** instructions take 1 CPU cycle and the **LOOP** instruction takes 8 CPU cycles [147]. We also assume it takes 7 CPU cycles for a **LOOP** instruction when the contents of **EAX** is zero. The equations explain the steps that calculate the counter for performing 1 ms time delay on our testbed.

$$TimeDelay = \frac{ClockCycles}{ClockSpeed}$$

$$1ms = \frac{1 + N * (1 + 1 + 1) + (N - 1) * 8 + 7}{2.2GHz} \implies N = 200,000$$

We also use the PC speaker to ensure the trusted path. TrustLogin plays simple music on the PC speaker when each key is pressed in the secure input mode. The users can choose their favorite melodies and embed them in the SMI handler. By recognizing their selected tone sequence, they can ensure that an SMI is triggered for their every key press. Thus, the selected music should be short but recognizable. TrustLogin plays a C major scale in the SMI handler to demonstrate this idea. Table 7.1 shows the notes and corresponding frequencies for one complete octave starting from a middle C. We set the middle C as 523.25 Hz based on a musical reference guide [148].

Table 7.1: Musical Notes of an Octave

Musical Note	Frequency (Hz)	Divisor
C	523.25	0x08E2
D	587.33	0x07EA
E	659.26	0x070D
F	783.99	0x06A8
G	783.99	0x05EE
A	880.00	0x0548
B	987.77	0x04B5
C	1046.50	0x0471

To play a tone, we program the Intel 8253 Programmable Interval Timer (PIT) in the SMI handler to generate musical notes. The 8253 PIT performs timing and counting functions, and it exists in all x86 machines. In modern machines, it is included as part of the motherboard's southbridge. This timer has three counters (Counters 0, 1, and 2), and we use the third counter (Counter 2) to generate tones via the PC Speaker. We can generate different kinds of tones by adjusting the output frequency. The output frequency is calculated by loading a divisor into the 8253 PIT.

$$Divisor = IF/OF,$$

where IF is the input frequency of the 8253 PIT. IF used by the PIT chip is about 1.19 MHz, and OF is the output frequency. Column 3 of Table 7.1 shows the calculated divisors for the musical notes of an octave based on their output frequencies.

Playing a note on the PC speaker takes the following steps: 1) Configure mode/command register of the PIT chip through port 0x43 with value 0xB6, which selects channel 2 to use and sets the mode to accept divisors; 2) load a divisor into channel 2 through port 0x42; 3) turn on bit 0 and bit 1 of port 0x61 to enable the connection between PIT chip and the PC speaker; 4) set a time for the note to play; 5) turn off the PC speaker by configuring port 0x61. Similar to the LED lights sequence, we need to set a time delay so that the users can easily identify the music. In TrustLogin, each note is produced in 1 ms, and we set the same time of the delay between every two notes.

7.1.4 Case Study

We study legacy and secure applications to demonstrate the effectiveness of TrustLogin. For the legacy applications that we referenced, they are normally built on a client-server architecture, and authentication occurs using a plaintext username/password pair. We consider Remote Shell (rsh), File Transfer Protocol (FTP), and Telnet legacy applications. Note that TrustLogin is OS-agnostic for legacy applications because it does not need to reconstruct the semantics of OS kernels and rebuild the packet in the NIC. For the secure applications that we noted, their network traffic is securely encrypted. We consider Secure Shell (SSH), Secure File Transfer Protocol (SFTP), and Transport Layer Security (TLS) secure protocols.

Hardware and Software Specifications

We conduct the case study on a physical machine, which uses an ASUS M2V-MX_SE motherboard with an AMD K8 Northbridge and a VIA VT8237r Southbridge. It has a 2.2 GHz AMD LE-1250 CPU and 2GB Kingston DDR2 RAM. We use a Dell PS/2 keyboard and PCI-based Intel 82541 Gigabit Ethernet Controller as the triggering devices. To program SMM, we use the open-source BIOS, Coreboot [20]. We also install Microsoft Windows 7 and CentOS 5.5 on this machine.

Case Study I: Legacy Applications

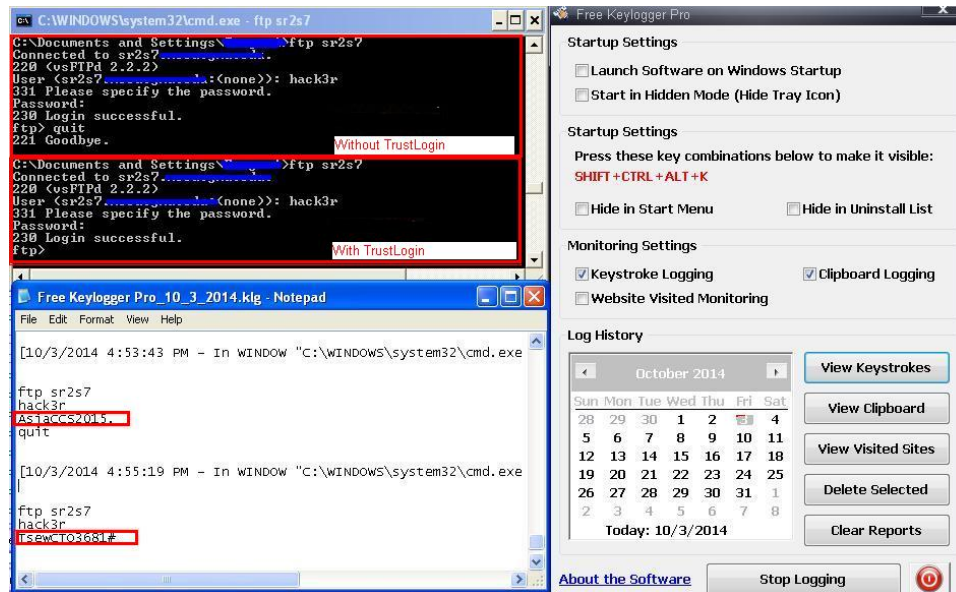
For legacy applications, we use FTP as the study example. Next, we demonstrate the effectiveness of our system on both Windows and Linux platforms. Figure 7.4 shows the screenshots of the FTP login with and without TrustLogin on Windows and Linux. We create an FTP account on a server. The username and password for the account is **hack3r** and **AsiaCCS.**, respectively. On Windows, we install the Free Keylogger Pro version 1.0 [149] and use the FTP client to connect to the server. We first start the keylogger to log keystrokes. Next, we login to the FTP server without TrustLogin enabled. As shown in subfigure 7.4(a), we can see that the keylogger records the timestamp, application name, username, and

password in a file. The red rectangle shows the password is **AsiaCCS**. as recorded by the keylogger. Then, we enable TrustLogin and login to the FTP server again. However, the password recorded by the keylogger has been changed to a random string generated by TrustLogin. In other words, the keylogger cannot steal the real password when TrustLogin is enabled. We install Logkeys version 0.1.1a [150] on the CentOS 5.5, and subfigure 7.4(b) shows the results. Similar to the experiments on Windows, we login to the FTP server with and without TrustLogin enabled. We can see that the keylogger logs the random placeholders when TrustLogin enabled, and the keylogger cannot steal the login password. Note that attackers can easily steal the passwords from legacy applications by sniffing out the network. However, TrustLogin is used to address the general problem of securing keystrokes on the local host. Here studying the legacy applications emphasizes that TrustLogin is a framework that works with various applications to prevent keyloggers.

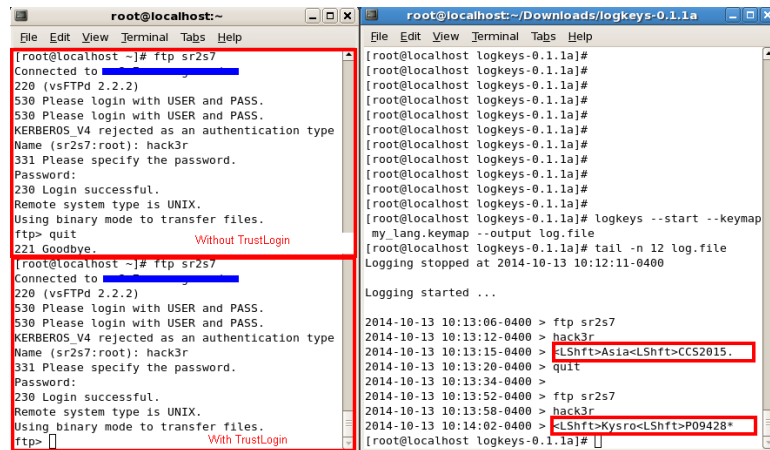
Case Study II: Secure Applications

TrustLogin replaces the password placeholders in a network packet when the packet is about to transmit. However, since most of the network packets are encrypted, we cannot simply search for the placeholder sequence though the encrypted data. For example, Transport Layer Security (TLS) encrypts the data of network connections in the application layer.

Secure Shell (SSH) is one of the most popular application protocols for access to shell accounts on Unix-based systems. There are several client authentication methods supported by SSH. For instance, we can use a public and private key pair to authenticate the client. However, we only consider the password-based authentication method in this paper. The password-based authentication method is the most commonly used authentication mechanisms in SSH. The SSH server simply uses a username and password to authenticate the client, and the password transmitted by the client to the server is encrypted by a session symmetric key. Unlike the legacy applications, we cannot simply search for the login packet and replace the original password. To address this problem, TrustLogin decrypts and re-encrypts the login packets using the session symmetric key. Next, we explain how we find



(a) FTP Login With and Without TrustLogin on Windows



(b) FTP Login With and Without TrustLogin on Linux

Figure 7.4: FTP Login With and Without TrustLogin on Windows and Linux

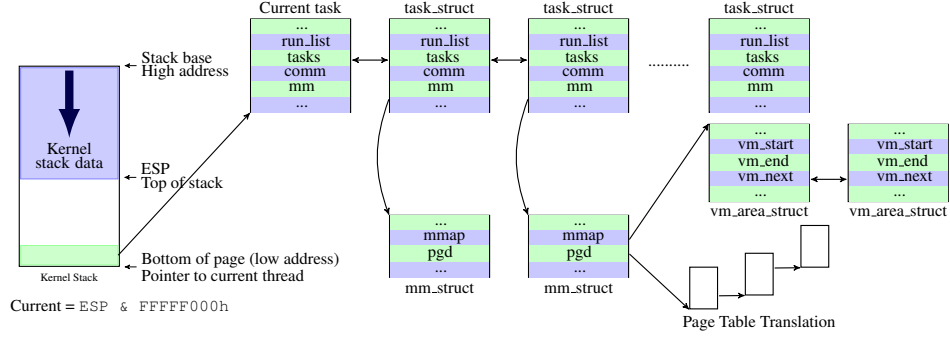


Figure 7.5: Filling the Semantic Gap by Using Kernel Data Structures in Linux

the session states like the session key.

As mentioned, SSH protocol uses a session symmetric key to encrypt the traffic. To decrypt the login packet, we first need to find the session states in the memory. Fortunately, SMM is able to access all of the physical memory because it has the highest privilege. TrustLogin uses signature-based searching. Typically, software data structures inevitably create some signatures in memory. For example, researchers extract SSL private key in memory by validating RSA/DSA structures on multiple applications including Apache, SSH, and OpenVPN [151].

We use `session_state` structure as the signature for the searching. The `session_state` structure stores the session information for SSH communication. By analyzing the source code of OpenSSH [152], some fields of `session_state` structure in the `packet.c` file are static before users authenticate themselves. For instance, the `max_packet_size` field is set to constant, 0x8000. Listing 7.2 shows part of `session_state` structure, and these fields are static and continuous. Thus, TrustLogin uses this static signature as the search string and then finds the session states in memory. Since we assume malware has ring 0 privilege, an advanced malware can prepare a fake session key for TrustLogin. This attack breaks the login process but cannot steal the real password. Note that a DoS attack is out of the scope of this paper.

```

struct session_state {
    /* default maximum packet size, 0x8000 */
    u_int max_packet_size;
    /* Flag indicating whether this module has been initialized, 0x1 */
    int initialized;
    /* Set to true if connection is interactive, 0x0 */
    int interactive_mode;
    /* Set to true if we are the server side, 0x0 */
    int server_side;
    /* Set to true if we are authenticated, 0x0 */
    int after_authentication;
    /* Default before login, 0x0 */
    int keep_alive_timeouts;
    /* The maximum time that we will wait to send or receive a packet, -1, 0xffffffff
       */
    int packet_timeout_ms;
}

```

Listing 7.2: Static Fields of Session Structure in SSH Source Code

One naive approach for implementation would be searching byte-by-byte through all of physical memory, but this is very slow. We instead search only the SSH instance in Trust-Login. Since only the physical memory is visible to SMM, we must reconstruct semantics of the raw data. In other words, we must understand the location of the SSH process and what its content means in memory. This is referred to as the “semantic gap problem.” Recently, researchers proposed an array of approaches to address this problem [35], including hand-crafted data structure signatures [32,66] and automated learning and bridging [33,34]. Similar to the previous systems [32,66], we manually reconstruct the semantics using kernel data structure signatures.

We first use the ESP value saved in the SMRAM to calculate the pointer to the current process’s `task_struct`. Alternatively, we can obtain the address of the `init_task` from the `System.map` file. Next, we traverse the doubly linked list of `task_structs` or `run_lists` to

locate the SSH process by comparing the `comm` field. Note that multiple instances of SSH could be running at the same time in the memory. We use the `prev` field for the transversal, which ensures that the first SSH process found is the last process launched. In this case, we assume the user interacts with the most recently launched SSH instance. Next, we obtain a pointer to the `mm_struct` from the `mm` field in `task_struct`. The `mmap` field in the `mm_struct` points to the head of the list of memory regions with the type `vm_area_struct`. The memory region object contains `vm_start` and `vm_end` fields, which define the start and end addresses of the memory region. Figure 7.5 shows the semantic reconstruction using kernel data structures in Linux. As pointed out in [35], all of the current solutions to the semantic gap assumes the kernel data structures are benign. Our semantic reconstruction approach also assumes this. We search the `session.state` signature in the memory regions of the SSH process, which achieves a better performance than the linear searching approach. Section 7.1.4 details the overheads of these two approaches.

All pointers in these structures are virtual. However, SMM does not use paging, meaning it addresses physical memory directly. Thus, we must translate addresses manually from virtual to physical space. For kernel-space structures (e.g., `task_struct` and `mm_struct`), there is a constant offset, `0xc0000000`, to move from virtual to physical space. For userspace structures (e.g., `vm_start` and `vm_end`), we locate and employ the process’s page tables. Fortunately, the `pgd` field in the `mm_struct` stores the `cr3` value that tells us the location of the global page directory. After we retrieve all of the required information from memory, we decrypt the data and replace the placeholder sequence with the real password in the packet. Finally, we rebuild the network packet with the corresponding checksum by using the functions from OpenSSH source code.

Performance Evaluation

In order to understand the performance overhead of our system, we measure the runtime of each individual operation in the SMI handler. In TrustLogin, we have two parts of the handling code in the SMI handler; one is to handle the SMI triggered by the keyboard (i.e.,

Table 7.2: Breakdown of the SMI Handler Runtime (Time: μs)

	Operations	Mean	STD	95% CI
KB SMI	Play music	26,244	3,675	[25,199,27,288]
	Show LED	6,317	251	[6,245,6,388]
	Disable KB SMI	1.47	0.21	[1.40,1.53]
	Read keystroke	2.38	0.33	[2.28,2.47]
	Enable NIC SMI	8.40	0.05	[8.39,8.419]
	Replace keystroke	8.94	1.27	[8.57,9.30]
	Enable KB SMI	1.14	0.17	[1.09,1.19]
	Total of KB SMI	32,583		
NIC SMI	Read NIC registers	3.96	0.10	[3.93,3.99]
	Search packets	18.27	1.18	[17.93,18.60]
	Disable NIC SMI	7.44	0.05	[7.42,7.45]
	Total of NIC SMI	29.67		
	Switch into SMM	3.29	0.08	[3.27,3.32]
	Resume from SMM	4.58	0.10	[4.55,4.61]
	Total of switching	7.87		

KB SMI), and the other part is executed when the NIC triggers an SMI (i.e., NIC SMI).

The KB SMI contains 7 steps.

1. Play a melody
2. Show an LED sequence
3. Disable keyboard SMIs (prevent reissuing)
4. Read the keystroke from the keyboard data register and save it in SMRAM
5. If Enter is pressed, break out and enable NIC SMIs
6. Generate random scan code and replace the keystroke
7. Enable keyboard SMIs for subsequent keystroke

Additionally, the NIC SMI consists of 3 operations.

1. Locate received packet in NIC memory
2. Search packets and inject original password
3. Disable NIC SMIs after password is injected

We measure the time delay for all of these operations in the SMI handler. The hardware automatically saves and resumes the CPU context when switching to SMM. We also measure

Table 7.3: Comparison between Linear Searching and Semantic Searching

Approaches	Search Space	Time
Linear Searching	2 GB	70.21 s
Semantic Searching	18 MB	(1.39+227.45) ms

the overhead induced by switching to and resuming from SMM.

Table 7.2 shows the time breakdown of each operation. We use the Time Stamp Counter (TSC) to measure the time delay for each operation. We first record the TSC value at the beginning and end of each operation took. Next, we use the CPU frequency to divide the difference in the TSC register to compute how much time this operation. We conduct the experiment based on the FTP login for 30 trials. We calculate the mean, standard deviation, and 95% confidence interval for each operation. From Table 6.5 we can see that the total time for KB SMIs is about 32 ms. Note that most of the time is consumed by playing the melody and showing the LED sequence. Each note in the melody and each part of the LED sequence takes 1 ms (See Section 7.1.3 for details). The total time of the NIC SMI code and SMM switching are only about 30 μs and 8 μs , respectively.

Searching the SSH session states after reconstructing the memory semantic (called semantic searching) achieves a better performance than linear searching. To demonstrate this, we compared the linear searching with the semantic searching. We conducted the experiment on the Linux machine. We installed the latest OpenSSH client version 6.6p1 on the testbed, and the testing machine has 2 GB physical memory. For linear searching, we compare the signature of the session states with the 2 GB memory byte-by-byte. As for semantic searching, we first find the SSH process in memory using kernel data structures and then only search the memory regions pointed by `mmap`. After a user types `ssh username@hostname` in a terminal, the SSH server waits for the password from the client. At this time, we trigger an SMI and let the SMI handler perform both searching methods. We also use the TSC to measure the time it takes for each approach.

Table 7.3 shows the comparison between linear searching and semantic searching. The linear searching has 2 GB of searching space and takes about 70 seconds to find the session states. The semantic searching only has about 18 MB of searching space; it takes 1.39 ms to fill the semantic gap and 227 ms for searching.

Chapter 8: Conclusions and Future Work

8.1 Conclusions

This thesis used hardware-level isolated execution environments for securing data and computer systems. System Management Mode (SMM) is a special CPU mode in the x86 architecture. SMM utilizes an isolated region of memory that is inaccessible from any other CPU modes. This caveat therefore allows SMM to be used as an isolated execution environment. To validate my research approach, I used SMM as an isolated execution environment to build defensive tools [66, 75, 106, 124, 137, 153] for protecting the computer systems. Note that the tools running in SMM are OS-agnostic since SMM is a hardware feature. Additionally, I co-developed a BIOS-assisted isolation environment that is capable of running a secure commodity OS [65].

8.1.1 System Introspection for Malware Detection

In my dissertation, I used SMM as an isolated execution environment, to introspect all layers of system software for malware detection with a minimal TCB. I developed three novel tools in SMM, which targets different layers of the system software for attack detection.

Firmware-level: I designed and implemented IOCheck [106, 153], a tool to quickly check the integrity of I/O configurations and firmware at runtime. I demonstrated the effectiveness of IOCheck by checking the integrity of a network card and video card, and the experiments showed that it can detect configuration and firmware attacks against the I/O devices.

Hypervisor- and OS-level: HyperCheck [75] is a hardware-assisted tampering detection system, that aims to protect the static code integrity of hypervisors and kernels running on commodity hardware. The experimental results show that HyperCheck operation is

lightweight, and it can complete one round of integrity checking of Xen hypervisor code and CPU register states in less than 90 milliseconds.

OS- and Application-level: I also designed and implemented Spectre [66], a dependable framework that inspects the memory of a live system. Besides checking static code, Spectre analyzes dynamic code and data of applications. It can detect memory attacks including heap spray, heap overflow, and rootkits on both Windows and Linux platforms with low overhead.

8.1.2 Transparent Malware Debugging

In my dissertation, I designed MalT [124], a novel approach that progresses towards stealthy debugging by leveraging SMM to transparently debug software on bare-metal. The system is motivated by the intuition that malware debugging needs to be transparent, and it should not leave artifacts introduced by the debugging functions. The main benefit of SMM is to provide a distinct and easily isolated processor environment that is transparent to the operating system and running applications. With the help of SMM, MalT is able to achieve a high level of transparency, which enables a strong threat model for malware debugging. Because we run debugging code in SMM, MalT exposes far fewer artifacts to the malware, enabling a more transparent execution environment for the debugging code than existing approaches.

I implemented a prototype of MalT on two physical machines connected by a serial cable. To demonstrate the efficiency and transparency of the approach, I tested MALT with popular packing, anti-debugging, anti-virtualization, and anti-emulation techniques. The experimental results show that MalT remains transparent against these techniques. Additionally, the experiments demonstrate that MalT is able to debug crashed kernels/hypervisors. MalT introduces a reasonable overhead: It takes about 12 microseconds on average to execute the debugging functions. Moreover, the prototype of MALT introduces moderate but manageable overheads on both Windows and Linux platforms.

8.1.3 Executing Sensitive Workloads

I designed and implemented TrustLogin [137], a system to securely perform login operations on commodity operating systems. Even if the operating system and applications are compromised, an attacker is not able to reveal the login password from the host. TrustLogin leverages SMM to transparently protect the login credentials from keyloggers. TrustLogin does not modify application- and OS-code, and it is transparent from both end-users and servers. The prototype of TrustLogin was implemented on legacy and secure applications. The experimental results showed that TrustLogin can prevent real-world keyloggers from stealing passwords on Windows and Linux platforms. TrustLogin is robust against spoofing attacks by ensuring the trusted path of SMM switching. The performance experiments show that TrustLogin is light-weight and efficient to use.

Additionally, I co-developed SecureSwitch [65], a BIOS-assisted mechanism to enable secure instantiation and management of isolated computing environments, tailored to separate security-sensitive activities from untrusted ones on the x86 architecture. A key design characteristic of this system is usability—the ability to quickly and securely switch between operating environments without requiring any specialized hardware or code modifications. SecureSwitch loads two OSes into the RAM at the same time and uses the ACPI S3 sleep mode to control switching between the two OSes. The prototype of the secure switching system used commodity hardware and both commercial and open source OSes (Microsoft Windows and Linux). It can switch between OSes in approximately six seconds.

8.2 Future Work

Commodity systems and software are complex, and the problem of security of computer systems is far from solved. The systems I developed—Malt [124], TrustLogin [137], IOCheck [106, 153], HyperCheck [75], Spectre [66]—only represent a few contributions to this field of research. However, the bulk of this work still lies ahead. In my future research, I will pursue three areas.

First, I will continue developing current defensive tools in hardware-level isolated execution environments. For instance, the prototype of TrustLogin [137] only protects the password-login of simple applications like SSH. I plan to extend TrustLogin to secure banking logins and transactions in popular browsers. Additionally, I plan to mitigate phishing attacks against TrustLogin by validating the destination host in SMM. IOCheck [106, 153] and HyperCheck [75] only verify the integrity of the static code of the firmware and hypervisor. Building upon this, I will check dynamic data of firmware and hypervisors for malware detection.

Second, I will investigate other hardware-level isolated execution environments for trustworthy computing on x86 architecture. Trust Computing Group introduced Dynamic Root of Trust for Measurement (DRTM). To implement this technology, Intel developed Trusted eXecution Technology (TXT), providing a trusted way to load and execute system software (e.g., OS or VMM). TXT uses a new CPU instruction, SENTER, to control the secure environment. AMD has a similar technology called Secure Virtual Machine, and it uses the SKINIT instruction to enter the secure environment. Last year, Intel introduced Software Guard Extensions (SGX), a set of instructions and mechanisms for memory accesses added to future Intel architecture processors. SGX is an up-and-coming technology. These extensions allow an application to instantiate a protected container, referred to as an enclave. An enclave could be used as an isolated execution environment, which provides confidentiality and integrity even in the presence of privileged malware.

Intel TXT, AMD SVM, and SGX provide isolated execution environments that execute code on the main CPU (i.e., processor). However, modern x86-based platforms always have micro-processors that help the main x86 processor (e.g., Intel Management Engine or AMD System Management Unit). These micro-processors have their own registers and memory, which provide isolated execution environments on a separate processor (we often refer to the kernel as having ring 0 privilege, hypervisor as ring -1, SMM as ring -2, and Intel ME/AMD SMU as ring -3). On one hand, we can create stealthy malware in these execution environments on separate processors from the main CPU. On the other hand,

we can build defensive tools to secure the main memory and code executed on the main processor.

Third, I will study the trustworthy execution environments on the ARM architecture. As smartphones and mobile computing become increasingly ubiquitous, trustworthy execution environments play a critical role for running security-sensitive workloads and applications. The ARM community introduced TrustZone technology, which enables a full trusted execution environment. I plan to study and use TrustZone to build defensive tools for securing mobile systems such as Android. Moreover, ARM is a child compared to the age of x86, I believe there will be more hardware-level isolated execution environments supported on ARM in the near future (e.g., SGX on ARM) by observing the evolution of the isolated execution environments on the x86 architecture. I believe this research direction will have impact due to the proliferation of mobile computing.

Bibliography

Bibliography

- [1] McAfee, “Threats Report: First Quarter 2014,” <http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q1-2014-summary.pdf>.
- [2] Kaspersky Lab, “Kaspersky Security Bulletin 2014,” <http://cdn.securelist.com/files/2014/12/Kaspersky-Security-Bulletin-2014-Predictions-2015.pdf>.
- [3] Symantec, “Internet Security Threat Report, Vol. 19 Main Report,” http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v19_21291018.en-us.pdf, 2014.
- [4] T. Garfinkel and M. Rosenblum, “A Virtual Machine Introspection Based Architecture for Intrusion Detection,” in *Proceedings of the 10th Annual Network and Distributed Systems Security Symposium (NDSS’03)*, 2003.
- [5] National Institute of Standards, NIST, “National Vulnerability Database,” <http://nvd.nist.gov>, access time: 2014.03.04.
- [6] K. Kortchinsky, “CLOUDBURST: A VMware Guest to Host Escape Story,” in *Black Hat USA*, 2009.
- [7] R. Wojtczuk, J. Rutkowska, and A. Tereshkin, “Xen Owing Trilogy,” in *Black Hat USA*, 2008.
- [8] S. T. King and P. M. Chen, “SubVirt: Implementing Malware with Virtual Machines,” in *Proceedings of the 27th IEEE Symposium on Security and Privacy (S&P’06)*, May 2006.
- [9] L. Dufлот and Y. A. Perez, “Can You Still Trust Your Network Card?” in *Proceedings of the 13th CanSecWest Conference (CanSecWest’10)*, 2010.
- [10] K. Chen, “Reversing and Exploiting an Apple Firmware Update,” *Black Hat*, 2009. [Online]. Available: <http://www.blackhat.com/presentations/bh-usa-09/CHEN/BHUSA09-Chen-RevAppleFirm-PAPER.pdf>
- [11] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario, “Towards an Understanding of Anti-Virtualization and Anti-Debugging Behavior in Modern Malware,” in *Proceedings of the 38th Annual IEEE International Conference on Dependable Systems and Networks (DSN ’08)*, 2008.
- [12] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin, “Compatibility is not Transparency: VMM Detection Myths and Realities,” in *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems (HotOS’07)*, 2007.

- [13] N. Falliere, “Windows Anti-Debug Reference,” <http://www.symantec.com/connect/articles/windows-anti-debug-reference>, 2010.
- [14] E. Bachaalany, “Detect If Your Program is Running inside a Virtual Machine,” <http://www.codeproject.com/Articles/9823/Detect-if-your-program-is-running-inside-a-Virtual>.
- [15] D. Quist and V. Val Smith, “Detecting the Presence of Virtual Machines Using the Local Data Table,” <http://www.offensivecomputing.net>.
- [16] T. Raffetseder, C. Kruegel, and E. Kirda, “Detecting System Emulators,” in *Information Security*. Springer Berlin Heidelberg, 2007.
- [17] J. Xiao, Z. Xu, H. Huang, and H. Wang, “Security Implications of Memory Deduplication in a Virtualized Environment,” in *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN’13)*, 2013.
- [18] J. Rutkowska, “Red Pill,” http://www.ouah.org/Red_Pill.html.
- [19] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki, “Flicker: An Execution Infrastructure for TCB Minimization,” in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2008.
- [20] Coreboot, “Open-Source BIOS,” <http://www.coreboot.org/>.
- [21] Y. Ding, T. Wei, T. Wang, Z. Liang, and W. Zou, “Heap Taichi: Exploiting memory allocation granularity in heap-spraying attacks,” in *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC ’10)*, 2010.
- [22] A. Sotirov, “Heap Feng Shui in JavaScript,” in *In Black Hat Europe*, 2007.
- [23] S. Designer, “JPEG COM marker processing vulnerability,” July 2000. [Online]. Available: <http://www.openwall.com/articles/JPEG-COM-Marker-Vulnerability>
- [24] Multi-State Information Sharing and Analysis Center., “Vulnerability in Adobe Reader and Adobe Acrobat could allow remote code execution.” [Online]. Available: <http://www.msisac.org/advisories/2009/2009-008.cfm>
- [25] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda, “Defending browsers against drive-by-downloads: Mitigating heap-spraying code injection attacks,” in *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA’09)*, 2009.
- [26] J. Vanegue, “Zero-sized heap allocations vulnerability analysis,” in *In Proceedings of the 2nd Conference on USENIX Workshop on Offensive Technologies (WOOT’10)*, 2010.
- [27] G. Novark and E. Berger, “DieHarder: Securing the Heap,” in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS’10)*, 2010.
- [28] P. Ratanaworabhan, B. Livshits, and B. Zorn, “Nozzle: A defense against heap-spraying code injection attacks,” in *Proceedings of the 18th USENIX Security Symposium*, 2009.

- [29] M. Cova, C. Kruegel, and G. Vigna, “Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code,” in *Proceedings of the 19th International World Wide Web Conference (WWW’10)*, 2010.
- [30] A. Seshadri, M. Luk, N. Qu, and A. Perrig, “SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes,” in *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP’07)*, 2007.
- [31] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, “TrustVisor: Efficient TCB reduction and attestation,” in *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.
- [32] X. Jiang, X. Wang, and D. Xu, “Stealthy Malware Detection Through VMM-based Out-of-the-box Semantic View Reconstruction,” in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS’07)*, 2007.
- [33] T. Leek, M. Zhivich, J. Giffin, and W. Lee, “Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection,” in *Proceedings of the 32nd IEEE Symposium on Security and Privacy (S&P’11)*, 2011.
- [34] Y. Fu and Z. Lin, “Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection,” in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P’12)*, 2012.
- [35] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion, “SoK: Introspections on Trust and the Semantic Gap,” in *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P’14)*, 2014.
- [36] A. Vasudevan and R. Yerraballi, “Stealth Breakpoints,” in *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC’05)*, 2005.
- [37] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, “Ether: Malware Analysis via Hardware Virtualization Extensions,” in *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS ’08)*, 2008.
- [38] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, “BitBlaze: A New Approach to Computer Security via Binary Analysis,” in *Proceedings of the 4th International Conference on Information Systems Security (ICISS’08)*, 2008.
- [39] Anubis, “Analyzing Unknown Binaries,” <http://anubis.iseclab.org>.
- [40] L.-K. Yan, M. Jayachandra, M. Zhang, and H. Yin, “V2E: Combining Hardware Virtualization and Software Emulation for Transparent and Extensible Malware Analysis,” in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE’12)*, 2012. [Online]. Available: <http://doi.acm.org/10.1145/2151024.2151053>
- [41] Z. Deng, X. Zhang, and D. Xu, “SPIDER: Stealthy Binary Program Instrumentation and Debugging Via Hardware Virtualization,” in *Proceedings of the Annual Computer Security Applications Conference (ACSAC’13)*, 2013.

- [42] D. Kirat, G. Vigna, and C. Kruegel, “BareBox: Efficient Malware Analysis on Bare-metal,” in *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC’11)*, 2011.
- [43] C. Willems, R. Hund, A. Fobian, D. Felsch, T. Holz, and A. Vasudevan, “Down to the Bare Metal: Using Processor Features for Binary Analysis,” in *Proceedings of the Annual Computer Security Applications Conference (ACSAC’12)*, 2012.
- [44] D. Kirat, G. Vigna, and C. Kruegel, “BareCloud: Bare-metal Analysis-based Evasive Malware Detection,” in *Proceedings of the 23rd USENIX Security Symposium*, 2014.
- [45] N. A. Quynh and K. Suzaki, “Virt-ICE: Next-generation Debugger for Malware Analysis,” in *Black Hat USA*, 2010.
- [46] IDA Pro, www.hex-rays.com/products/ida/.
- [47] OllyDbg, www.ollydbg.de.
- [48] D. Bruening, Q. Zhao, and S. Amarasinghe, “Transparent Dynamic Instrumentation,” in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE’12)*, 2012.
- [49] Windbg, www.windbg.org.
- [50] A. Vasudevan, B. Parno, N. Qu, V. Gligor, and A. Perrig, “Lockdown: A Safe and Practical Environment for Security Applications (CMU-CyLab-09-011),” Tech. Rep., 2009.
- [51] J. M. McCune, A. Perrig, and M. K. Reiter, “Safe passage for passwords and other sensitive data,” in *NDSS*, 2009.
- [52] L. Martignoni, P. Poosankam, M. Zaharia, J. Han, S. McCamant, D. Song, V. Paxson, A. Perrig, S. Shenker, and I. Stoica, “Cloud Terminal: Secure Access to Sensitive Applications from Untrusted Systems,” in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC’12)*, 2012.
- [53] K. Borders and A. Prakash, “Securing network input via a trusted input proxy,” in *Proceedings of the 2nd USENIX workshop on Hot topics in security*, 2007.
- [54] L. Dufлот, D. Etiemble, and O. Grumelard, “Using CPU System Management Mode to Circumvent Operating System Security Functions,” in *Proceedings of the 7th CanSecWest Conference (CanSecWest’04)*, 2004.
- [55] S. Embleton, S. Sparks, and C. Zou, “SMM rootkits: A New Breed of OS Independent Malware,” in *Proceedings of the 4th International Conference on Security and Privacy in Communication Networks (SecureComm’08)*, 2008.
- [56] BSDaemon, coideloko, and D0nAnd0n, “System Management Mode Hack: Using SMM for ‘Other Purposes’,” *Phrack Magazine*, 2008.
- [57] J. Rutkowska and R. Wojtczuk, “Preventing and Detecting Xen Hypervisor Subversions,” 2008.

- [58] R. Wojtczuk and J. Rutkowska, “Attacking SMM Memory via Intel CPU Cache Poisoning,” 2009. [Online]. Available: http://invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf
- [59] L. Dufлот, O. Levillain, B. Morin, and O. Grumelard, “Getting into the SM-RAM: SMM Reloaded,” in *Proceedings of the 12th CanSecWest Conference (CanSecWest’09)*, 2009.
- [60] —, “System Management Mode Design and Security Issues,” http://www.ssi.gouv.fr/IMG/pdf/IT_Defense.2010_final.pdf.
- [61] R. Wojtczuk and A. Tereshkin, “Attacking Intel BIOS,” <https://www.blackhat.com/presentations/bh-usa-09/WOJTCZUK/BHUSA09-Wojtczuk-AtkIntelBios-SLIDES.pdf>.
- [62] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky, “HyperSentry: Enabling Stealthy In-Context Measurement of Hypervisor Integrity,” in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS’10)*, 2010.
- [63] Intel, “64 and IA-32 Architectures Software Developer’s Manual.” [Online]. Available: <http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>
- [64] R. Wojtczuk and C. Kallenberg, “Attacking UEFI Boot Script,” 31st Chaos Communication Congress (31C3), <http://events.ccc.de/congress/2014/Fahrplan/system/attachments/2566/original/venamis-whitepaper.pdf>, 2014.
- [65] K. Sun, J. Wang, F. Zhang, and A. Stavrou, “SecureSwitch: BIOS-Assisted Isolation and Switch between Trusted and Untrusted Commodity OSes,” in *Proceedings of the 19th Annual Network & Distributed System Security Symposium (NDSS’12)*, 2012.
- [66] F. Zhang, K. Leach, K. Sun, and A. Stavrou, “SPECTRE: A Dependable Inspection Framework via System Management Mode,” in *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN’13)*, 2013.
- [67] Trusted Computing Group, “TPM Main Specification Level 2 Version 1.2, Revision 116,” http://www.trustedcomputinggroup.org/resources/tpm_main_specification, 2011.
- [68] VIA, “VT8237R Southbridge,” <http://www.via.com.tw/>.
- [69] E. Barbosa, “Finding some non-exported kernel variables in Windows XP,” <http://www.reverse-engineering.info/SystemInformation/GetVarXP.pdf>.
- [70] T. Kong. (2004, May) KProcCheck by SIG^2: Win2K kernel hidden process/module checker. [Online]. Available: <http://www.security.org.sg/code/kproccheck.html>
- [71] “Fu rootkit,” <http://www.f-secure.com/v-descs/fu.shtml>, 2006.

- [72] “KBeast rootkit,” <http://packetstormsecurity.org/files/108286/KBeast-Kernel-Beast-Linux-Rootkit-2012.html>, 2012.
- [73] Chkrootkit, “chkrootkit rootkit detector.” [Online]. Available: <http://www.chkrootkit.org/>
- [74] Rkhunter, “rkhunter rootkit detector.” [Online]. Available: <http://www.rootkit.nl/projects/rootkit-hunter.html/>
- [75] F. Zhang, J. Wang, K. Sun, and A. Stavrou, “HyperCheck: A Hardware-assisted Integrity Monitor,” in *IEEE Transactions on Dependable and Secure Computing (TDSC’14)*, 2014.
- [76] R. Wojtczuk and J. Rutkowska, “Following the White Rabbit: Software Attacks against Intel VT-d,” 2011. [Online]. Available: <http://invisiblethingslab.com/itl/Resources.html>
- [77] D. Chisnall, *The definitive guide to the Xen hypervisor*. Prentice Hall Press Upper Saddle River, NJ, USA, 2007.
- [78] MITRE, “Cve-2007-4993.” [Online]. Available: <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-4993>
- [79] R. Wojtczuk, “Subverting the Xen hypervisor,” 2008. [Online]. Available: <http://invisiblethingslab.com/resources/misc08/xenfb-adventures-10.pdf>
- [80] K. Adamyse, “Handling interrupt descriptor table for fun and profit. Phrack 59,” 2002.
- [81] B. Prochzka, T. Vojnar, and M. Drahansk, “Hijacking the linux kernel,” <http://drops.dagstuhl.de/opus/volltexte/2011/3063/pdf/7.pdf>, 2011. [Online]. Available: <http://drops.dagstuhl.de/opus/volltexte/2011/3063/pdf/7.pdf>
- [82] A. Regenscheid and K. Scarfone, “BIOS Integrity Measurement Guidelines (Draft), NIST-800-155,” <http://csrc.nist.gov/publications/drafts/800-155/draft-SP800-155-Dec2011.pdf>, December 2011.
- [83] “TCG PC Client Specific Implementation Specification for Conventional BIOS,” http://www.trustedcomputinggroup.org/files/resource_files/CB0B2BFA-1A4B-B294-D0C3B9075B5AFF17/TCG_PCCClientImplementation_1-21_1.00.pdf, February 2012.
- [84] Michael Howard and Matt Miller and John Lambert and Matt Thomlinson, “Windows ISV Software Security Defenses,” December 2010.
- [85] PaX Team., “<http://pax.grsecurity.net/>.” [Online]. Available: <http://pax.grsecurity.net/>
- [86] Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba, “ACPI,” <http://www.acpi.info/>.
- [87] S. Devik, “Linux on-the-fly kernel patching without LKM,” *Phrack Magazine*, 2001.

- [88] S. Zhang, L. Wang, R. Zhang, and Q. Guo, “Exploratory study on memory analysis of Windows 7 operating system,” in *Advanced Computer Theory and Engineering (ICACTE), 2010 3rd International Conference on*, vol. 6, 2010, pp. V6–373–V6–377.
- [89] S. Schreiber, *Undocumented Windows 2000 secrets: A programmer’s cookbook*. Addison-Wesley, 2001.
- [90] D. Bovet and M. Cesati, *Understanding the Linux kernel*, 3rd ed. O’Reilly Media, 2005.
- [91] J. Wang, K. Sun, and A. Stavrou, “A Dependability Analysis of Hardware-Assisted Polling Integrity Checking Systems,” in *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN’12)*, 2012.
- [92] “SLOCCount.” [Online]. Available: <http://www.dwheeler.com/sloccount/>
- [93] Unixbench. [Online]. Available: <http://code.google.com/p/byte-unixbench/>
- [94] Mitre, “Vulnerability list,” <http://cve.mitre.org/cve/cve.html>.
- [95] A. J. Bonkoski, R. Bielawski, and J. A. Halderman, “Illuminating the Security Issues Surrounding Lights-out Server Management,” in *Proceedings of the 7th USENIX Conference on Offensive Technologies (WOOT’13)*, 2013.
- [96] P. Stewin and I. Bystrov, “Understanding DMA Malware,” in *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA’12)*, 2012.
- [97] D. Aumaitre and C. Devine, “Subverting Windows 7 x64 Kernel With DMA Attacks,” in *HITBSecConf Amsterdam*, 2010. [Online]. Available: <http://esec-lab.sogeti.com/dotclear/public/publications/10-hitbamsterdam-dmaattacks.pdf>
- [98] A. Triulzi, “Project Maux Mk.II,” in *CanSecWest*, 2008.
- [99] F. Sang, V. Nicomette, and Y. Deswarte, “I/O Attacks in Intel PC-based Architectures and Countermeasures,” in *SysSec Workshop (SysSec’11)*, 2011.
- [100] P. Stewin, “A Primitive for Revealing Stealthy Peripheral-Based Attacks on the Computing Platforms Main Memory,” in *Research in Attacks, Intrusions, and Defenses (RAID’13)*, 2013.
- [101] F. Sang, E. Lacombe, V. Nicomette, and Y. Deswarte, “Exploiting an I/OMMU vulnerability,” in *5th International Conference on Malicious and Unwanted Software (MALWARE’10)*, 2010, pp. 7–14.
- [102] R. Wojtczuk and J. Rutkowska, “Another Way to Circumvent Intel Trusted Execution Technology,” <http://invisiblethingslab.com/resources>, 2009.
- [103] Trusted Computing Group, “TCG PC Client Specific Implementation Specification for Conventional BIOS, Specification Version 1.21,” <http://www.trustedcomputinggroup.org>, February 2012.
- [104] —, “TCG D-RTM Architecture Document Version 1.0.0,” http://www.trustedcomputinggroup.org/resources/drtm_architecture_specification, June 2013.

- [105] Intel, “Trusted Execution Technology,” <http://www.intel.com/content/www/us/en/trusted-execution-technology/trusted-execution-technology-security-paper.html>.
- [106] F. Zhang, H. Wang, K. Leach, and A. Stavrou, “A Framework to Secure Peripherals at Runtime,” in *Proceedings of The 19th European Symposium on Research in Computer Security (ESORICS’14)*, 2014.
- [107] L. Dufлот, Y.-A. Perez, and B. Morin, “What If You Can’t Trust Your Network Card?” in *Recent Advances in Intrusion Detection (RIAD’11)*, 2011.
- [108] Y. Li, J. McCune, and A. Perrig, “VIPER: Verifying the Integrity of PERipherals’ Firmware,” in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS’11)*, 2011.
- [109] H. Moon, H. Lee, J. Lee, L. Kim, P. Y., and K. B., “Vigilare: Toward Snoop-based Kernel Integrity Monitor,” in *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS’12)*, 2012.
- [110] J. Zaddach, A. Kurmus, D. Balzarotti, E. O. Blass, A. Francillon, T. Goodspeed, M. Gupta, and I. Koltsidas, “Implementation and Implications of a Stealth Hard-Drive Backdoor,” in *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC’13)*, 2013.
- [111] Broadcom Corporation, “Broadcom NetXtreme Gigabit Ethernet Controller,” <http://www.broadcom.com/products/BCM5751>.
- [112] Flashrom, “Firmware Flash Utility,” <http://www.flashrom.org/>.
- [113] Advanced Micro Devices, Inc., “BIOS and Kernel Developer’s Guide for AMD Athlon 64 and AMD Opteron Processors,” <http://support.amd.com/TechDocs/26094.PDF>. [Online]. Available: <http://support.amd.com/us/ProcessorTechDocs/26094.PDF>
- [114] H. William, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, New York, 2007.
- [115] “MD5 Hash Functions,” <http://en.wikipedia.org/wiki/MD5>.
- [116] Intel, “82574 Gigabit Ethernet Controller Family: Datasheet,” <http://www.intel.com/content/www/us/en/ethernet-controllers/82574l-gbe-controller-datasheet.html>.
- [117] “RWEverything Tool,” <http://rweverything.com/>.
- [118] SuperPI, <http://www.superpi.net/>.
- [119] Advanced Micro Devices, Inc., “AMD K8 Architecture,” http://commons.wikimedia.org/wiki/File:AMD_K8.PNG.
- [120] R. Wojtczuk and J. Rutkowska, “Attacking Intel Trust Execution Technologies,” 2009.
- [121] —, “Attacking Intel TXT via SINIT Code Execution Hijacking,” http://www.invisiblethingslab.com/resources/2011/Attacking_Intel_TXT_via_SINIT_hijacking.pdf, November 2011.

- [122] A. Fattori, R. Paleari, L. Martignoni, and M. Monga, “Dynamic and Transparent Analysis of Commodity Production Systems,” in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE’10)*, 2010.
- [123] R. R. Branco, G. N. Barbosa, and P. D. Neto, “Scientific but Not Academical Overview of Malware Anti-Debugging, Anti-Disassembly and Anti-VM Technologies,” in *Black Hat*, 2012.
- [124] F. Zhang, K. Leach, A. Stavrou, H. Wang, and K. Sun, “Using Hardware Features for Increased Debugging Transparency,” in *Proceedings of The 36th IEEE Symposium on Security and Privacy (S&P’15)*, 2015.
- [125] J. Rutkowska, “Blue Pill,” <http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html>, 2006.
- [126] VIA Technologies, Inc., “VT8237R South Bridge, Revision 2.06,” December 2005.
- [127] S. Vogl and C. Eckert, “Using Hardware Performance Events for Instruction-Level Monitoring on the x86 Architecture,” in *Proceedings of the 2012 European Workshop on System Security (EuroSec’12)*, 2012.
- [128] G. Pek, B. Bencsath, and L. Buttyan, “nEther: In-guest Detection of Out-of-the-guest Malware Analyzers,” in *Proceedings of the 4th European Workshop on System Security (EuroSec’11)*, 2011.
- [129] checkvm: Scoopy doo, http://www.trapkit.de/research/vmm/scoopydoo/scoopy_doo.htm.
- [130] DynamoRIO, “Dynamic Instrumentation Tool Platform,” <http://dynamorio.org/>.
- [131] CLOC, “Count lines of code,” <http://cloc.sourceforge.net/>.
- [132] “Keylogger Malware Found on UC Irvine Health Center Computers,” <http://www.scmagazine.com/keylogger-malware-found-on-three-uc-irvine-health-center-computers/article/347204/>.
- [133] “Credit Card Data Breach at Barnes & Noble Stores,” http://www.nytimes.com/2012/10/24/business/hackers-get-credit-data-at-barnes-noble.html?_r=3&.
- [134] T. Holz, M. Engelberth, and F. Freiling, “Learning More About the Underground Economy: A Case-Study of Keyloggers and Dropzones,” in *Proceedings of The 14th European Symposium on Research in Computer Security (ESORICS’09)*, 2009.
- [135] Ohloh, “Black Duck Software, Inc,” <http://www.ohloh.net>, access time: 7/16/2014.
- [136] “Common Vulnerabilities and Exposures list,” http://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/cvssscoremin-7/cvssscoremax-7.99/Linux-Linux-Kernel.html, access time: 07/06/2014.
- [137] F. Zhang, K. Leach, H. Wang, and A. Stavrou, “TrustLogin: Securing Password-Login on Commodity Operating Systems,” in *Proceedings of The 10th ACM Symposium on Information, Computer and Communications Security (AsiaCCS’15)*, 2015.

- [138] “Keylogger Products,” <http://www.keylogger.org>.
- [139] S. Sagioglu and G. Canbek, “Keyloggers,” *Technology and Society Magazine, IEEE*, 2009.
- [140] “Keyboard Scan Code Set 1,” <http://www.computer-engineering.org/ps2keyboard/scancodes1.html>.
- [141] F. Wecherowski, “A Real SMM Rootkit: Reversing and Hooking BIOS SMI Handlers,” *Phrack Magazine*, 2009.
- [142] Intel, “Universal Host Controller Interface (UHCI) Design Guide,” <ftp.netbsd.org/pub/NetBSD/misc/blymn/uhci11d.pdf>.
- [143] —, “Enhanced Host Controller Interface Specification for Universal Serial Bus,” <http://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/ehci-specification-for-usb.pdf>.
- [144] —, “eXtensible Host Controller Interface for Universal Serial Bus (xHCI),” <http://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/extensible-host-controller-interface-usb-xhci.pdf>.
- [145] J. Schiffman and D. Kaplan, “The SMM Rootkit Revisited: Fun with USB,” in *Proceedings of 9th International Conference on Availability, Reliability and Security (ARES’14)*, 2014.
- [146] Intel, “PCI/PCI-X GbE Family of Controllers: Software Developer Manual,” <http://www.intel.com/content/www/us/en/ethernet-controllers/pci-pci-x-family-gbe-controllers-software-dev-manual.html>.
- [147] “Intel 64 and IA-32 Architectures Optimization Reference Manual,” <http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html>.
- [148] “C-Scale Frequency Reference Guide for Musicians,” <http://www.ronelmm.com/tones/cscale.html>.
- [149] “Free Keylogger Pro,” <http://freekeyloggerpro.com/>.
- [150] “Logkeys Linux keylogger,” <https://code.google.com/p/logkeys/>.
- [151] N. Collignon, “In-memory Extraction of SSL Private Keys,” <http://c0decstuff.blogspot.com/2011/01/in-memory-extraction-of-ssl-private.html>, 2011.
- [152] “OpenSSH,” <http://www.openssh.com>, access time: 09/01/2014.
- [153] F. Zhang, “IOCheck: A Framework to Enhance the Security of I/O Devices at Runtime,” in *Proceedings of The 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN’13)*., 2013.

Curriculum Vitae

Fengwei Zhang received his M.S. degree in Computer Science from Columbia University in 2010. He also received a dual B.S. degree in Computer Science from North China University of Technology and Southern Polytechnic State University in 2008. His current research focuses on trustworthy execution, memory introspection, system integrity checking, and transparent malware debugging.