
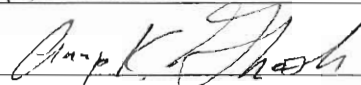
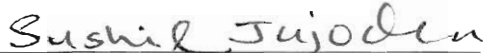
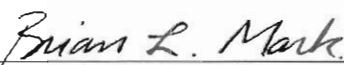


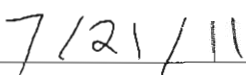


HARDWARE-ASSISTED PROTECTION AND ISOLATION

by

Jiang Wang
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Information Technology

Committee:

 _____	Dr. Angelos Stavrou, Dissertation Director
 _____	Dr. Anup K. Ghosh, Committee Member
 _____	Dr. Sushil Jajodia, Committee Member
 _____	Dr. Brian L. Mark, Committee Member
 _____	Dr. Daniel Menasce, Senior Associate Dean, Department Chair
 _____	Dr. Lloyd J. Griffiths, Dean, Volgenau School of Engineering
Date:  _____	Summer Semester 2011 George Mason University Fairfax, VA

Hardware-Assisted Protection and Isolation

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

By

Jiang Wang
Master of Engineering
Zhejiang University, 2001
Bachelor of Engineering
Zhejiang University, 1998

Director: Angelos Stavrou, Professor
Department of Computer Science

Summer Semester 2011
George Mason University
Fairfax, VA

Copyright © 2011 by Jiang Wang
All Rights Reserved

Dedication

To my parents.

Acknowledgments

I would like to thank some people who helped me to earn my PhD.

I would like to thank my advisor, Dr. Angelos Stavrou. We spent countless hours discussing the topics in this dissertation, and he taught me how to do research, design experiments, and write papers.

I would like to thank my committee members: Dr. Anup Ghosh, Dr. Sushil Jajodia, and Dr. Brian L. Mark. They spent time attending to my proposal and defense meetings, and they provided me with valuable advice.

I would like to thank Dr. Duminda Wijesekera. He brought me to George Mason University and helped me to obtain financial support during my early years here. Lately, Dr. Anup Ghosh and my advisor have provided me with financial support.

I would like to thank many coworkers and students in the Center for Secure Information Systems(CSIS) and the Distributed Security Lab. From Dr. Yih Huang, I learned a lot about Linux kernels. Dr. Kun Sun provided many insightful ideas and much feedback about the topics in this dissertation. I had many interesting discussions with my friends, Lei Zhang and Zhaohui Wang, about research ideas. My friend Fengwei Zhang helped me a great deal with coreboot coding and experiments. I would also like to thank CSIS students Quan Jia, Meixing Le, Bo Yu, Chen Liang, Brian Schulte, Haris Andrianakis, Javier Almillategui, Nelson Nazzicari, Ryan Johnson, Sharath Hiremagalore, and Spyros Panagiotopoulos.

I would like to thank my parents for their support during my PhD study.

Table of Contents

	Page
List of Tables	viii
List of Figures	ix
Abstract	xi
1 Introduction	1
1.1 Thesis Contributions	5
1.2 Thesis Organization	6
2 Related Work	7
3 Background	10
3.1 Hypervisor Architecture	10
3.2 QEMU	12
3.3 Typical Architecture of a Computer	12
3.4 System Management Mode	13
3.4.1 ACPI Sleeping States	14
3.4.2 BIOS, UEFI and Coreboot	14
3.4.3 DQS Settings and DIMM MASK	15
4 HyperCheck	17
4.1 Introduction	17
4.2 Threat Model	17
4.2.1 Attacker’s Capabilities	17
4.2.2 General Assumptions	17
4.2.3 In-scope Attacks	18
4.2.4 Limitations	18
4.3 System Architecture	19
4.3.1 Acquiring the Physical Memory	20
4.3.2 Translating the Physical Memory	22
4.3.3 Reading and Verifying the CPU Registers	23
4.4 Implementation	23
4.4.1 Memory Acquiring Module	25

4.4.2	Analysis Module	26
4.4.3	CPU Register Checking Module	27
4.4.4	HyperCheck-II	29
4.5	Evaluation	30
4.5.1	Verifying the Static Property	30
4.5.2	Detection	31
4.5.3	Monitoring Overhead	31
5	Evasion Attacks	36
5.1	Introduction	36
5.2	Design of Existing SMM-based Integrity Checking Systems	37
5.3	Threat Model of Evasion Attacks	38
5.3.1	Type I: Directly Intercepting SMI	39
5.3.2	Type II: Indirectly Deriving Periodic SMI	42
5.3.3	Type III: Avoiding Random SMI	46
5.4	Defense Strategies	47
5.4.1	Preventing Evasion Attacks	48
5.5	Implementation	49
5.5.1	SMM-Based Integrity	49
5.5.2	Evasion Attacks Implementation	49
5.6	Evaluation	54
5.6.1	Performance Analysis	54
5.6.2	Detecting Attacks	58
5.6.3	Detecting Evasion Attacks	62
6	Hardware-assisted Isolation	64
6.1	Introduction	64
6.2	Thread Model and Assumptions	66
6.3	SecureSwitch Framework	67
6.3.1	Secure OS Loading	68
6.3.2	Secure OS Switching	69
6.3.3	Secure OS Isolation	71
6.4	System Design	73
6.4.1	Loading Two OSes	73
6.4.2	Switching Between Two OSes	74
6.4.3	Enforcing System Isolation	77
6.5	Implementation & Experimental Results	81

6.5.1	Trusted Computing Base (TCB)	82
6.5.2	OS Loading and Switching Latency	82
6.5.3	Comparison with Other Methods	88
7	Hardware-assisted Forensic	90
7.1	Introduction	90
7.2	Architecture	92
7.2.1	Off-line Investigation	93
7.2.2	On-line Investigation	95
8	Security Analysis and Open Problems	99
8.1	Security Analysis of HyperCheck	99
8.2	Security Analysis of SecureSwitch	100
8.3	Open Problems	102
8.3.1	Checking Dynamic Data	102
8.3.2	Memory Isolation Granularity	102
9	Conclusions and Future Work	104
9.1	Conclusions	104
9.2	Future Work	105
	Bibliography	106

List of Tables

Table	Page
4.1 Symbols for Xen hypervisor, Domain 0, Linux, and Windows	31
4.2 Time overhead of HyperCheck and other methods	35
4.3 Comparison between HyperCheck and other methods	35
5.1 System Overhead of SMI Detector	56
5.2 Detection probability of NTEA, both intervals are uniformly distributed between 1 and the number in the column.	59
5.3 Detection probability of NTEA, SMM intervals are normally distributed between 1 and the number in the column.	60
6.1 Switching Time	83
6.2 Comparing SecureSwitch with Other systems	88

List of Figures

Figure	Page
1.1 HyperCheck can offer protection to services running above BIOS	3
3.1 The architecture of a Type1 hypervisor (Xen)	11
3.2 The architecture of a Type2 hypervisor (QEMU)	11
3.3 Typical hardware layout of a computer	12
4.1 The architecture of HyperCheck	19
4.2 Network overhead for variable packet sizes.	32
4.3 Network overhead for variable data sizes.	32
4.4 Overhead of the operations in SMM	33
4.5 Overhead of the XOR data in SMM	34
5.1 Directly Intercepting SMI. The attacker inserts a code preamble before the SMM-based integrity checks can occur.	40
5.2 Intercept port 0xB2 SMI triggering event. This attack requires the scanning of memory to identify the call locations for 0xB2	41
5.3 Launch of attacks between two SMIs.	43
5.4 Detecting the invocation of SMI: T_D denotes the detection duration while we check the Time Stamp Counter every $T_d + T_{DI}$, where T_d is the duration time for one counter checking process and T_{DI} is the time interval between two counter checking processes.	44
5.5 Measuring the SMM duration(T_{SMM}): T_{SI} is the time interval between two SMIs. $T_d + T_{DI}$ is the time delay between two counter polling processes when the system is not in SMM mode. t_a is the last polling time before the system enters SMM, and t_b is the first polling time after the system exits SMM. . .	45
5.6 Layout of the MSI Message Data Register [1].	50
5.7 CaffeineMark 3.0 Micro-benchmarks of System Performance. The legend notation is “sample duration:check interval” in milliseconds where (R) denotes random check interval in [1, xxx]. The longer and more frequent the sampling, the more impact it has on system performance.	55

5.8	System Overhead of SMI Detector	56
5.9	SMI Detector's Detection Probability.	57
5.10	Detection Probability of TEA, unified distribution.	59
5.11	Detection Probability of TEA, normal distribution	60
6.1	Architecture of SecureSwitch System	67
6.2	State Machine for OS Switching.	68
6.3	Switching Flow from Untrusted OS to Trusted OS.	74
6.4	User Space Suspend Breakdown	85
6.5	Kernel Space Suspend Breakdown	85
6.6	Kernel Space Wakeup Breakdown	86
6.7	User Space Wakeup Breakdown	87
7.1	The architecture of the memory forensic tool.	92

Abstract

HARDWARE-ASSISTED PROTECTION AND ISOLATION

Jiang Wang, PhD

George Mason University, 2011

Dissertation Director: Dr. Angelos Stavrou

Software is prone to contain bugs and vulnerabilities. To protect it, researchers normally go to a lower layer, such as protecting the applications from the kernel or protecting the operating systems from the hypervisor, because the upper layer is controlled and depends on the lower layer. However, even a small hypervisor, which partitions the system hardware resources into different domains to support and isolate multiple virtual machines, may contain some vulnerabilities and is hard to protect within itself.

In this dissertation, we use a hardware-assisted method to monitor the integrity of the software running on top it. We present HyperCheck, a hardware-assisted tampering detection framework designed to protect the integrity of hypervisors or operating systems (OS). HyperCheck leverages the CPU System Management Mode (SMM), present in x86 systems and a dedicated commercial network card, to securely generate and transmit the full state of the protected machine to an external server. Using HyperCheck, we were able to ferret-out rootkits that targeted the integrity of both the Xen hypervisor and traditional OSes. Moreover, HyperCheck is robust against attacks that aim to disable or block its operation. Our experimental results show that HyperCheck can produce and communicate a scan of the state of the protected software in less than 40ms.

In addition to detecting the intrusion, another promising approach to protect the end user's computer is to separate sensitive tasks, such as financial-related activities, from un-sensitive tasks. For this purpose, we designed a system which has two operating systems installed: one trusted and the other untrusted. The trusted OS runs only the trusted applications and is guaranteed to be separated from the untrusted OS. Without using a hypervisor, we leverage the commercial hardware and the BIOS to enforce the isolation between the two OSes. By utilizing the standard ACPI S3 sleep, we also achieve a short delay when switching between the two OSes.

Chapter 1: Introduction

An Operating System (OS) provides services to all of the applications running above it, so it becomes a common target of adversaries. Since modern operating systems have to enable various types of hardware and support various types of applications, they are normally quite complex and include millions of lines of code. For example, Linux source code contains more than ten million lines after 2.6.27, and the Microsoft Windows kernel is known to be even larger. As a consequence, the number of vulnerabilities in the operating systems is large and difficult to reduce, providing adversaries with ample opportunities to attack.

To protect operating systems, many researchers employ a hypervisor¹, which separates the system hardware resources into different partitions and can run multiple different operating systems (guest operating systems) in each one. These partitions are also referred to as virtual machines (VM). Hypervisors have become the de facto standard in server consolidation because they decrease the energy footprint and management costs of modern computing clusters. In addition, hypervisors are increasingly used as components to enforce system security and resilience [2–8]. Since the hypervisors run beneath the operating systems, they can monitor the integrity of the guest operating systems and enforce the confidentiality [9] of some virtual machines.

Moreover, researchers also divide the applications into untrusted and trusted ones, and they use hypervisors to guarantee the isolation between them [4, 5, 9–14]. Web browsing, online gaming, and social web portals are examples in the former category; online banking, shopping, and business email belong in the latter category. The researchers provide different virtual machines for different applications, and the hypervisor can intercept the illegal access from the untrusted OS to the trusted OS.

¹Also referred to as Virtual Machine Monitors VMMs

The hypervisor-based protection and isolation relies on the assumptions that the virtual machine monitor (VMM) is not compromised and that there is no exposed path between the different virtual machines. However, this widespread adoption of virtualization has attracted the attention of attackers towards VMM vulnerabilities. Indeed, there has recently been a surge in the reported vulnerabilities of commercial and open-source hypervisors [15]. Moreover, the number and nature [16, 17] of attacks against the hypervisors are poised to grow.

This increasing attack trend has spurred research towards reducing the trusted computing base (TCB) of current commercial hypervisors [18]. Some researchers have developed new specialized prototype hypervisors [7, 12]. However, having a small code base can only limit the code exposure and, as a result, the attack surface of the hypervisor; it cannot provide strong guarantees about the code integrity of all hypervisor components. Other researchers have tried to improve hypervisor security by adding in-kernel protections [14, 19].

In this thesis, we try to answer the following questions: What if the hypervisor is compromised? Can we protect the operating systems' integrity and isolate them without using hypervisors, and how? To answer these questions, we go to a lower layer, using the BIOS and the physical hardware. To protect the integrity of the hypervisors and operating systems, we designed and implemented a system named HyperCheck; to isolate the untrusted operating systems from trusted operating systems, we designed and implemented a system named SecureSwitch.

HyperCheck is designed to protect the integrity of VMMs and, for some classes of attack, the underlying operating system (OS). To achieve this, HyperCheck harnesses the CPU System Management Mode (SMM), which is present in all x86 commodity systems to create a snapshot view of the current state of the CPU and the memory registers of the protected machine. This information is securely and verifiably transmitted using a network card to a remote analysis server. Using this information, the analysis server can identify any tampering by comparing the newly-generated view with the one recorded when the machine was initialized. If the two views do not match, a human operator is notified. As shown

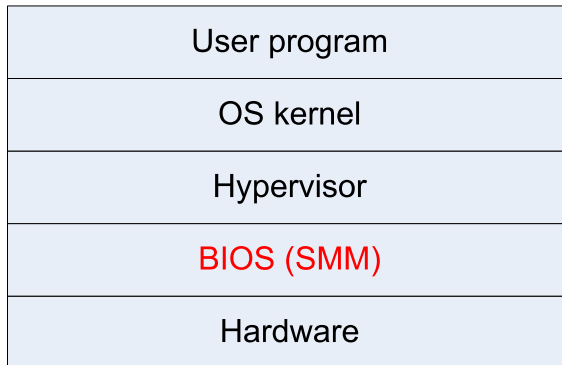


Figure 1.1: HyperCheck can offer protection to services running above BIOS

in Figure 1.1, HyperCheck works at the BIOS level and can protect the software above it. Our assumptions are that the attacker does not have physical access to the machine and that the SMM BIOS is locked and thus cannot be altered during run. We do not explicitly require trusted boot to initialize HyperCheck [12,13]. However, having a machine equipped with trusted boot can prevent attacks against HyperCheck that simulate a hardware reset.

Unlike previous work [20] that used specialized PCI hardware, we are able to acquire a complete view of the target machine’s state, including the entire memory and CPU registers. In addition, our approach is able to thwart attacks aimed at disabling, blocking, or even taking over PCI devices. To evaluate the validity and performance of our approach, we implemented two prototypes for HyperCheck. HyperCheck-I uses QEMU [21], a full-system emulator, to emulate the PCI NIC, while HyperCheck-II is based on an Intel e1000 physical NIC. Using our prototypes, we were able to ferret out rootkits aimed at Xen [22] hypervisor, Xen Domain 0, Linux, and Windows. Our experimental results indicate that HyperCheck does not cause prohibitive performance overhead and requires only a few milliseconds to completely transmit each snapshot. We also evaluate the security of HyperCheck and other SMM-based protection mechanisms. After identifying possible attacks to these systems, we show the defense mechanisms. Moreover, we try to use a similar framework for computer forensics.

In addition to protecting integrity, we also attempt to tackle the secure OS isolation

problem without using a hypervisor or any mutable shared code. We designed a firmware-assisted system called *SecureSwitch*, which allows users to switch between a trusted and an untrusted operating system on the *same* physical machine with a short switching time. The basic input/output system (BIOS, or firmware) is the only trusted computing base that ensures the resource isolation between the two OSes and enforces a trusted path for switching between them. The attack surface within our system is significantly smaller than the hypervisor or software-based systems: we can protect the integrity of the BIOS code by using hardware lock (e.g., BIOS_CNTL register [23] in Intel ICHs) to set the BIOS code as read-only, or by using TPM to verify the integrity of the BIOS code. Furthermore, our system guarantees a strong resource isolation between the trusted and untrusted OSes. If the untrusted OS has been compromised, it still cannot read, write, or execute any of the data and applications in the trusted OS.

Overall, our system can ensure isolation on the following computer components:

- **Memory Isolation:** All OS environments run in separate Dual In-line Memory Modules (DIMM). A physical-level memory isolation is ensured by the BIOS because only the BIOS can initialize and enable the DIMMs. No software can initialize or enable DIMMs after the system boots up.
- **CPU Isolation:** The different operating systems never run concurrently. When one OS is switched off, the entire CPU state is saved and flushed. We use ACPI S3 sleep mode to help achieve CPU suspend/restore.
- **Hard Disk Isolation:** Each OS can have its own dedicated encrypted hard disk. We use RAM disk to save the temporary sensitive data in the trusted OS. The untrusted OS cannot access the RAM disk in the trusted OS due to the memory isolation.
- **Other I/O Isolation:** When one OS is switched off, all contents maintained by the device drivers (e.g, graphics card, network card) are saved and the devices are then powered off. This guarantees that the untrusted OS cannot steal any sensitive data from the I/O devices.

To prevent fake OS attacks, we must enforce a trusted path when the system switches from the untrusted OS to the trusted OS. This guarantees that the system really suspends the untrusted OS and wakes up the trusted OS. Otherwise, a sophisticated adversary may fake an S3 sleep in the untrusted OS by manipulating the hardware (e.g., shutting down the monitor) and then deceiving the user with a faked trusted OS environment, which is controlled by the compromised untrusted OS. We refer to such events as “Fake OS attacks.” To prevent such an attack, we use the power button and power LED to indicate to the user when the system enters the BIOS after one OS is suspended. The BIOS will then wake up one OS according to a system OS flag that indicates which OS should be woken. The value of the flag can only be manually changed by the user; it cannot be changed by any software.

We harness the Advanced Configuration and Power Interface (ACPI) [24] S3 sleep mode to help achieve a short OS switching latency. Because two OSes are maintained in RAM memory at the same time, the switching latency is only about six seconds, which is much faster than switching between two OSes on a multi-boot computer or switching using ACPI S4 mode [25]. It is slower than the hypervisor-based solutions; however, we don’t need to worry about the potential vulnerabilities in the hypervisor. Moreover, our system can be used as a complementary approach to existing hypervisor- and OS-protection solutions.

1.1 Thesis Contributions

1. Designed a novel hardware-assisted tampering detection framework that creates a complete snapshot of the state of the system with commercial hardware and no modification to the installed software.
2. Implemented two prototypes for hardware-assisted protection: one based on QEMU, and the other based on the real hardware. The latter has overhead in the order of a few milliseconds. Using our prototype, we demonstrate that we can successfully detect rootkits and code integrity attacks against Xen VMM, Xen Domain 0, Linux, and Windows.

3. Provided a systematic analysis of attacks to the SMM-based protecting systems. Implemented prototypes that demonstrate various attacks and their effectiveness, and developed defense mechanisms to curtail these attacks.
4. Designed and implemented a Secure OS switching system that does not use any mutable software layer. Our system depends on the BIOS and some hardware properties to enforce a trusted path when switching between the two OSes. Our solution requires no modification of the commodity OS, and the switching time for our prototype is approximately six seconds.

1.2 Thesis Organization

Chapter 2 describes the related work in this area. Chapter 3 introduces background knowledge. We then describe the design, implementation, and performance evaluation of the hardware-assisted integrity monitor(HyperCheck) in Chapter 4. Next, we analyze the security attacks to HyperCheck in Chapter 5. Chapter 6 introduces the design, implementation, and evaluation of the hardware-assisted isolation system (SecureSwitch), and Chapter 7 describes the hardware-assisted forensic. Chapter 9 concludes the thesis.

Chapter 2: Related Work

Protecting software from integrity attacks using hardware-assisted techniques is not new: researchers used a special-purpose PCI device to acquire the physical memory either for rootkit detection [20, 26] or for forensic purpose [27] in the past. The closest system to our work is Copilot [20], which employed a special PCI device to poll the physical memory of the host and periodically send it to an admin station. In HyperCheck, we do not require specialized hardware, only an out-of-the-box network card. We also offer a complete view of the CPU state, including its registers. Such a view is important to prevent copy-and-change attacks that can mislead the PCI card to scan the wrong regions of memory and report erroneously that the system is not affected.

Another closely related work is HyperGuard [28]. Rutkowska *et al.* suggested using SMM of the x86 CPU to monitor the integrity of the hypervisors. Although we have goals similar to the HyperGuard project, the use of a network card allows us to outsource the analysis of the state snapshot. This results in a drastic improvement in the performance of the system, reducing the system busy time from seconds to milliseconds. Due to its low performance overhead, HyperCheck can also monitor the code and data of the privileged domain and underlying OSes. Another difference is that the monitoring machine can be used to detect the DoS attacks to the SMM code.

DeepWatch [17] also offers detection of hypervisor rootkits, called virtualization malware in DeepWatch, by using the embedded micro-controller(s) in the chipset. DeepWatch is signature-based and is used to detect rootkits relying on hardware-assisted virtualization technologies, such as Intel VT-d [29]. Contrarily, HyperCheck performs anomaly detection and can thus identify a larger class of software rootkits.

Flicker [13] uses a TPM-based method to provide a minimum Trusted Code Base (TCB), which can be used to detect modification to the kernels. Flicker requires advanced hardware

features, such as Dynamic Root of Trust Measurement (DRTM) and late launch. In contrast, HyperCheck uses the static Platform Configuration Registers (PCRs) to secure the booting process. In addition, by sending out the data, HyperCheck has a lower overhead on the target machine compared with Flicker. To reduce the overhead of Flicker, TrustVisor [12] has a small footprint hypervisor to perform some cryptography operations. However, all legacy applications should be ported for TrustVisor to work. TrustVisor requires DRTM as well.

Another branch of research attempts to improve the security of the hypervisor by adding hooks [30] and enforcing security policies between virtual machines [31]. These methods are hypervisor-specific and run at the same level as the hypervisor. HyperCheck monitors the hypervisor state from a lower level and, thus, is complementary to these methods.

Furthermore, there is a plethora of research aimed towards protecting the Linux kernel [2, 4–8, 26]. Baliga [26] *et al.* use a PCI device to acquire the memory and automatically derive the kernel invariance. We currently discover the kernel invariance manually, but we could employ their techniques directly and without modification. Litty [2] *et al.* developed a technique to discover the address of key data structures that are instantiated during runtime by relying on processor hardware and executable file specifications. However, they also rely on the integrity of the underlying hypervisors. HyperCheck first obtains the virtual addresses of these symbols through the symbol file, but it then calculates the physical addresses through CPU registers. Therefore, HyperCheck can obtain the correct view of the system memory even if the underlying OS or hypervisor is compromised and the page tables are altered. Other existing research [5–8], including work by Jiang *et al.*, depend upon the integrity of the hypervisor to protect the kernel. Our work is complementary and can be employed as a meta-protection mechanism to guard the integrity of OS-level defenses. A lot of recent work has gone towards using SMM to generate efficient rootkits [32–35]. These rootkits can be used either to obtain root privilege or as key-stroke loggers. We use SMM to offer integrity protection by monitoring the states of hypervisors and operating systems.

SecureSwitch was inspired by Lampson’s Red/Green separation idea [36]. The closest

in terms of concept is the Lockdown [25] system, which places two OSes on one machine and isolates them with help of a small hypervisor. To switch, it hibernates one OS and then wakes up another one. If implemented carefully, Lockdown can provide isolation between two OSes. Unfortunately, it requires more than 40 seconds to switch because hibernating requires writing the whole main memory content to the hard disk and reading it back later on. In contrast, SecureSwitch can accommodate two OSes into the memory at the same time and offers switching times of approximately 6 seconds. In addition, Lockdown relies on mutable shared code using a light-weight hypervisor, while SecureSwitch does not.

There is a line of research that uses hypervisors to add an extra layer of control between the OSes and the underlying hardware, including HyperSpace [37], Terra [9], Safefox [10], Tahoma system [11], Overshadow [38], and Nettop [39]. Others attempt to protect the integrity of the hypervisor [14, 19, 30, 40, 41], or to protect the kernel [7, 8, 42]. All of these systems depend upon the integrity of the shared hypervisor code for the isolation between two environments. Nevertheless, attacks against the hypervisors are more and more frequent today [15, 16, 43]. Although the hypervisor may have a smaller attack surface compared to the traditional OSes, it is still vulnerable to attack. SecureSwitch employs immutable BIOS-protected code so that minimal code is shared between the trusted and the untrusted environments. The SecureSwitch system is capable of running the legacy applications in the trusted OS, and it can run for a long time, unlike Flicker; SecureSwitch does not require DRTM, while Flicker and TrustVisor do.

Chapter 3: Background

In this chapter, we describe the architecture of typical hypervisors, the hardware components of a computer, and the System Management Mode.

3.1 Hypervisor Architecture

Hypervisors (also called virtual machine monitors) partition the hardware resource, such as CPU, memory, and I/O, in different domains, and each domain can run an operating system. These domains are also called virtual machines. There are two types of hypervisors: type 1 (also called native or bare-metal) and type 2 (also called hosted). The former runs directly on the hardware, as shown in Figure 3.1, while the latter runs as an application or as services on an operating system, as depicted in Figure 3.2. Here we are mainly concerned with type 1 hypervisors. Xen and Microsoft Hyper-V [44] are typical examples. While the type 2 hypervisors could be protected by current host-based intrusion detection systems (HIDS), the type 1 hypervisors are not fully compatible with HIDS.

Type 1 hypervisors incorporate with another privileged entity to manage other virtual machines, such as Domain 0 in Xen. The unprivileged virtual machines are also called Domain U in Xen. In X86-32 architecture of Xen, the hypervisor runs in ring 0 and Domain 0 runs in ring 1 [45]. In addition, there are some user level (ring 3) programs with higher privilege in Domain 0. One example is pygrub in Xen, which is used to boot other paravirtualized domains. Similarly, Hyper-V runs directly on the hardware and uses a root partition to manage other virtual machines (called partitions in Hyper-V) [44]. The root partition of Hyper-V is similar to the Domain 0 in Xen. These privileged domains are indispensable and should be monitored.

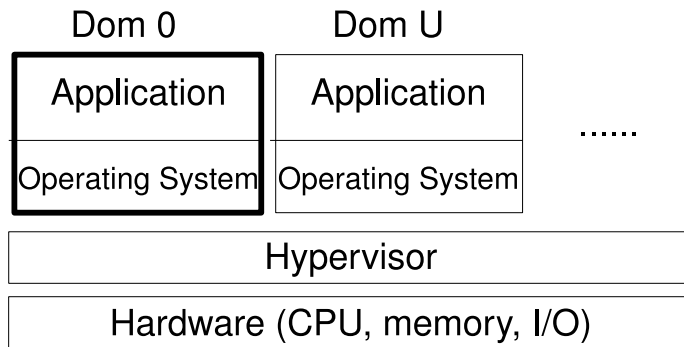


Figure 3.1: The architecture of a Type1 hypervisor (Xen)

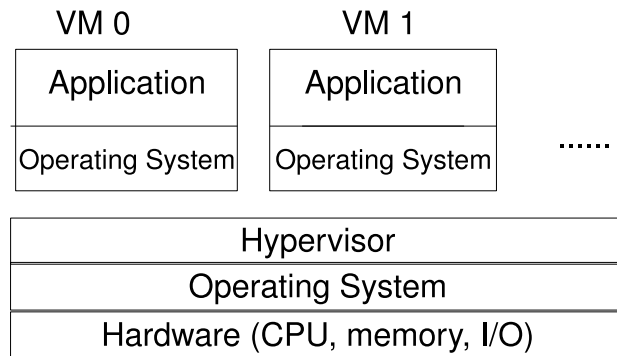


Figure 3.2: The architecture of a Type2 hypervisor (QEMU)

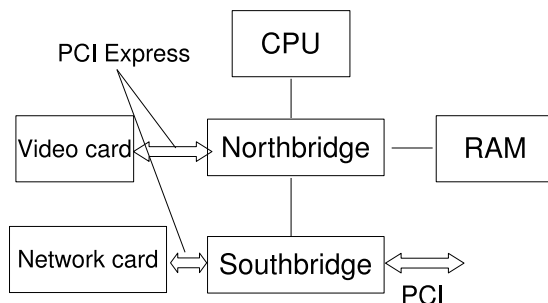


Figure 3.3: Typical hardware layout of a computer

3.2 QEMU

QEMU is a fast processor emulator using a portable dynamic translator. QEMU supports full-system emulation, in which a complete and unmodified operating system is run in a virtual machine, and Linux user mode emulation, where a Linux process compiled for one target CPU can be run on another CPU [21]. It can be used as Type 2 hypervisors. For HyperCheck-I, we use full system emulation to emulate the target machine, including the Ethernet cards.

3.3 Typical Architecture of a Computer

Figure 3.3 presents the internal architecture of a computer with one processor. The top is the Central Processing Unit. Northbridge connects the CPU with the RAM and video card. Southbridge is connected with Northbridge, the PCI bus, and other devices. HyperCheck uses a dedicated PCI Ethernet card, which is connected at Southbridge to read the physical memory. Note that both Northbridge and Southbridge provide PCI device interfaces so that they can be accessed and configured through PCI configuration methods. Recently, the PCI Express [46] bus standard was developed to replace the conventional PCI, PCI-X, and AGP bus. PCI Express could be connected with Northbridge and Southbridge. PCI

Express also introduces a much higher transfer rate than PCI. The highest transfer rate for PCI Express 2.0 is 5.0GT/s, while conventional PCI is normally only 133MB/s (32bit at 33MHz). The high speed of PCI Express provides an opportunity for HyperCheck to scan the physical memory quickly and with relatively low overhead.

3.4 System Management Mode

Intel IA-32 CPU provides three operation modes: real-address mode, protected mode, and system management mode [47]. Intel x86-64 CPU supports all of these modes and adds one more: IA-32e. CPU enters real-address mode following power-up or reset. The modern operating system then typically switches to protected mode. System management mode provides an operating system or executive with a transparent mechanism for implementing platform-specific functions, such as power management and system security. The processor enters SMM when the external SMM interrupt pin (SMI#) is activated or when an SMI is received from the advanced programmable interrupt controller (APIC). For current operating systems, such as Windows and Linux, the CPU runs in protected mode most of the time. Switching to SMM rarely occurs.

In SMM, the processor switches to a separate address space while saving the basic context of the currently-running program or task. SMM-specific code may then be executed transparently. Upon returning from SMM, the processor is placed back into its state prior to the system management interrupt. SMM was introduced with the Intel386 SL and Intel486 SL processors and became a standard IA-32 feature with the Pentium processor family [47]. The separate address space used by SMM is called the system management RAM (SMRAM). The memory model used to address bytes in this address space is similar to the real-address mode model. SMM is designed to be transparent to the operating system, and the SMRAM can be made inaccessible from other operating modes. We use this feature of SMM to run the register integrity checking code and to reliably drive the NIC.

3.4.1 ACPI Sleeping States

The Advanced Configuration and Power Interface (ACPI) establishes industry-standard interfaces that enable OS-directed configuration, power management, and thermal management of computer platforms [24]. ACPI defines four global states: *G0*, *G1*, *G2*, and *G3*. *G0* is the working state when a machine is fully running. *G1* is the sleeping state to achieve different levels of power saving. *G2* is called “Soft Off,” wherein the computer consumes only a minimal amount of power. In *G3*, the computer is completely shut down; aside for the real-time clock, the power consumption is zero.

G1 is subdivided into four sleeping states: *S1*, *S2*, *S3*, and *S4*. From *S1* to *S4*, the power saving increases, but the wakeup time also increases. In *S3*, all system context (CPU, chipset, cache) is lost aside from the RAM. *S3* is also referred to as *Standby* or *Suspend to RAM*. In *S4*, all main memory content is saved to non-volatile memory, such as a hard drive, and the machine (including the RAM) is powered off. *S4* is also referred to as *Hibernation* or *Suspend to Disk*. In both *S3* and *S4*, all of the devices may be powered off.

Not every machine or operating system supports all of the ACPI states. For instance, neither *S1* nor *S2* is used by Windows. *S3* and *S4*, however, are supported by all Linux 2.4 and 2.6 series kernels and recent Windows distributions (XP, Vista, 7). Our SecureSwitch uses *S3* operations provided by the operating system to help save the system context and later restore it. This dramatically saves our developing efforts.

3.4.2 BIOS, UEFI and Coreboot

The BIOS is an indispensable component of all computers. The main function of the BIOS is to initialize the hardware, including the processor, main memory, northbridge, southbridge, and hard disk, as well as some other necessary IO devices, such as the keyboard. The BIOS code is normally stored on a non-volatile ROM chip built into the system on the mother board.

When a computer is booted up, the main memory is not initialized and the BIOS is unaware of whether the memory modules exist or not. Therefore, the BIOS must scan the

hardware to find out the number, size, and additional parameters of the installed memory modules. The BIOS then initializes the memory to be useable. Similar steps are carried out for other hardware resources as well. The entire process is referred to as the Power-On Self-Test (POST). Finally, the BIOS initializes enough hardware so that it can transfer the control to an OS or a bootloader.

The BIOS is traditionally written by assembly language and works in real-address mode. In recent years, a new generation of BIOS, referred to as Unified Extensible Firmware Interface (UEFI) [48], has become increasingly popular in the market. UEFI is a specification that defines the new software interface between OS and firmware. One purpose of UEFI is to ease development by switching to the protected mode in a very early stage and writing most of the code in C language. A portion of the Intel UEFI framework (known as Tiano Core) is open source; however, the main function of the UEFI (i.e., to initialize the hardware) is still closed source.

Coreboot [49] (formerly known as LinuxBIOS) is an open source project aimed at replacing the proprietary BIOS (firmware) in most of today's computers. It performs a small amount of hardware initialization and then executes a so-called payload. In some sense, coreboot is similar to the UEFI-based BIOS. Coreboot also switches to the protected mode in a very early stage and is written mostly in C language. Our prototype implementation is based on coreboot V4. The reason why we choose to use coreboot instead of UEFI is that Coreboot has done all of the work of hardware initialization, but we need to implement UEFI firmware from scratch, including finding out all of the data sheets for our motherboard.

3.4.3 DQS Settings and DIMM MASK

There are many different types of RAM, and one of the most popular ones is the Double Data Rate Synchronous Dynamic Random Access Memory (DDR SDRAM). One feature of these DDR memories is that they include a special electrical signal referred to as “data strobes” (DQS). For proper memory reads to occur, the DQS must be properly timed to

align with the data valid window of the data (DQ) lines. The data valid window refers to the specific period of time when the DRAM chip drives (i.e., makes active) the DQ lines for the memory controller to read its data. If the DQS signal is not properly aligned, the memory access will *fail*. For DDR1, the parameters of DQS can be automatically set by the hardware. For DDR2 and DDR3, the DQS settings should be programmed by the BIOS [50]. We use DDR2 memory in our system.

A motherboard usually has more than one DIMM slot. The BIOS normally uses a variable named “DIMM_MASK” to enable the DIMMs. Our SecureSwitch system assigns one DIMM to one OS. When one OS is running, the BIOS will only enable the DIMM assigned to that OS with the corresponding DQS settings.

Chapter 4: HyperCheck

4.1 Introduction

In this chapter, we describe the design, implementation, and performance evaluation of HyperCheck. HyperCheck is a hardware-assisted integrity monitor system, which can check the integrity of hypervisors or operating system by using SMM and a PCI NIC.

4.2 Threat Model

4.2.1 Attacker’s Capabilities

We assume that the adversary has the following capabilities: he/she is able to exploit vulnerabilities in any software running in the machine after boot-up. This includes the VMM and all of its privileged components. For instance, the attacker can compromise a guest domain and escape to the privileged domain. In Xen 3.0.3, pygrub [45] 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 [51]. The attacker can also modify the hypervisor code or data using any known or zero-day attack. For instance, the DMA attack [16] hijacks a device driver to perform unauthorized DMA to the hypervisor’s code or data.

4.2.2 General Assumptions

The attacker cannot tamper with or replace the installed PCI NIC with a malicious NIC using the same driver interface. In addition, if the SMM code is integrated with BIOS, we assume the SMRAM is properly setup by BIOS upon boot time. If the SMM code is not included in the BIOS, it must be reliably uploaded to the SMRAM during boot. This can

be done either by using trusted boot or by using the management interface to bootstrap the computer. In this case, to initialize the SMM code, a trusted bootstrap mechanism must be employed. The SMRAM is locked once it is properly set up. Once it is locked, we assume that it cannot be subverted by the attacker (an assumption supported by current hardware). Attacks that attempt to modify the SMM code [52–54] are beyond the scope of this thesis.

4.2.3 In-scope Attacks

HyperCheck aims to detect the in-memory, Ring-0 level (hypervisor or general OS) rootkits and the rootkits in privileged domains of hypervisors. A rootkit is a set of programs and code that allows a permanent or consistent, undetectable presence on a computer [55]. One type of rootkit only modifies the memory and/or registers and runs in the kernel level. For example, the `idt-hook` rootkit [56] modifies the interrupt descriptor table (IDT) in the memory and then gains the control of the complete system. A stealthier version of the `idt-hook` rootkit could keep the original IDT unchanged by copying it to a new location and altering it. The attacker could then change the IDTR register to point to the new location. When it comes to the hypervisor-level rootkit, there is yet another kernel, the hypervisor kernel, which runs underneath the operating system kernel. There are existing methods to detect in-memory, kernel-level rootkits. We try to bridge this gap by introducing HyperCheck.

4.2.4 Limitations

Currently, our analysis cannot protect against attacks that modify dynamic data. There are two types of threats: (1) modification to the dynamically-generated function pointers and (2) return-oriented attacks. In these attacks, the control flow is redirected to memory location controlled by the attacker. There are techniques to thwart such attacks: the non-executable bit in new CPUs and Address Space Layout Randomization, to name a few. HyperCheck can leverage and integrate these techniques to provide full protection, but this

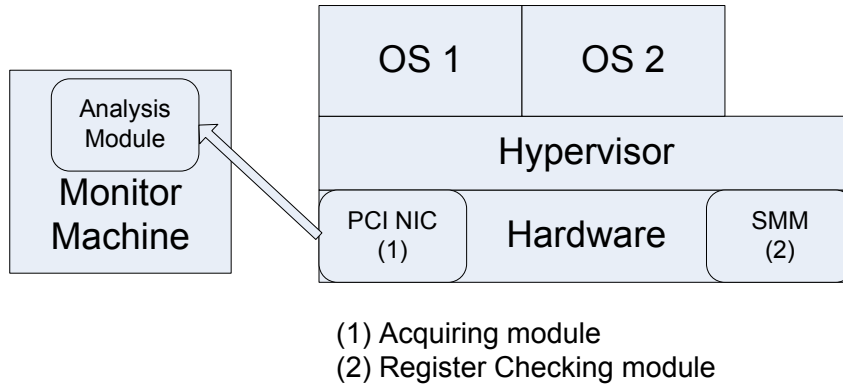


Figure 4.1: The architecture of HyperCheck

was not part of our implementation in this thesis. Having said this, we can still detect the presence of the malfeasance if it tries to interfere with the VMM code or the statically-defined function pointer.

4.3 System Architecture

HyperCheck is composed of three key components: the physical memory-acquiring module, the analysis module, and the CPU register-checking module. The memory-acquiring module reads the contents of the physical memory of the protected machine and sends them to the analysis module. The analysis module then checks the memory contents and verifies whether anything has been altered. The CPU register-checking module reads the registers and validates their integrity. The overall architecture of HyperCheck is shown in Figure 4.1. Before introducing the key components, we first describe our design principles.

Our main design principle is that HyperCheck should not rely on any software running on the machine aside from the boot loader. Since the software may be compromised, one cannot trust even the hypervisor. Therefore, we use hardware – a PCI Ethernet card – as a memory-acquiring module and SMM to read the CPU registers. Usually, Ethernet cards are PCI devices with bus master mode enabled and are capable of reading the physical memory through DMA, which does not need help from the CPU. SMM is an independent

operating mode and could be made inaccessible from protected mode, which is what the hypervisor and privileged domains run in.

Previous researchers only used PCI devices to read the physical memory. However, CPU registers are also important because they define the location of active memory used by the hypervisor or by an OS kernel such as CR3 and IDTR registers. Without these registers, the attacker can launch a copy-and-change attack. This means that the attacker copies the memory to a new location and modifies it, then he/she updates the register to point to the new location. PCI devices cannot read the CPU registers, thereby failing to detect this kind of attack. By using SMM, HyperCheck can examine the registers and report the suspicious modifications.

Furthermore, HyperCheck uses the CR3 register to translate the virtual addresses used by the kernel to the physical addresses captured by the analysis module. Since the acquiring module relies on the physical address to read the memory, HyperCheck needs to find the *physical* addresses of the protected hypervisor and the privileged domain. For this purpose, HyperCheck checks both symbol files and CPU registers. From symbol files, HyperCheck can read the virtual addresses of the target memory. HyperCheck then utilizes CPU registers to find the physical addresses corresponding to the virtual ones. Previous systems only used the symbol files to read the virtual addresses and calculate the physical addresses. Such systems can not detect attacks that modify page tables and leave the original memory untouched. Another possible way to get the physical addresses without using registers is to scan the entire physical memory and use pattern matching to find all potential targets. However, this method is not scalable or even efficient, especially since hypervisors and operating system kernels have small memory footprints.

4.3.1 Acquiring the Physical Memory

In general, there are two methods of acquiring the physical memory: software and hardware. The software method uses the interface provided by the OS or the hypervisor to access the physical memory, such as `/dev/kmem` on Linux [57] or `\Device \PhysicalMemory` on

Windows [58]. This method relies on the integrity of the underlying operating system or the hypervisor. If the operating system or the hypervisor is compromised, the malware may provide a false view of the physical memory. Moreover, these interfaces to access memory can be disabled in future versions of the operating systems. In contrast, the hardware method uses a PCI device [20, 27] or other types of hardware [17]. The hardware method is more reliable because it depends less on the integrity of the operating system or the hypervisor.

We choose the hardware method to read the physical memory. There are also multiple options for the hardware components, such as a PCI device, a FireWire bus device, or customized chipset. We selected a PCI device because it is the most commonly-used hardware. Moreover, existing commercial Ethernet cards need drivers to function. These drivers normally run in the operating system or the driver domain, each of which is vulnerable to the attacks and may be compromised in our threat model. To avoid this problem, HyperCheck puts these drivers into the SMM code. Since the SMRAM memory is going to be locked after booting, it will not be modified by the attacker. In addition, to prevent the attacker from using a malicious NIC driver in the OS to spoof the SMM driver, we use a secret key. The key is obtained from the monitor machine when the target machine is booting up and then stored in the SMRAM. The key is then used as a random seed to selectively hash a small portion of the data in order to avoid data replay attacks.

Another class of attacks is denial of service (DoS) attacks. Such attacks aim to stop or disable the device. For instance, according to ACPI [24] specification, every PCI device supports the D3 state. This means that an ACPI-compatible device can be suspended by an attacker who takes over the operating system; ACPI was designed to allow the operating system to control the state of the devices. Of course, the OS is not a trusted component in our threat model. Therefore, one possible method of attack is to selectively stop the NIC without stopping any other hardware. To prevent ACPI DoS attacks, we need an out-of-band mechanism to verify that the PCI card is not disabled. The remote server that receives the state snapshots plays this role.

4.3.2 Translating the Physical Memory

In practice, there is a semantic gap between the physical memory that we monitor and the virtual memory addressing used by the hypervisor. To translate the physical memory, the analysis module must be aware of the semantics of the physical memory layout, which depends on the specific hypervisor we monitor. On the other hand, the acquiring module may support many different analysis modules with few or no modifications.

The current analysis module depends on three properties of the kernel memory: linear mapping, static nature, and persistence. Linear mapping means that the kernel (OS or hypervisor) memory is linearly mapped to physical memory and that the physical addresses are fixed. For example, on x86 architecture, the virtual memory of Xen hypervisor is linearly mapped into the physical memory. Therefore, in order to translate the physical address to a given virtual address in Xen, we have 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 changes for different machines but remains the same on a given machine. Moreover, other operating system kernels, such as Windows [59], Linux [60], or OpenBSD [35], also have this property when they are running directly on the real hardware.

Static nature means that the contents of the monitoring part of the hypervisor must be static. If the contents are changing, then there may be a time window between the CPU changing the contents and our acquiring module reading them. This could result in inconsistencies for analysis and verification. Persistence property means that the memory used by hypervisors will not be swapped out to the hard disk. If the memory is swapped out, then we cannot identify and match any content by only reading the physical memory. We would have to read the swap file on the hard disk.

The current version of HyperCheck relies on these three properties (linear mapping, static nature, and persistence) to work correctly. Aside from the Xen hypervisor, most operating systems also hold these three properties.

4.3.3 Reading and Verifying the CPU Registers

Since the Ethernet card cannot read the CPU registers, we must use another method to read them. Again, there are software- and hardware-based methods. For the software method, one could install a kernel module in the hypervisor, and it could then obtain registers by reading directly from the CPU. However, this method is vulnerable to the rootkits, which can potentially modify the kernel module or replace it with a fake one. For the hardware method, one could use a chipset to obtain the registers.

We choose to use SMM in the x86 CPU, which is similar to a hardware method. As we mentioned earlier, SMM is a different CPU mode from the protected mode in which the hypervisor or the operating system reside. When the CPU switches to SMM, it saves the register context in the SMRAM. The default SMRAM size is 64K Bytes, beginning at a base physical address in the physical memory called the SMBASE. The SMBASE default value, following a hardware reset, is 0x30000. The processor looks for the first instruction of the SMI handler at the address $[SMBASE + 0x8000]$. It stores the processor's state in the area from $[SMBASE + 0xFE00]$ to $[SMBASE + 0xFFFF]$ [47]. In SMM, if the SMI handler issues `rsm` instruction, the processor will switch back to the previous mode (typically the protected mode). In addition, the SMI handler can still access I/O devices. HyperCheck verifies the registers in SMM and reports the result by sending it via the Ethernet card to the monitor machine. HyperCheck focuses on monitoring two registers: IDTR and CR3. IDTR should never change after system initialization. For CR3, SMM code can use it to translate the physical addresses of the hypervisor kernel code and data. The offsets between physical addresses and virtual ones should never change, as we discussed in Section 4.3.2.

4.4 Implementation

We implemented two prototypes for HyperCheck: HyperCheck-I uses QEMU full-system emulation, while HyperCheck-II runs on a physical machine. We first developed HyperCheck-I for quick prototyping and debugging. To measure the overall system performance, we

implemented HyperCheck-II on non-virtualized hardware. Both utilize the Intel e1000 Ethernet card as the acquiring module.

In HyperCheck-I, the target machine is a virtual machine that uses QEMU. The analysis module runs on the host operating system of QEMU. For the acquiring module, we placed a small NIC driver into the SMM of the target machine. Using the driver, we can program the NIC to transmit the contents of physical memory as an Ethernet frame. On the monitoring machine, an analysis module receives the packet from the network. The analysis module compares contents of the physical memory with the original (initial) versions. If a new snapshot of the memory contents differs from the original, then the module will report the event to a systems operation that can decide how to proceed. Moreover, another small program runs in the SMM and collects and sends out the CPU registers, also via the Ethernet card.

For HyperCheck-II, we used two physical machines: one as the target and the other as the monitor. On the target machine, we installed Xen 3.1 natively and used the physical Intel e1000 Ethernet card as the acquiring module. We also modified the default SMM code on the target machine to enable our system similarly to our QEMU implementation. The analysis module runs on the monitor machine and is the same as the one in HyperCheck-I. HyperCheck-II is mainly used for performance measurement.

As we mentioned earlier, we used QEMU for HyperCheck-I. QEMU is suitable for debugging potential implementation problems. However, it comes with two drawbacks. First, the throughput of a QEMU network card is much lower than that of a real NIC device. For our QEMU-based prototype, the network card throughput is approximately 10MB/s, although Gigabit Ethernet cards are common in practice. Second, the performance measurement on QEMU may not reflect the real world performance. HyperCheck-II help us to overcome these problems.

4.4.1 Memory Acquiring Module

The main task in implementing the acquiring module is to port the e1000 network card driver into the SMM to scan the memory and send it out. Normally, SMM code is one part of the BIOS. Since we don't have the source code of the BIOS, we used a method similar to the one mentioned in [33] to modify the default SMM code. Basically, it writes the SMM code in 16bit assembly and then uses a user-level program to open the SMRAM and copy the assembly code to the SMRAM.

To overcome the limitations of [33], we divided the e1000 driver into two parts: initialization and data transfer. The initialization part is complex and very similar to the Linux driver. The communication part is simpler and different from the Linux driver. Therefore, we modified the existing Linux e1000 driver to initialize the network card and only program the transferring part in assembly. The e1000 driver on Linux was changed to only initialize the NIC; it does not send out any packets. The assembly code was compiled to an ELF object file. We then wrote a small loader that can parse the ELF object file and load the code and data to the SMM.

For this implementation, the NIC driver is ported to the SMM. The next step is to modify the driver to scan the memory and send it out. HyperCheck uses two transmission descriptors per packet: one for the header and the other for the data. The content of the header should be predefined. Since the NIC is already initialized by the OS, the driver in SMM has only to prepare the descriptor table and write it to the Transmit Descriptor Tail (TDT) register of the NIC. The NIC will send the packet to the monitoring machine using DMA. The NIC driver in SMM prepares the header data and lets the descriptor point to this header. For the payload, the descriptor is directly pointed to the address of the memory that needs to be scanned. In addition, e1000 NIC supports CRC offloading.

To prevent replay attacks, a secret key is transferred from the monitor machine to the target machine upon booting. 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 large, random number

used for a one-time pad encryption and another set of serial random numbers. The serial random numbers are used as indexes for the positions of the memory being scanned. The content at these positions is then XORed with the one-time pad with the same length before starting NIC DMA. After the transmission is complete, the memory content is 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 data integrity, the SMM NIC driver stays in the SMM until all of the packet is written to the internal FIFO of the NIC. 64KB more data is added to the end to flush the internal FIFO of the NIC. Therefore, 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.

4.4.2 Analysis Module

On the monitoring machine, a dedicated network card is connected with the acquiring module. The operating system of the monitoring machine is CentOS 5.3. We run `tcpdump` to filter the packets from the acquiring module; the output of `tcpdump` is then sent to the analysis module. The analysis module written in a Perl script reads the input and checks for any anomalies. The analysis module first recovers the contents using the same secret key. Afterwards, it compares every two consecutive memory snapshots bit by bit. If they are different, the analysis module outputs an alert on the console, as we are checking the persistent and static portion of the hypervisor memory. The administrator can then decide whether it is a normal update of the hypervisor or an intrusion. Note that during the system boot time, the contents of the control data and code are changing.

Currently, the analysis module can check the integrity of the control data and code. The control data includes the IDT table, hypercall table, and exception table of Xen, and the code is the code part of Xen hypervisor. To find out the physical address 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 address of these symbols

is `virtual_address - 0xff00,0000` on x86-32 architecture with PAE. The address of the Xen hypervisor code is between `_stext` and `_etext`. HyperCheck can also monitor the control data and codes of Domain 0. This includes the system call table and the code part of Domain 0 (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 test 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 input these parameters to the acquiring module to improve the scan efficiency.

Note that these control tables are critical to system integrity. If their contents are modified by any malware, they could potentially run arbitrary code in the hypervisor level (i.e., the most privileged level). An antivirus software or intrusion detection system that runs in Domain 0 finds it difficult or impossible to detect this hypervisor-level malware because each relies on the hypervisor to provide correct information. If the hypervisor itself is compromised, it may provide fake information to hide the malware. The part of the HyperCheck that checks for the code of the hypervisor enables HyperCheck to detect the attacks that do not modify the control table but simply modify the code invoked by those tables.

4.4.3 CPU Register Checking Module

HyperCheck uses SMM code to acquire and verify CPU registers. In a product, the SMI handler should be integrated into the BIOS, or it can be set up during the system boot time. This requires the bootstrap to be protected by some trusted bootstrap mechanism. In addition, most chipsets provide a function to lock the SMRAM. Once it is locked, the SMM handler cannot be changed until reboot. Therefore, the SMRAM should be locked once it is set up. In our prototype, we used the method mentioned in Section 4.4.1 to modify the default SMM code.

There are three steps for CPU register checking: (1) triggering the SMI to enter SMM, (2) checking the registers in SMM, and (3) reporting the result. SMI is a hardware interrupt

and can only be triggered by hardware. Normally, the I/O Controller Hub (ICH), also called Southbridge, defines the events to trigger SMI. For HyperCheck-I, the QEMU emulates the Intel 82371SB chip as the Southbridge. It supports some device idle events to generate SMI. SMI is often used for power management, and Southbridge provides some timers to monitor the state of a device. If the device remains idle for a long time, it will trigger SMI to turn off that device. The resolutions of these timers are typically one second. However, on different motherboards, the method to generate the SMI may be different. Therefore, we employ the Ethernet card to trigger the SMI event.

For register checking, HyperCheck monitors the IDTR and CR3 registers. The contents of IDTR should never change after system boot. The SMM code simply reads this register by `sidt` instruction. HyperCheck uses CR3 to find out the physical addresses of hypervisor kernel code and data given their virtual addresses. Essentially, it walks through all of the page tables just 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. For example, it is `0xff000000` for Xen 32bit with PAE. The Domain 0 has the same property. The SMM code requires the virtual address range as the input, which can be obtained through the symbol file and sent to the SMM in the boot time, and afterwards checks their physical addresses. If any physical address is not equal to virtual address offset, this signifies a possible attack. The SMM code reports the result of this checking via the Ethernet card. The assembly code is just 67 LOC.

The SMM code uses the Ethernet card to report the result. Without the Ethernet card, it is difficult to reliably send the report without stopping the whole system. For example, the SMM code could write the result to a fixed address of physical memory. However, according to our threat model, the attacker has access to that physical memory and can easily modify the result. Alternatively, the SMM code could write it to the hard disk, but this can be altered by the attacker as well. Since security cannot rely on the obscurity, the only option left without a network card is to stay in the SMM mode and place the warning message on the screen. This is reliable, but the system in the protected mode

becomes completely frozen. Sometimes, this may not be desirable and could be abused by the attacker to launch Denial of Service attacks.

4.4.4 HyperCheck-II

The main difference between HyperCheck-II and HyperCheck-I is the acquiring module. We ported the SMM NIC driver from QEMU to a physical machine. Both of them have the same model of the NIC: 82540EM Gigabit Ethernet card. However, the SMM NIC driver from the QEMU VM does not work on the physical machine, and it took one week to debug the problem. We finally found that the main difference between a QEMU VM and the physical machine (Dell Optiplex GX 260) is that the NIC can access the SMRAM in a QEMU VM, but it cannot do so on the physical machine. For the HyperCheck-I SMM NIC driver, the TX descriptor is stored in the SMRAM and works well. For HyperCheck-II, the NIC cannot read the TX descriptor in the SMRAM and therefore does not transmit any data.

To solve this problem, we reserved a portion of physical memory by adding a boot parameter: `mem=500M` to the Xen hypervisor or Linux kernel. Since the total physical memory on the physical machine is 512MB, we reserved 12MB for HyperCheck by using the `mem` parameter. This 12MB is used to store the data used by SMM NIC and the TX descriptor ring. We also modified the loader to be a kernel module; it calls `ioremap()` to map the physical memory to a virtual address and to load the data there. In a product, the TX descriptor ring should be prepared every time by the SMM code before transmitting the packet. In our prototype, since we don't have the source code of the BIOS, we used the loader to load the TX descriptor.

Finally, we built a debugging interface for the SMM code on the physical machine. We use the reserved physical memory to pass the information between the SMM code and the normal OS. This interface is also used to measure the performance of the SMM code, as we will discuss in Section 4.5.

4.5 Evaluation

To validate the correct operation of HyperCheck, we first verified the properties that need to hold for us to be able to protect the underlying code, as we discussed in Section 4.3.2. We then tested the detection for hypervisor rootkits and measured the operational overhead of our approach. We have worked on two testbeds: Testbed 1 is mainly used for HyperCheck-I and also as the monitor machine for HyperCheck-II; Testbed 2 uses HyperCheck-II to produce the plotted performance overhead on the real hardware. Testbed 1 was equipped with a Dell Precision 690 with 8GB RAM and one 3.0GHz Intel Xeon CPU with two cores. The host operating system was CentOS 5.3 64bit. The QEMU version was 0.10.2 (without `kqemu`). The Xen version was 3.3.1, and Domain 0 was CentOS 5.3 32bit with PAE. Testbed 2 was a Dell Optiplex GX 260 with one 2.0GHz Intel Pentium 4 CPU and 512MB memory. Xen 3.1 and Linux 2.6.18 were installed on the physical machine, and Domain 0 was CentOS 5.4.

4.5.1 Verifying the Static Property

An important assumption is that the control data and respective code are statically mapped into the physical memory. We used a monitoring module designed to detect legitimate control data and code modifications throughout the experiments. This enabled us to test our approach against data changes and self-modifying code in the Xen hypervisor and Domain 0. We also tested the static properties of Linux 2.6 and Windows XP 32bit kernels. In all of these tests, the hypervisor and the operating systems are booted into a minimal state. The symbols used in the experiments are shown in Table 4.1. During the tests, we discovered that during boot the control data and the code changes. For example, the physical memory of IDT is all 0 when the system first boots up. But after several seconds, it becomes non-zero and static. The reason for this is that the IDT table is initialized later in the boot process.

Table 4.1: Symbols for Xen hypervisor, Domain 0, Linux, and Windows

System	Symbol	Use
Xen	idt_table	Hypervisor’s Interrupt Descriptor Table
	hypercall_table	Hypervisor’s Hypercall Table
	exception_table	Hypervisor’s Exception Table
	._stext	Beginning of hypervisor code
	._etext	End of hypervisor code
Dom0	sys_call_table	Domain 0’s System Call Table
	._text	Beginning of Domain 0’s kernel code
	._etext	End of Domain 0’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

4.5.2 Detection

To verify whether HyperCheck can detect attacks against the hypervisor, we implemented DMA attacks [16] on the Xen hypervisor and then tested HyperCheck-I’s response on Testbed 1. We ported the HDD DMA attacks to modify the Xen hypervisor and Domain 0. There were four attacks to the Xen hypervisor and two attacks to Domain 0. We also modified the pcnet network card in QEMU to perform the DMA attack from the hardware directly. The modified pcnet NIC was used to attack Linux and Windows operating systems. There were three attacks to Linux 2.6.18 kernel and two attacks to Windows XP SP2 kernel, each targeting one control table or the code. They were able to modify the IDT table and other tables of the kernel. HyperCheck-I correctly detected all of these attacks by reporting that the memory contents of the target machine had been changed.

4.5.3 Monitoring Overhead

The primary source of overhead comes from the transmission of the memory contents to the external monitoring machine. In addition, to ensure that the memory contents have

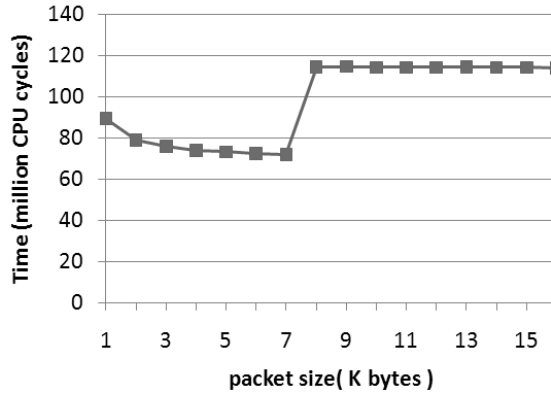


Figure 4.2: Network overhead for variable packet sizes.

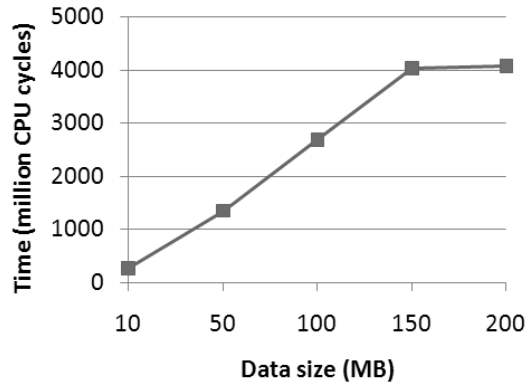


Figure 4.3: Network overhead for variable data sizes.

not been tampered with, HyperCheck must remain in SMM and wait until the NIC has finished. Otherwise, the attacker may control the OS and modify the memory contents or the transmit descriptor in the main memory while transmitting. Initially, we measured the time to transmit a single packet varying its payload size. The packet flushed out when the Transmit Descriptor Head register (TDH) was equal to the Transmit Descriptor Tail register (TDT). We calculated the elapsed time using the `rdtsc` instruction to read the time stamp before and after each operation. As expected, the time increased linearly as the size of the packet increased.

Next, we measured the bandwidth by using different packet sizes to send out a fixed

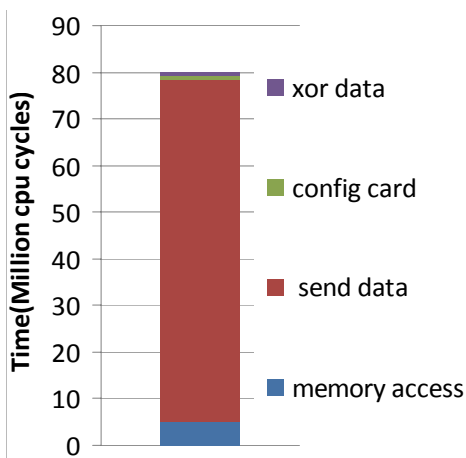


Figure 4.4: Overhead of the operations in SMM

amount of data: 2881 KB memory (the size of Xen code plus Domain 0 code). The result is depicted in Figure 4.2. When the packet size is less than 7 KB, the time required to send the data is similar to a constant value. When the packet size becomes 8KB, the overhead increases dramatically and remains high. The reason for this is that the internal NIC transfer FIFO is 16KB. Therefore, when the packet size becomes 8KB or larger, the NIC cannot hold two packets in the FIFO at the same time, and this introduces delay.

Since HyperCheck can be used to monitor different-sized hypervisors and OSes, we measured the time required to send different amounts of data. The results are illustrated in Figure 4.3. In this set of experiments, we used 7KB as the packet size since it introduced the shortest delay in our testbed. We can see that the time also nearly linearly increased with the amount of memory. In addition to PCI scanning, HyperCheck also triggers SMI interrupt every one second and checks the registers in SMM. To measure the overall overhead of entering SMM, executing SMM code and return from SMM, we wrote a kernel module running in Domain 0.

The tests were conducted on Testbed 2 (HyperCheck-II), and each test was repeated many times. Here we present the average of the results. The overall time is composed of four parts: first, the time taken to XOR the data with the secret key; second, the time to access the memory; third, the time to configure the card and switch from protected mode to

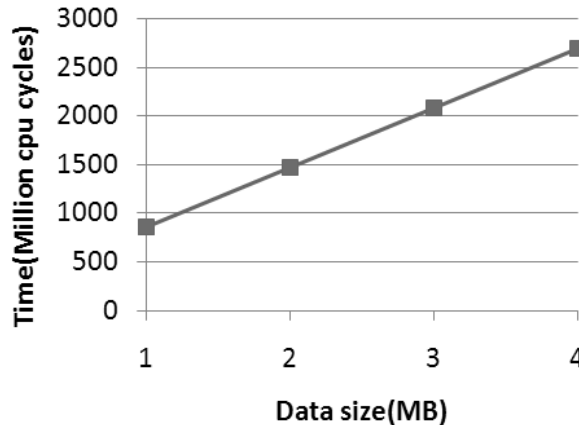


Figure 4.5: Overhead of the XOR data in SMM

SMM and back; and fourth, the time to send out the data through the NIC. To find out how much time was spent in each part, we wrote two more test programs. One was a dummy SMM code that did nothing but returns from SMM to CPU-protected mode. The other did not access the main memory but simply used the registers to simulate the verification of IDTR and CR3. We then tested the running time for these two SMM codes. From the first one, we can obtain the time for switching between protected mode and SMM and then back again. From the second, we can obtain the time for the CPU computation part of the verification of IDTR and CR3.

The results are presented in Figure 4.4. Most of the time is spent in sending the data, which totals 73 Million cycles. Next is the time spent accessing the main memory: 5.28 Million cycles. Others took a very small portion. The total time is 80 Million cycles. Since the CPU of Testbed 2 is 2 GHz, the SMM code consumes 4.0% of the CPU cycles and takes 40 ms.

We also measured the code size of our SMM code, which is just about 300 Bytes. On the monitor machine, the overhead for reading the memory contents and comparing them with the previous state took 230 ms, including 49 ms for simply comparing the data. Note that it is possible to reduce the time for reading the memory contents from the file if we use pipe or other memory sharing-based communication between tcpdump and the perl script.

Table 4.2: Time overhead of HyperCheck and other methods

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

Table 4.3: Comparison between HyperCheck and other methods

	Memory	Registers	Overhead
HyperCheck	x	x	Low
SMM	x	x	High
PCI	x		Low
TPM	x	x	High

In contrast, previous research suggests using SMM to read the memory and hashing it on the target machine. We call this the SMM-only method. To compare this approach with ours, we wrote a program to XOR the memory in SMM with different sizes. The result is shown in Figure 4.5.

The time for XOR data linearly increases with the amount of data and typically uses hundreds of millions of CPU cycles. We also compared our approach with a TPM-based approach [13], which can also be used to monitor the integrity of the kernels. The result is shown in Table 4.2, wherein HC stands for HyperCheck. We can see that the overhead of HyperCheck is one magnitude lower than the SMM-only and TPM-based methods. For SMM-only, it must hash the entire data to check its integrity, while HyperCheck only hashes a random portion of the data and then sends out the entirety using an Ethernet card. For the TPM-based method, the most expensive operation is the TPM quote, which alone took 972 ms. Note that the test machine of the TPM-based method is better than our Testbed 2. An overall comparison between HyperCheck and other methods is shown in Table 4.3. We can see that only HyperCheck can monitor both memory and registers with low overhead.

Chapter 5: Evasion Attacks

5.1 Introduction

The previous chapter describes HyperCheck, which uses SMM to monitor the integrity of hypervisors or native operating systems. Some other researchers use SMM for similar purposes, such as HyperGuard [28], and HyperSentry [19]. All of these SMM-based systems protect hypervisor integrity by periodically checking static control structures and the states of the hypervisor, then comparing them to pristine, known, and trusted states. Illegal modification to the hypervisor code or important structures can be detected.

In this chapter, however, we demonstrate that the SMI triggering and memory validation mechanism could be defeated by what we call “evasion attacks.” Indeed, if a compromised hypervisor can predict, detect, or subvert the invocation of an SMI, it can clean up the attack trace every time before the integrity measurement starts. It can then reload itself to the system after the integrity measurement ends or at a later point in time. This class of attack is feasible because, when in SMM, the CPU halts the execution of regular programs. Therefore, the protection mechanisms must be carefully crafted to consume few cycles, thus avoiding prohibitive execution costs. We take advantage of this limitation to craft attacks that exploit the small duration, frequency, and sometimes periodicity of SMI invocations.

Our analysis is two-pronged and aims to answer the following questions: (1) What are the potential ways that an attacker can evade an SMI-based integrity checking system? (2) Can we prevent these evasion attacks?

We show that an adversary can detect SMM occurrence either by directly intercepting SMI or by indirectly inferring it using time characteristics. In the first case, the attacker can modify the flow of SMI invocation by placing his/her code as a preamble to the hardware SMI. This can be accomplished by modifying APIC tables used to trigger SMI and triggering

a general interrupt controlled by the attacker instead. This direct attack requires that the adversary have access to the same SMI event trigger that the SMM-based integrity check relies on. However, the invocation of SMM can also be detected indirectly by measuring the time spent outside of the hypervisor (or general operating system kernel). To achieve this, the adversary can rely on hardware timers that remain active while in SMM. For example, the SMI detector [61] can measure the time elapsed outside of the hypervisor or the general kernel, and infer the presence of SMM. We show how an attacker can exploit this information to successfully launch an evasion attack.

Naturally, the next step is to determine how evasion attacks can be prevented. To this end, we evaluate several defense mechanisms that can prevent both types of evasion attacks. One is to hide the time spent in SMM (i.e., make the timer inaccessible or compensate for the time spent in SMM). A second potential solution is to minimize and, at the same time, randomize the scan interval. Third, the SMM code may attempt to detect the evasion attacks by scanning specific registers. We compare all of these defense strategies and evaluate their effectiveness in mitigating evasion attacks. Furthermore, we study the performance overhead of attacks and countermeasures through implementation of proof-of-concept prototypes, which we run on unmodified commodity x86 hardware.

5.2 Design of Existing SMM-based Integrity Checking Systems

In this section, we describe the design of existing SMM-based integrity checking systems. Although they tried to avoid some attacks, they still have some design faults that can be exploited by adversaries to launch evasion attacks against them. In addition, we add one more sample design of an SMM-based integrity checking system, which we refer to as “HyperSimple.” HyperSimple illustrates other pitfalls that may happen when designing such a system.

All of the SMM-based systems must follow three steps. First, they must use some kind

of SMI event to trigger SMI. Second, they must run some code while in SMM. Third, the code in SMM must run in a short time because the operating system is suspended during SMM.

HyperSimple is a sample SMM-based integrity checking system. A kernel module installed in the operating system writes *randomly* to the 0xB2 port to trigger an SMI. Then, in SMM, the code checks the integrity of the operating system kernel code and static data. Since these codes and data are static, HyperSimple compares their current state with the pristine clean state and reports any changes by freezing the machine in SMM and displaying a message on the screen. When the check is complete, the code exits SMM.

HyperGuard [28] is the first SMM-based integrity checking system. It uses a hardware timer to periodically trigger SMI. Then, in SMM, the code checks the hash of most privileged software code running in protected mode, whether it is an operating system or a hypervisor. If the hash is different from the clean state, then the system halts.

HyperCheck [41] uses a PCI network card to periodically trigger SMI. This ensures that the same SMI cannot be triggered by the adversary. In SMM, the code scans the static part of the kernel and sends the data out to a remote server where another analysis module receives the data and checks whether or not it has been tampered with.

HyperSentry [19] is the most recent SMM-based integrity checking system. It uses the Intelligent Platform Management Interface (IPMI) and baseboard management controller (BMC) presented on the server computers to *periodically* trigger SMI. During SMM, in addition to checking the integrity of the monitored code, the code also checks the reason for SMI detecting any fake SMIs invoked by the adversary.

5.3 Threat Model of Evasion Attacks

In a nutshell, an evasion attack attempts to take over a hypervisor or an operating system. Unlike regular attack methods, however, the malware is equipped with functionality to evade detection by carefully removing ‘all’ attack traces before the SMM-based integrity checking defenses examine the system. After the SMM, the malicious code reloads itself

to the system and continues its execution. An adversary can launch evasion attacks to maintain persistent control of the compromised system, even if the system is protected by a certain SMM-based integrity checking mechanism.

In this section, we present a systematic analysis of evasion attacks from an adversary’s point of view. In the meantime, we point out the design deficiencies made by the systems mentioned in Section 5.2. We then propose countermeasures in the next section to prevent such evasion attacks. We assume the process of initial compromise is not detected by the integrity monitor running in the SMM. This is highly possible considering that the current SMM integrity monitor only runs once for several seconds, and that compromising the hypervisor may take just a few instructions. Evasion attacks can be accomplished using either of the following two mechanisms: *directly intercepting SMI* or *indirectly deriving SMI*. Directly intercepting SMI means that the adversary can intercept or disable and then reissue the SMI to hide the malicious activities from being detected in SMM. HyperSimple and HyperCheck are vulnerable to these attacks. Indirectly deriving SMI means that the adversary cannot perform a direct interception and, as an alternative, tries to use system resources (e.g., timers) to derive the time properties of SMI and hide the malicious activities. HyperGuard and HyperSentry are vulnerable to these attacks. In addition, we discuss how to launch evasion attacks against randomized SMM-checking mechanisms.

5.3.1 Type I: Directly Intercepting SMI

Figure 5.1 illustrates how an adversary directly intercepts SMI and launches evasion attacks. We assume that (1) either the hypervisor or the operating system has been compromised, and (2) the adversary has root-level privileges. To intercept SMI and launch an evasion attack, the adversary needs to locate the invocation of SMI and modify that part to add some code before and after SMI.

In Step 1, the code before SMI recovers the system to the ‘clean’ state, except for the malware reloading part, and then triggers SMI. One round of SMM-based integrity checking is then executed from Step 2 to Step 5. The attacker need not change anything between Step

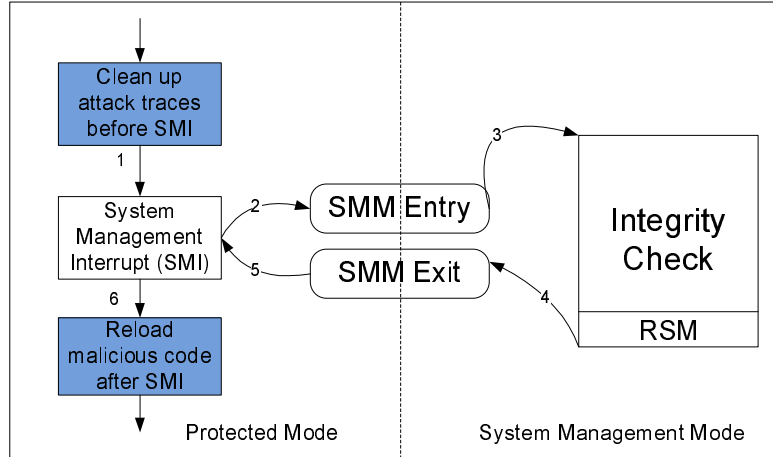


Figure 5.1: Directly Intercepting SMI. The attacker inserts a code preamble before the SMM-based integrity checks can occur.

2 and Step 5. In Step 6, the code after SMI will reload the malicious code to compromise the system again, and the original execution path then continues in the protected mode.

Attack Scenarios

We assume that an attacker can identify the SMI-triggering event by investigating the details of the SMM-based integrity-checking mechanisms or the simple enumeration of the potential SMI-triggering events. Most SMI-triggering events can either be intercepted (e.g., written to an 0xB2 port) or rerouted (e.g., a PCI device-triggered SMI). This section focuses on the SMI events that can be intercepted or rerouted by the attacker; for those events that cannot be intercepted or rerouted, the attacker can launch the indirectly-deriving SMI evasion attacks, as detailed in Section 5.3.2. This section also considers two attacking scenarios that focus on whether or not the attacker has the capability to reissue the same SMI event as the intercepted SMI event.

Scenario 1: An attacker cannot reissue the same SMI triggering event as the intercepted one.

In HyperCheck [41], the SMI-based integrity checking is triggered by a PCI network card,

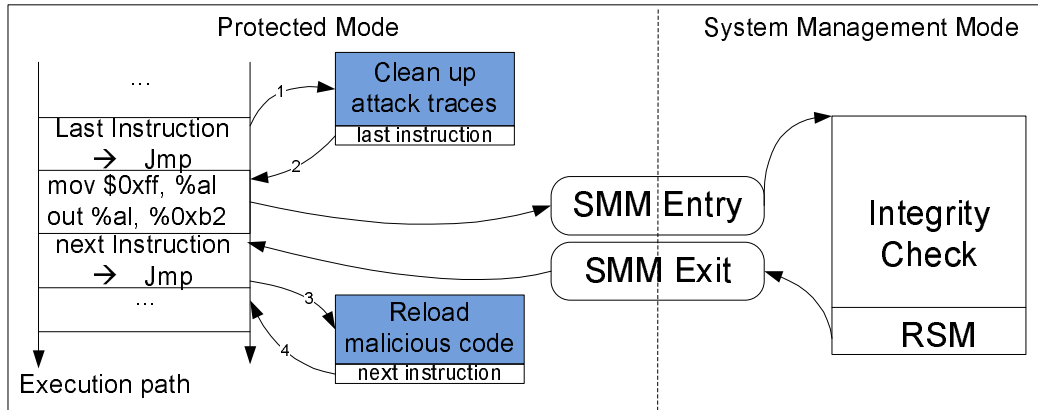


Figure 5.2: Intercept port 0xB2 SMI triggering event. This attack requires the scanning of memory to identify the call locations for 0xB2

which makes it difficult for an attacker to retrigger the same SMI. To do so, the attacker would need to find the MAC address of the network card used by HyperCheck and then use another computer to send an authenticated packet to that network card. However, the attacker can reroute the PCI interrupt to a normal interrupt and then invoke SMI by writing to the 0xB2 port.

Interrupt rerouting is possible in this example because PCI interrupt is configurable through a register. A compromised hypervisor can write to the register and change the original SMI interrupt to some normal interrupt controlled by the attacker, then trigger SMI by other means, such as writing to port 0xB2. The details are discussed in Section 5.5.2. Since PCI SMI and port-writing SMI trigger different SMI events, this attack can be easily detected if the SMM code checks the reason for triggering SMI, which has been implemented in HyperSentry [19].

Scenario 2: An attacker can reissue the same SMI triggering event as the intercepted one.

If SMM code checks the reason for triggering SMI, then the attacker must trigger the same SMI event again after disabling the SMI or rerouting the SMI to a normal interrupt. Otherwise, the integrity mechanism will notice the loss of reports from SMI and launch

further investigation. If the attacker can trigger the same SMI event, it is difficult to find out whether the SMI is triggered normally or has been intercepted and then retriggered. For HyperSimple, the SMI is triggered by writing to the port 0xB2; an adversary can easily reissue this event, as shown in Figure 5.2. To locate the invocation of the SMI, the attacker can search the following signature of the code that writes to the port 0xB2.

```
b0 ff mov $0xff, %a1
e6 b2 out %a1, %0xb2
```

The machine code for the instructions given above is 0xb0, 0xff, 0xe6, 0xb2. Therefore, the attacker can search the memory for this machine code. The attacker replaces the last instruction before the SMI triggering address with one that jumps to the code that recovers the system to its clean state, then appends the “last instruction” to the code before it jumps back to writing to the port 0xB2. Similarly, the attacker replaces the instruction following the SMI-triggering address with another jump instruction. Replacing one or two instructions with a jump is a “standard” hacking technique, the details of which can be found in [62].

5.3.2 Type II: Indirectly Deriving Periodic SMI

If the SMI-triggering events cannot be intercepted, rerouted, or reissued, then the attacker cannot successfully launch direct-interception SMI attacks. However, an attacker can still indirectly derive the SMI time information using other sources, such as hardware timers. If an attacker knows the initial SMI time, SMM duration, and SMI interval, then he/she can launch evasion attacks in a time period between two SMIs, as shown in Figure 5.3. After one SMI ends, the attacker reloads the malicious code to compromise the system. At some time before the next SMI, the attacker cleans up the attack traces.

An attacker follows three steps to derive SMI time information. First, the attacker checks whether or not the system integrity is protected by SMM. Second, the attacker finds out the SMI duration and whether it is triggered periodically or randomly. Third, the

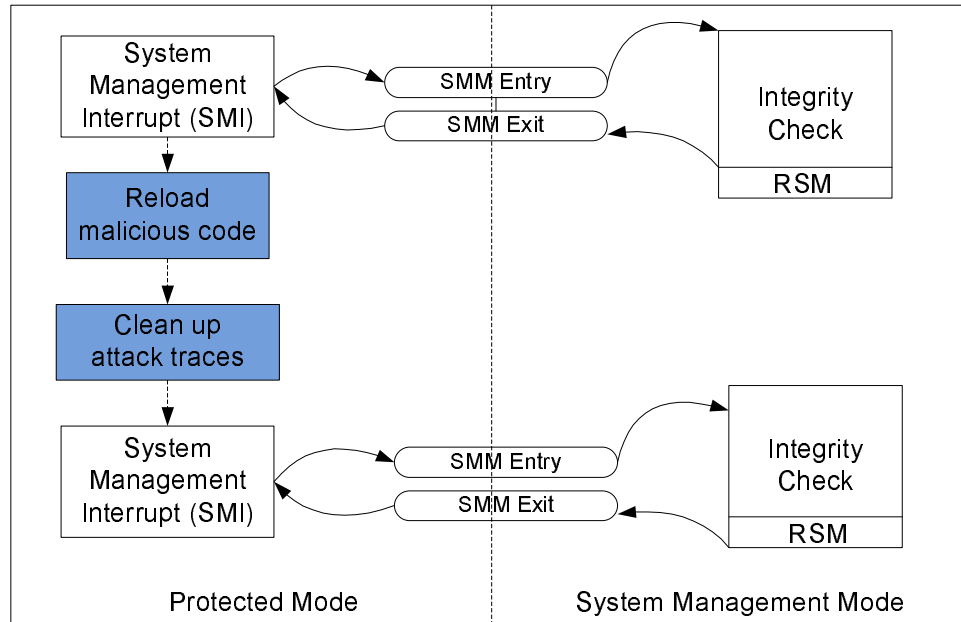


Figure 5.3: Launch of attacks between two SMIs.

attacker learns the start and end times of SMIs. With this knowledge, an attacker can clean up and reload the malicious code directly before and after the SMI event. We refer to the component that derives SMI time information as the *SMI detector*.

Presence of SMI Events

All x86 microprocessors include a CLK input pin, which receives the clock signal of an external oscillator. Starting with the Pentium, many recent x86 microprocessors include a 64-bit Time Stamp Counter (TSC) register that can be read by means of the *rdtsc* assembly language instruction. This register is a counter that is incremented at each clock signal. For example, if the clock ticks at 400 MHz, then the Time Stamp Counter is incremented once every 2.5 nanoseconds.

The basic idea of an SMI detector is to occupy the CPU for configurable amounts of time, poll the Time Stamp Counter (TSC) register for some period, then look for gaps in the TSC data. Because SMI has the highest priority, the SMI detector (which runs in

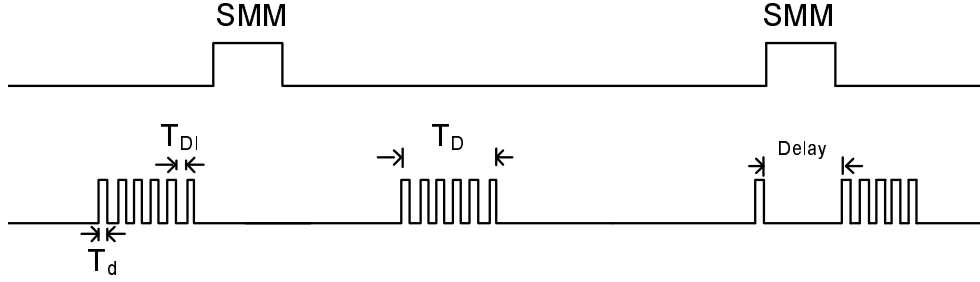


Figure 5.4: Detecting the invocation of SMI: T_D denotes the detection duration while we check the Time Stamp Counter every $T_d + T_{DI}$, where T_d is the duration time for one counter checking process and T_{DI} is the time interval between two counter checking processes.

protected mode) is frozen in SMM. As the TSC timer continues to run, any gap indicates a time when the polling was interrupted. The only reason for this would be an SMI. This idea was first mentioned by [61].

An SMI detector may detect the existence of SMI in many ways. Figure 5.4 illustrates one example. In the time period T_D , the SMI detector checks the Time Stamp Counter every $T_d + T_{DI}$, where T_d is the duration time for one counter checking process and T_{DI} is the time interval between two counter checking processes. During T_D , if SMI occurs, then T_d or T_{DI} will be dramatically increased by the delay. Thus, the SMI detector becomes aware of the presence of SMI.

SMI may be triggered by other non-integrity checking events, such as power management events; however, such events seldom occur. During a set time period, if the SMI detector observes that the system enters SMM mode multiple times, then it knows that the system is protected by SMM.

Detect SMM Duration and Interval

To launch an indirect evasion attack, an attacker must know the SMM duration and interval, as well as whether the SMI is triggered periodically.

An attacker can either run the SMI detector continuously for a time period long enough to capture several sequential SMIs, or else run the SMI detector for a short time and wait

for a constant (or random) amount of time before running the SMI detector again until a number of SMIs are captured. Both methods can derive the time duration and interval for a periodic, SMM-based checking mechanism. The continuous SMI detector can determine SMM duration and interval in less time than the random SMI detector. However, since it disables all other interrupts during its detection period, it has a higher system overhead and increases the chances of being detected by defenders.

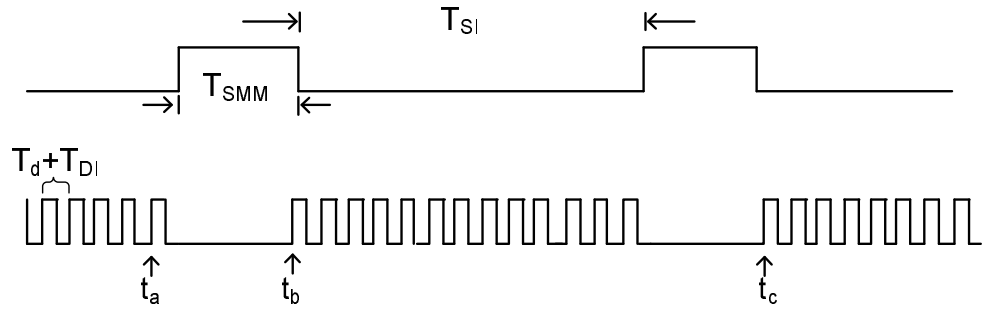


Figure 5.5: Measuring the SMM duration(T_{SMM}): T_{SI} is the time interval between two SMIs. $T_d + T_{DI}$ is the time delay between two counter polling processes when the system is not in SMM mode. t_a is the last polling time before the system enters SMM, and t_b is the first polling time after the system exits SMM.

Method 1: Continuous SMI Detector

Figure 5.5 shows how continuous SMI detectors are used to determine the interval and duration of SMMs. T_{SMM} is the time duration of SMM, and T_{SI} is the time interval between two SMIs. $T_d + T_{DI}$ is the time delay between two counter polling processes when the system is not in SMM mode. t_a is the last polling time before the system enters SMM, and t_b is the first polling time after the system exits SMM. $t_b - t_a$ is the time delay between the two continuous polling processes when the system runs in SMM. In SMM, the TSC counter continues to increase, and the SMM detector cannot read the counter value until the system exits SMM. Therefore, $t_b - t_a$ is much larger than $T_d + T_{DI}$, allowing us to derive the SMI duration $T_{SMM} = t_b - t_a - T_d - T_{DI}$. Supposing that t_c is the time when the system exits the

next round of SMM, we can obtain the SMI interval $T_{SI} = t_c - t_b - T_{SMM}$. SMM interval time T_{SI} is typically much larger than the SMM duration T_{SMM} due to the high overhead in SMM. Moreover, we know the SMI will be triggered at times $t_a + n * (T_{SI} + T_{SMM})$, where n is the round number of SMM.

Method 2: Random SMI Detector

The mechanism shown in Figure 5.5 is accurate but introduces high overhead because the SMI detector occupies all of the CPU in order to keep T_d small. In a worst-case scenario, this may cause the operating system to hang. Another method that detects SMI interval with low CPU overhead is to check SMI for a short time and then sleep for a random period, as shown in Figure 5.4. This method will miss some SMIs; however, if the SMI detector can run for a long time then the attacker can determine that the SMI interval is equal to the minimum SMI interval being detected. For example, if an SMI detector checks for a period of time and finds 4 SMIs, and the interval between the first and second SMIs is 10s, the interval between the second and the third SMIs is 5s, and the interval between the third and the fourth SMIs is 15s, then the SMI detector can determine that the SMI interval should be 5 seconds, given that it has run for a while.

After deriving the periodical SMI time information, the SMI detector finishes its job and quits. This means that the SMI detector is used only once and is difficult to be detected by defenders.

5.3.3 Type III: Avoiding Random SMI

For periodical integrity checking, an attacker can detect the SMI time information, clean up the attack space before the system enters SMM, and reload the malicious code after the system exits SMM. However, since the attacker cannot derive the next initial time of SMI, this does not work for random SMI integrity checking.

To attack randomized SMI, an attacker can try to detect when the system will exit SMM mode then immediately reload the malicious code, perform a short task (e.g., send

out one packet), and clean up the attack traces. The rationale is that the attacker does not know when the next SMI will occur; he/she, however, knows when the SMI will not occur. That is to say that a short time interval must exist between two SMIs.

To be more specific, the following requirements must be satisfied for this attack to be successful. First, the attacker should be aware of when the system will exit SMM. This information can be discovered if the SMI detector is running and reading the TSC counter at all times. An attacker can also capture some of the events that signal when the system exits SMM.

Second, the attacker should verify that the system will not enter SMM twice in a very short time. This is true because (1) frequent SMIs will increase the system overhead dramatically and block all other operations, and (2) regardless of which pseudo-random number generator is used to generate the next time interval, there is a lower threshold involved, and the probability for generating a small time interval is low. Moreover, an attacker could disable or reroute the SMI before the attack ends.

Third, the attacker's action should be quick in order to finish the current attack before the next SMI. The time involved varies for different attack scenarios. For example, an attacker may divide a large task into many small tasks that can be finished within a short time. This may limit the number of attacks that can be launched.

Fourth, the attacker must be able to avoid the detection of evasion attacks, which we will discuss in section 5.6.3.

In addition, one limitation of this attack is that it may still be captured by the defender. Since the attacker does not know the minimum interval between two SMIs, he/she can only guess and try to perform the attack as quickly as possible. If the minimum interval occurs and is smaller than his/her guess, then the attack will be detected.

5.4 Defense Strategies

Given the analysis of evasion attacks in Section 5.3, we propose two defensive strategies: (1) preventing evasion attacks from occurring, and (2) detecting evasion attacks that exist

within the system. If the defender can break the assumptions of evasion attacks, such as by hiding the SMI triggering events or by triggering SMI randomly, he/she can prevent or mitigate some of the evasion attacks. This strategy is effective towards preventing Type I (Directly Intercepting SMI) and Type II (Indirectly Deriving Periodic SMI) attacks, but not Type III (Avoiding Random SMI) attacks. On the other hand, if the defender can generate the signature of the SMI detector or discover the extra overhead of the SMI detector, he/she can detect the evasion attacks, including Type III.

5.4.1 Preventing Evasion Attacks

Type I evasion attacks can be prevented if attackers cannot detect, intercept, and reroute the SMI triggering events for SMI-based integrity mechanisms. Type II evasion attacks can be prevented by compensating time counters in SMM or by randomizing the time intervals between SMIs.

Obfuscating SMI Triggering Code

In Scenario 2 of a Type I attack, an attacker can use the code signature `0xb0, 0xff, 0xe6, 0xb2` to locate the code that triggers SMI by writing to port `0xB2`.

A defender may try to obfuscate this code. For example, the integrity monitor can add random numbers of *nop* (or other similar) operations before writing the port number using *out* instruction, or he/she can add some fake *out* instructions. In addition, the defender could use some algorithms to generate the port writing code dynamically. However, these techniques only slightly increase the workload of the attacker. Since the attacker may already compromise the OS, he/she could find out all of the *out* instructions, identify those that are actually code (not data), and then hook them. A better method for the defender is to use hardware to trigger the SMI, such as the PCI network card used by HyperCheck or the IPMI used by HyperSentry.

5.5 Implementation

Initially, we implemented an SMM-based integrity checking mechanism that could be triggered by writing to port 0xB2 or by a PCI network card. We further coded the proof of concept prototypes for the evasion attacks described in Section 5.3.

5.5.1 SMM-Based Integrity

An SMM-based integrity-checking mechanism usually consists of two modules: (1) the computer status-acquiring module and (2) the analysis module. The computer status-acquiring module is responsible for collecting computer content and status information, such as the physical memory and CPU registers of the protected machine, and for sending the collected information to the analysis module, which reviews it and validates the computer's integrity.

We implemented a prototype of an SMM-based integrity checking mechanism where the SMM code employs the PCI network card to scan the physical memory of the hypervisor or the operating system kernel, and then to send it to a remote server. Furthermore, the SMM code reads the CPU registers in the protected mode and verifies their integrity. The analysis module is implemented on a remote machine. Two machines are directly connected through a network cable. To verify the validity of the various types of evasion attacks, our SMM-based integrity checking mechanism could be triggered by two different SMI events. The first event uses port 0xB2 and a kernel module. The second SMI event is hardware-based and can be produced by the PCI network card after receiving a packet over the network. In both cases, the time intervals for triggering the SMI events can be configured in our experiments.

5.5.2 Evasion Attacks Implementation

Here, we provide a detailed implementation of the critical components of evasion attacks, including mechanisms to disable or reroute the SMI, SMI detector, and malicious code reloader.

Disable or Reroute SMI

An attacker can use the `stop_machine_run()` function provided by Linux to disable all interrupts and kernel preemptions. Other operating systems offer similar functionality. When the `stop_machine_run()` is running, the attacker can take full control of the CPU, and no other user-level programs or kernel modules can run during this period.

Now let us consider the SMI activated by writing to port 0xB2. Since the code writing to the port 0xB2 is either implemented as a user-level program or a kernel module, it will also be stopped. Therefore, the SMM-based integrity checking mechanism won't be triggered during this time period, and the attacker can safely run the malicious code. This attack method works well for software-triggered SMI only; the `stop_machine_run()` functionality cannot disable the SMI triggered by hardware such as a PCI network card.

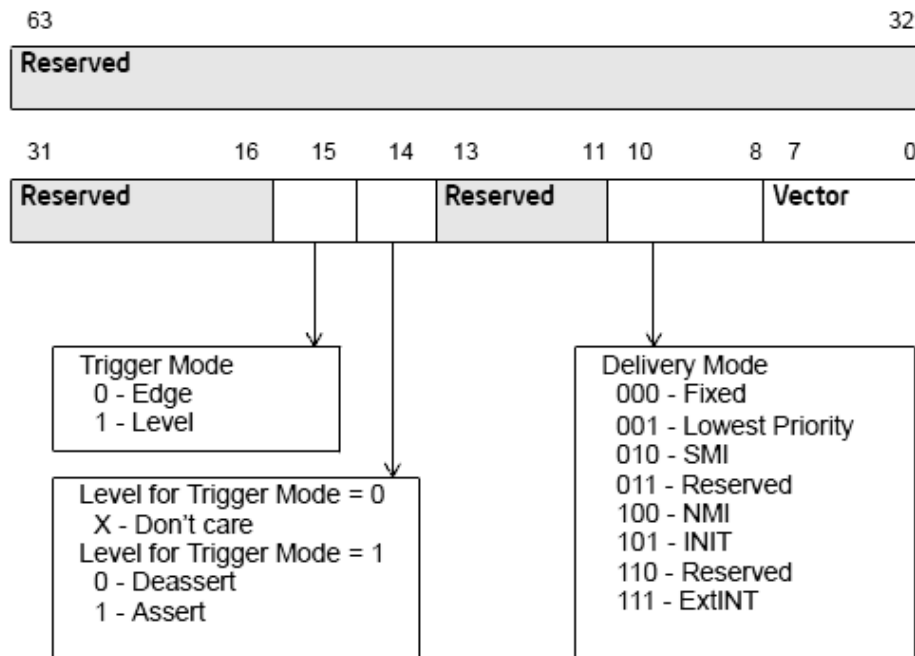


Figure 5.6: Layout of the MSI Message Data Register [1].

To extend the attack to include hardware-instigated SMIs, rather than attempting to disable the SMI triggered by PCI network card, an adversary can reroute the SMI to a normal interrupt that is already under his/her control. This will allow the insertion of a preamble to the start of the SMI routine that can be used to remove traces of the attack. But how difficult is such an attack? It appears that someone has only to rewrite one register to configure the interrupt type. This register is Message Data Register [1], used by Message Signaled Interrupts, and it is supported by PCI 3.1 and above and by PCI Express. Bits 8, 9, and 10 of the register define the delivery mode of the interrupt. 000 indicates fixed mode, and 010 indicates SMI mode. The attacker can modify this register to generate a normal interrupt then register the Interrupt Service Routine (ISR) for this interrupt. After cleaning up any traces, the attacker can reissue SMI by writing to port 0xB2.

SMI Detector

We implement an SMI detector prototype to detect the periodical SMIs triggered by a PCI network card. The basic idea is shown in the following code segment. The SMI detector measures the time interval `diff` between two time readings from the TSC counter in a busy loop, `t1` and `last`, when all other interrupt and kernel preemption are disabled. When there is no SMI, the time intervals are between $10\ \mu\text{s}$ and $18\ \mu\text{s}$, so we set the threshold as $20\ \mu\text{s}$.

Since only the SMM can stop the busy loop and, in essence, “steal” time from it, one SMI is detected when `diff` is larger than the threshold. The current time is then recorded in to the `spike` variable, and we can calculate the duration and interval of SMM. Supposing that the maximum `diff` is $diff_{max}$ and the normal `diff` is $diff_n$, then the SMM duration is $T_{SMM} = diff_{max} - diff_n$. The SMM interval is $T_{SI} = spike - spike_{last}$, where $spike_{last}$ is the time when the last SMI was detected. The attacker can calculate an accurate SMM interval only when it can identify two continuous SMIs.

```
static int smi_get_sample(void *data)
{
```



```

...

start = ktime_get(); /* start timestamp */
last = start;
do {
    i++;
    t1 = ktime_get();
    diff = ktime_to_us(ktime_sub(t1, last));

    if (diff > smi_data->threshold)
        spike = t1;

    total = ktime_to_us(ktime_sub(t1, start));
    last = t1;
} while (total <= 1000*smi_sample_ms);
...
}

```

Although we could potentially keep the loop busy running for a long time, the SMI detector relies on disabling all interrupts except for SMI. This can cause the system to hang if it remains running for too long. (We will later quantify the time-frame of “long” for commodity x86 systems). Therefore, the SMI detector cannot run continuously; instead, it creates sampling time periods while “sleeping” in between. We define the total time for a busy loop run, our sample duration, as `ms_per_sample`. The time between two samples is denoted with `ms_between_samples`. As indicated by the name, this unit is in milliseconds. We show how these two parameters can be adjusted to evaluate the detection performance and overhead of the SMI detector. Due to real-time scheduling and permission requirements,

we implemented the SMI detector as a kernel module. Although running it as a user-level process is possible, it may not yield the same detection results due to potential scheduling time delays and the lack of privileges. Moreover, some hardware timers (e.g., Real Time Stamp Counter) may be inaccessible to user-level processes.

Furthermore, the *stop_machine_run()* function in the SMI detector disables all software and hardware interrupts aside from hardware SMIs. Therefore, the SMI detector cannot be used to detect SMIs triggered by software, such as writing to port 0xB2. Instead, for a port 0xB2 writing triggered SMI, the attacker can search the code signature in the memory. In our experiment, we searched the code signature 0xb0, 0xff, 0xe6, 0xb2 in the kernel memory of CentOS 5.5. We identified only one instance of the code signature, which is the one that triggers the SMI.

Reload of the Malicious Code

It is challenging for an attacker to regain control of the system after the SMI checking is complete. The system has to be considered “untampered with,” or else the integrity checker would have raised an alarm. How can the attacker reinsert himself into the normal execution? The answer to this question lies in the limitations of the SMM in monitoring all control flow decisions inside the hypervisor or the underlying software in general. A sophisticated attacker can carefully place the attack code by altering the control flow decision of regular programs. Such small changes are difficult to detect. For example, *return_to_libc* or return-oriented rootkits [63,64] can be used to keep the reloader stealthy. An attacker can compromise the stack of one running process and use return-oriented attacks to ensure that the process compromises the system again the next time it runs. As explained in [63,64], return-oriented attacks are Turing-complete and can perform any functionality, including modifying the system call tables or the Interrupt Descriptor Tables.

We tested the loading and unloading times of a famous user space, Linux keylogger LKL [65]. After running 100 times, the average loading time was 1.232 μ s, and the average unloading time was 1.637 μ s. In total, it was almost 3 μ s. The actual running time depends

on the attacking code.

5.6 Evaluation

Using the previously-mentioned prototypes, we measured and analyzed the performance and system overheads of evasion attacks. Our experimental results show that the evasion attacks are effective in evading the existing SMM-based integrity checking solutions, such as HyperCheck [41] and HyperSentry [19]. In addition, their stealthiness depends on the amount of resources available. Finally, we show that the introduced attacks can be detected or prevented by applying the proposed defense mechanisms that are explained in detail in Section 5.4.

In all of our experiments, we used a testbed consisting of a Dell Optiplex GX 260 with one 2.0 GHz Intel Pentium 4 CPU and 512MB memory. Xen [22] 3.1 and Linux 2.6.18 were installed on the physical machine, and the Domain 0 is CentOS 5.4.

5.6.1 Performance Analysis

First, we evaluate the performance of the SMI detector used by the attacker. We then measure the detection probability of the SMI detector.

System Overhead of SMI Detector

The C source code of our SMI detector is 7660 bytes. After compiling, the size of the kernel module is 103462 bytes, but the .text section is only 1376 bytes, and most of the remaining parts are the kernel library called by the SMI detector. We also studied the system overhead of the SMI detector using a Java Micro-benchmark, CaffeineMark 3.0 [66]. The CaffeineMark contains a series of tests that measure the speed of Java programs running in various hardware and software configurations. The score for each test is proportional to the number of times the test was executed, divided by the time taken to execute the test. Since CaffeineMark uses an internal scoring metric, it is only useful for relative comparisons.

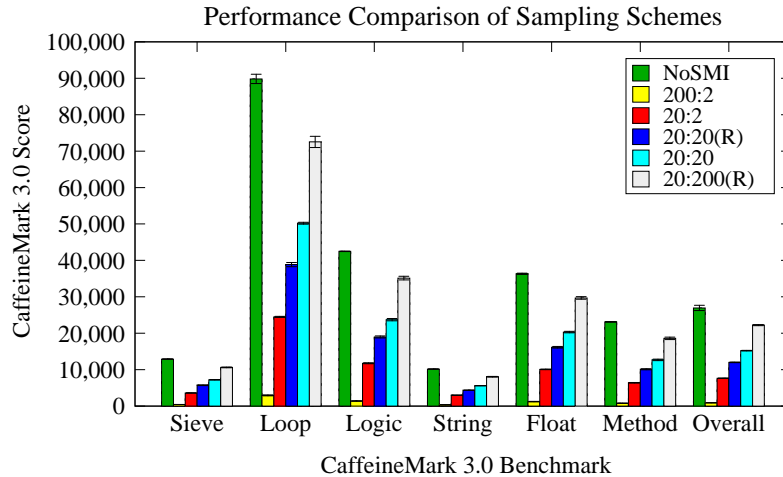


Figure 5.7: CaffeineMark 3.0 Micro-benchmarks of System Performance. The legend notation is “sample duration:check interval” in milliseconds where (R) denotes random check interval in $[1, xxx]$. The longer and more frequent the sampling, the more impact it has on system performance.

Figure 5.7 shows the execution time results. The Sieve test is the classic sieve of Eratosthenes that finds prime numbers. The Loop test uses sorting and sequence generation to measure the compiler optimization of loops. Logic tests the speed of executing decision-making instructions. String tests string concatenation and search. The Method test executes recursive function calls. The Float test simulates a 3D rotation of objects around a point. The Overall Score combines the scores of all of the tests.

In our experiments, the PCI network card triggered the SMI every 10 seconds. We adjusted two parameters: the SMI sampling duration (`ms_per_sample`) and the check interval between SMI samplings (`ms_between_samples`). Figure 5.7 shows results for the different settings where `ms_per_sample` was set to 20 or 200 ms and `ms_between_samples` were set to 2, 20, or 200. For example, 20:200(R) means that the SMI sampling duration was 20 ms, and 200(R) means that the SMI-sampling time interval was set to a random value between 0 and 200 ms. (R) stands for random.

For each setting, we ran the test 10 times, and the error bars in Figure 5.7 indicate 95% confidence intervals. System performance is at its best when no SMI detector is running; it

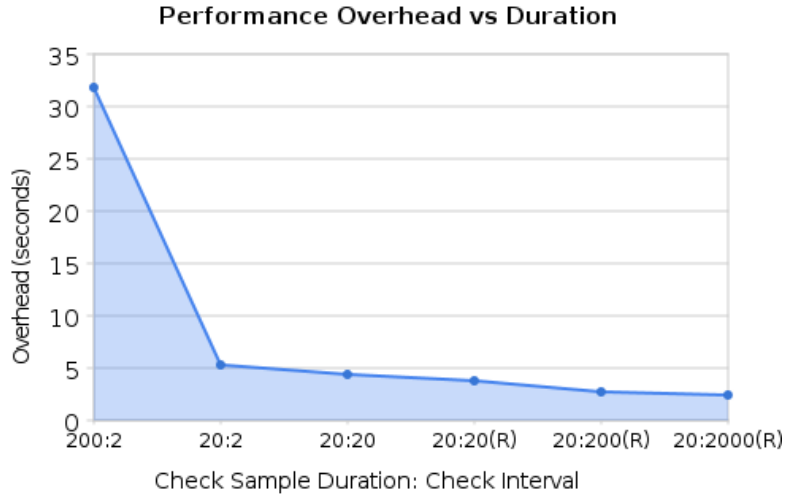


Figure 5.8: System Overhead of SMI Detector

Table 5.1: System Overhead of SMI Detector

Sampling	2:2	2:20	2:20(R)	2:200(R)	2:2000(R)	No SMI
Overhead(s)	2.379	2.389	2.343	2.381	2.347	2.332

goes down as the SMI sampling duration increases, and it goes up as the SMI sampling time interval increases. These results are consistent with implementation expectations, since the SMI detector disables all normal interrupts and kernel preemption.

We also used a macro-benchmark to test the system overhead using the “tar” command to compress a large file. Figure 5.8 shows the average time needed to finish the command when the SMI detector runs using different settings. We ran the SMI detector 20 times for each setting, excluding the first two runs to remove unrelated initialization costs.

We can see that the system overheads decrease when the SMI sampling intervals increase. When `ms_per_sample` is set to 200 ms and `ms_between_samples` is set to 2 ms, the file compression can be finished in 31.825 seconds, and we can see the dramatic slowdown of the system. We also tested “2000:2” in our experiments, but the system halted.

Table 5.1 shows the results when we set the SMI sampling time to 2 ms and the SMI

sampling interval to 2, 20, 2000, and 2000 ms. We found that the overhead was not significantly affected as we decreased the checking intervals between SMI samples. This means that if the attacker reduces the SMI sampling time to a small number, it won't have an impact on the system performance. However, increasing the checking interval will dramatically decrease the detection probability of the SMI events.

Detection Probability of SMI Detector

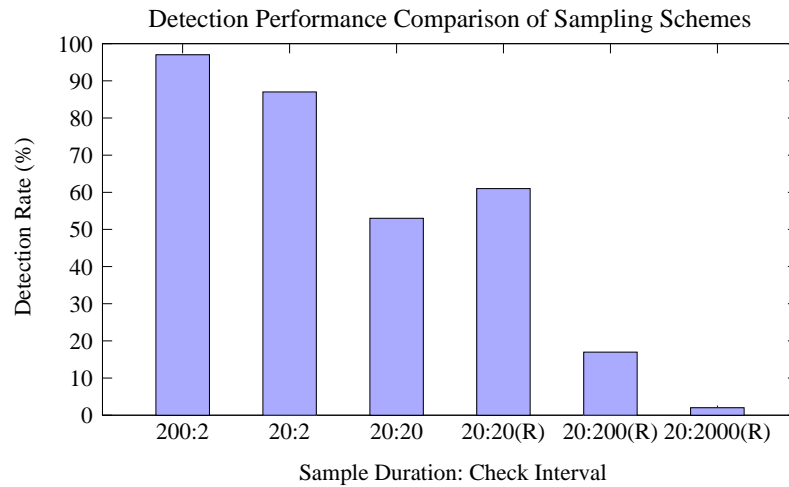


Figure 5.9: SMI Detector's Detection Probability.

Figure 5.9 depicts the SMI detection probability of an SMI detector, which is the percentage of detected SMIs among all SMIs that were triggered within the duration of the experiment. In the case of a 200:2 scenario, which means a 200ms running time with a 2ms inactivity time, almost all of the SMI events can be identified. As expected, the detection probability decreases when the sample duration decreases and the check interval increases. This is because, during the interval, the SMI detector is not active and can miss the SMIs. When the duration and interval ratio was 20:20, the randomized interval setting detected more SMIs than the fixed interval setting. Note that an attacker can still evade detection but at a loss of running time; each time that his/her code runs, the probability of being detected increases, leading to eventual detection. In all of these experiments, the total

number of SMIs is 50 in order to maintain statistical significance and reduce experimental noise.

5.6.2 Detecting Attacks

After evaluating the effectiveness of the attacks, we then evaluated the effectiveness of the defense. Since Type I and Type II evasion attacks can be easily prevented by guarding the SMI triggering events, checking SMI reasons, and using random SMIs, we focused on Type III evasion attacks, which try to avoid random SMIs.

Type III evasion attacks can be further divided into two subtypes. The first subtype tries to detect the return from SMM by using the techniques similar to SMI detector and then launches the attack. We refer to this one as “targeted evasion attacks (TEA).” The second subtype does not detect the return from SMM; it randomly launches an attack and tries its luck. We refer to this as “non-targeted evasion attacks (NTEA).”

We simulated these two attack types by using two programs written in Matlab. The results for TEA are shown in Figure 5.10 and Figure 5.11. Both figures indicate that the detection probability (Y axis) of the TEA increases when the duration of the attack (X axis) increases. The total number of tests was 4,000,000 for both tests. The SMM duration was set to 40ms. For Figure 5.10, the line with the plus sign indicates that the SMM checking interval is a uniform distribution between 1 to 5,000 ms; the line with the circle sign indicates that the SMM interval is uniformly distributed between 1 and 10,000 ms. For Figure 5.11, the line with the plus sign indicates that the SMM-checking interval is a normal distribution with a mean of 1000 (ms) and a standard deviation 500 (ms); the line with the circle sign indicates that the SMM interval is a normal distribution with a mean of 2500 (ms) and a standard deviation of 500 (ms). Both figures confirm that the detection probability will increase when the attack duration increases and the SMM interval decreases.

The results for NTEA are shown in Table 5.2 and Table 5.3. From Table 5.2, we can see that the detection probability is mostly determined by the SMM checking intervals. The

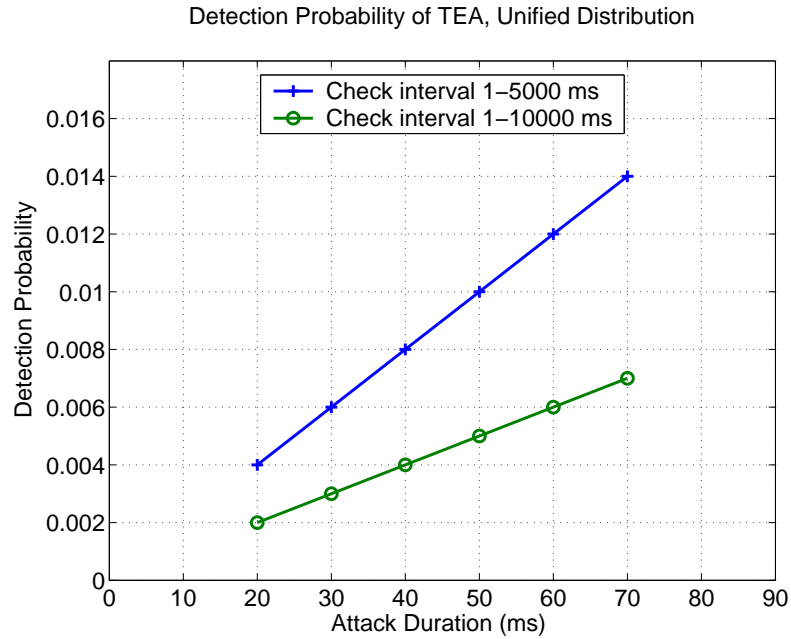


Figure 5.10: Detection Probability of TEA, unified distribution.

attack intervals do not affect the detection probability too much. From Table 5.3, we can see that detection probability dropped linearly when the SMM checking interval increased linearly. This differs from TEA, where the SMM is normally distributed. In that case, the detection probability drops at a logarithmic scale. In these tests, the SMM checking duration is 40ms, and the attack duration is 30ms. The total number of SMM checks was 100,000.

Table 5.2: Detection probability of NTEA, both intervals are uniformly distributed between 1 and the number in the column.

SMM interval(ms)	1000			
attack interval(ms)	500	1000	2000	3000
detection probability	5.40%	6.00%	5.80%	6.20%

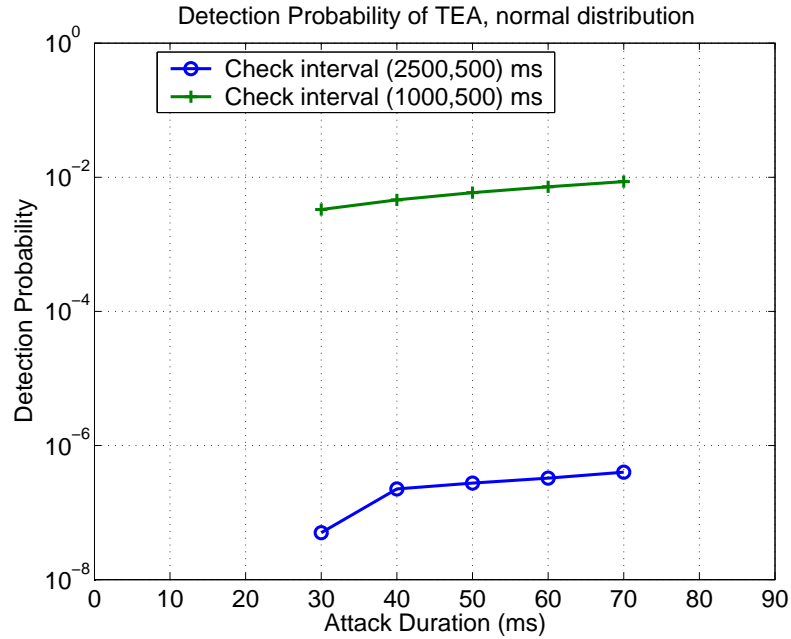


Figure 5.11: Detection Probability of TEA, normal distribution

Table 5.3: Detection probability of NTEA, SMM intervals are normally distributed between 1 and the number in the column.

SMM interval(ms)	1000	2500	5000	10000
detection probability	2.40%	1.40%	0.80%	0.25%

Compensating Time Counter in SMM

Now consider the attacks using TSC timers to detect SMI. Since the TSC timers will not stop when the processor enters SMM, the SMI detector in Type II and III evasion attacks can detect SMI and derive SMI timing information from the TSC data. One solution is to compensate for the time discrepancy when the processor runs in SMM and protected mode. SMM could modify the TSC so that the timer counter delay caused by SMM is removed. To do so, the defender can use *rdmsr* and *wrmsr* instructions to read and write TSC. When the system first switches into SMM, the SMM code records the current TSC (we call it “starting TSC”). Then, before exiting SMM, the code can modify the TSC (we

call it “ending TSC”) to be the same as or marginally smaller than the starting TSC. In this way, the SMI detector cannot detect the SMM by reading the TSC. However, this must be done carefully. For example, if the ending TSC is modified to be the same as the starting TSC, then there is still a delay for switching from the protected mode to SMM, which is not compensated for. The SMI detector may detect this delay and then detect the SMI.

Evasion attacks could be prevented if the defender can modify all potential hardware resources and make them transparent to the hypervisor or operating system when the processor enters SMM. However, current x86 hardware does not meet this requirement, and it is difficult to achieve cooperation from all impacted hardware manufacturers. In addition, it is difficult to enumerate all of the hardware components that may be used to indirectly derive SMI information. For example, there are many hardware timers on the system, such as Real Time Clock (RTC), Programmable Interval Timer (PIT), High Precision Event Timer (HPET), and ACPI Power Management Timer. Some of them cannot be changed, even in the SMM code, so their timers may be used by the attacker to detect the presence of SMI.

Randomizing SMI Time Intervals

The most simple and effective mechanism for defeating Type II evasion attacks is to randomize the time intervals between SMIs. An attacker will then be unable to predict the next occurrence of the SMI and will fail to accurately launch evasion attacks. However, this mechanism alone cannot prevent Type I evasion attacks, during which an attacker can intercept the SMI regardless of when it is triggered. It cannot prevent Type III attacks either.

To prevent Type III evasion attacks, the defender must be careful to choose the pseudo-random number generator (PRNG) functions and the parameters for the minimum and maximum time intervals. When the minimum interval is very small and occurs frequently, it becomes more difficult for the attacker to launch Type III attacks. Since attackers may disable or reroute SMI, the defender should be alerted when some SMI events are missed.

5.6.3 Detecting Evasion Attacks

The defender can detect evasion attacks by searching their attack traces. For the attacks in Scenario 1 of Type I, the SMM-based integrity mechanism could check the SMI triggering events and make sure that they can only be triggered by the originally-designed events. If the SMI triggering reasons are different, then an evasion attack can be detected.

Evasion attacks may be detected by searching the signatures of evasion attack code in the memory. The Type II and Type III attacks require the help of SMI detectors, which reside in the memory at the beginning of the Type II attacks and throughout Type III attacks. The code for reloading malicious code will reside in the memory at all times; otherwise, after all of the attacking trace is cleaned up, the attack cannot be resumed.

Checking SMI Events

HyperCheck [41] only checks the memory integrity of the kernel and the CPU registers; it does not check the reason for SMI being triggered. Therefore, an attacker can disable or reroute the SMI triggered by a PCI network card and later trigger the SMI by writing to port 0xB2, as described in the Scenario 1 of Type I attacks.

To prevent such an attack, the SMM code should check the SMI triggering events to ensure that it should only be triggered by a specific SMI event. HyperSentry [19] implemented this defense mechanism. An attacker can still disable or reroute the SMI, but it can be easily detected due to the lack of reporting response from the integrity-checking mechanism. However, if an attacker can discover how to replicate the same SMI event after rerouting the original SMI event (e.g., writing to port 0xB2), then he/she can defeat this defense mechanism.

Checking Kernel Module Integrity

Type I attacks can be detected by checking the integrity of the code-triggering SMI. For example, suppose that SMIs are triggered by writing to port 0xB2 and that the triggering code exists in a kernel module. To defeat SMI-intercepting attacks, the SMM code should

store a hash of the pristine kernel modules and check their integrity during SMM. This mechanism can force the attacker to remove the jump instruction before the SMI is invoked to avoid being detected by the SMM. Thus, the attacker cannot simply modify the kernel module and add two jumps before and after SMI triggering code.

Detecting SMI Detector and Reloading Code

If a defender can grasp the SMI detector or reloading codes, he/she may generate a signature and use it to detect whether there is any malicious code in the memory to help launch evasion attacks. This defense mechanism has two limitations. First, the attacker may obfuscate the SMI detector code in order to hide the SMI detector. The attacker can use similar obfuscation techniques as those used by defenders to obfuscate the SMI-triggering code. It is difficult for defenders to generate a complete set of signatures. Second, it is difficult to detect an SMI detector implemented as a loadable kernel module for a third-party device driver whose signature is unknown or unavailable. Moreover, it is still a challenge to check the integrity of the dynamic parts (e.g., stack and heap) of the kernel.

Chapter 6: Hardware-assisted Isolation

Previous chapters introduce a hardware-assisted method of detecting attacks against operating system integrity. A complementary method is to divide the tasks into two categories, trusted and untrusted, and to then isolate these tasks into two separate environments. In this chapter, we introduce such a method.

6.1 Introduction

Desktop computers are being employed nowadays for multiple tasks ranging from pleasure to business: web browsing, online gaming, and social web portals are examples in the former category; online banking, shopping, and business emails belong in the latter. Unfortunately, modern software has a large and complex code base that typically contains a number of vulnerabilities [67]. To make matters worse, modern desktop applications usually operate on foreign content that is received over the network. Current operating system (OS) environments offer user- and process-level isolation for different activities; however, these levels of isolation can be easily bypassed by malware through privilege escalation or by other attacking techniques. Researchers have pointed out the need for trustworthy environments where, based on context and requirements, the user can segregate different activities in an effort to lower risk by reducing the attack space and data exposure.

To this end, there is an ongoing effort to employ virtual machine monitors (VMMs, also referred to as hypervisors) to isolate different activities and applications [4,5,9–14]. As long as the virtual machine monitor is not compromised and there is no exposed path or covert channel between the different environments, applications in different VMs remain isolated. However, their widespread adoption has attracted the attention of attackers towards VMM vulnerabilities [15] and the number and nature of attacks [16,68] against the hypervisors

are poised to grow. Researchers have noticed this problem and have begun to improve hypervisor security [14, 19, 41].

An alternative to software isolation is hardware isolation: in many military and civilian installations users have to use multiple physically-isolated computers, merely switching controls and displays. Although attractive in terms of isolation, hardware increases the operational and maintenance cost because it requires more space, cooling, and energy. It is inflexible and cannot support the current need for a range of trusted environments. Moreover, it is inconvenient for users to switch between two computers to finish their tasks. Multi-boot supports the installation of multiple OSes on the same machine and uses a boot loader to choose between the OSes. Unfortunately, it is inconvenient and time consuming to shutdown one OS and boot up another. For instance, Lockdown [25] combines a hypervisor with ACPI S4 Sleep (also known as hibernation or Suspend to Disk) to provide a secure environment for sensitive applications. However, the switching latency is still too long, in many cases more than 40 seconds, rendering the system difficult to use in practice.

In this chapter, we attempt to tackle the secure OS isolation problem without using a hypervisor or any mutable shared code. We designed a firmware-assisted system called *SecureSwitch*, which allows users to switch between a trusted and an untrusted operating system on the *same* physical machine with a short switching time. The basic input/output system (BIOS) is the only trusted computing base that ensures the resource isolation between the two OSes and enforces a trusted path for switching between the two OSes. The attack surface in our system is significantly smaller than hypervisor- or software-based systems; we can protect the integrity of the BIOS code by using a hardware lock (e.g., BIOS_CNTL register [23] in Intel ICHs) to set the BIOS code as read-only, or by using TPM to verify the integrity of the BIOS code. Furthermore, our system guarantees a strong resource isolation between the trusted and untrusted OSes. If the untrusted OS has been compromised, it still cannot read, write, or execute any of the data and applications in the trusted OS.

6.2 Thread Model and Assumptions

Our system operates under the assumption that an adversary can subvert the untrusted OS using any known or zero-day attacks on software applications, device drivers, user-installed code, or operating system. We assume that the attacker cannot access the physical machine or launch local physical attacks, such as opening the case or removing a hard disk or a memory DIMM.

An adversary may launch various attacks against the trusted OS after compromising the untrusted OS. A data exfiltration attack aims at stealing sensitive data from the trusted OS. Furthermore, the adversary may modify the code of trusted OS and compromise its integrity. In a fake OS attack, a sophisticated attacker can create a fake trusted OS environment, which is fully controlled by the attacker, and deceive the user into performing sensitive transactions there. An attacker can perform a denial-of-service (DoS) attack against the trusted OS by crashing the untrusted OS; however, such attacks can be easily detected and resolved. Therefore, we do not prevent DoS attacks against our system.

We assume that the trusted OS can be trusted when the BIOS boots it up, but this does not mean that the trusted OS is bug-free. In other words, the trusted OS may be compromised from network attacks using vulnerabilities within the OS or the applications. There are several mechanisms to alleviate these network attacks; however, they lie beyond the scope of this paper.

We assume that the BIOS code, including the option ROMs on devices (e.g., video cards), does not contain vulnerabilities and can be trusted. The operating system must support ACPI S3 sleeping mode, which has been widely supported by modern OSes, such as Linux and Windows. Our system does not require hardware virtualization support (e.g, Intel VT-x or AMD-V).

6.3 SecureSwitch Framework

Figure 6.1 illustrates the overall architecture of the SecureSwitch system. Two OSes are loaded into the RAM at the same time. Instead of relying on a hypervisor, we modify the BIOS to control the loading, switching, and isolation between the two OSes. Commercial OSes that support ACPI S3 can be installed and executed without any changes. We require that the computer have at least two DIMMs and two hard disks and that the two OSes be installed on each hard disk.

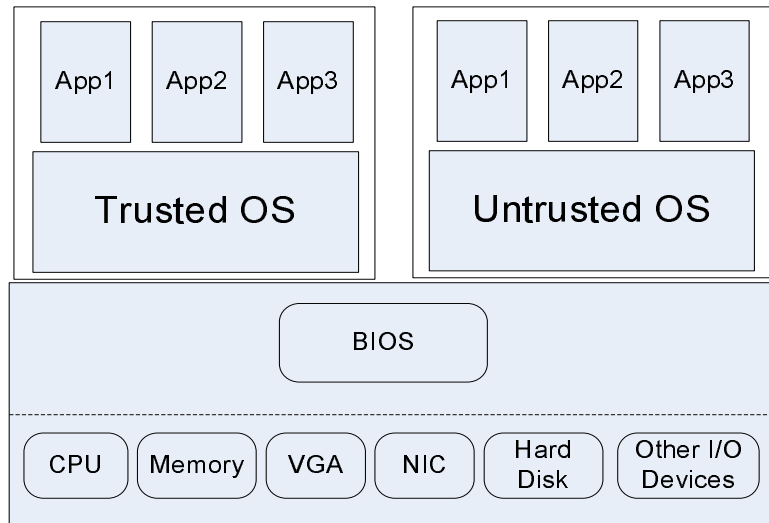


Figure 6.1: Architecture of SecureSwitch System

Secure Switching 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 and the untrusted OS second. In the OS switching stage, the system can suspend one OS and then wake up another. In particular, it must guarantee a trusted path when the system switches from the untrusted 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

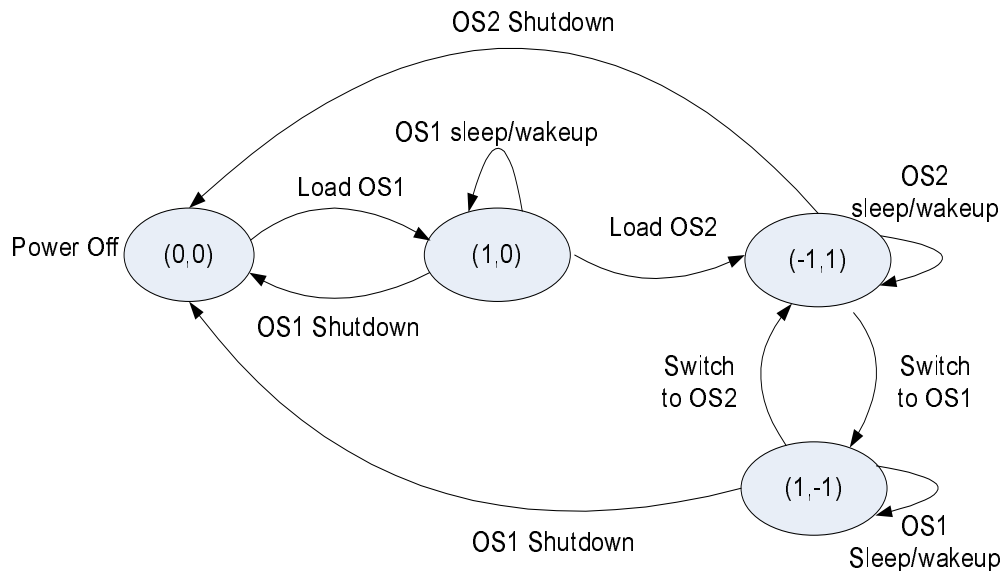


Figure 6.2: State Machine for OS Switching.

cannot access any data or execute any code on the trusted OS.

6.3.1 Secure OS Loading

Figure 6.2 shows the state machine for loading and switching between two operating systems in the system. Two variables are maintained to denote the respective states of two OSEs. In each parenthesis, the first and second number records the state of OS1 and OS2 respectively. 0 means the OS has not been loaded into the system; 1 means the OS has been loaded and is currently running; -1 means the OS has been loaded but switched out (i.e. is in sleep mode). For instance, (-1, 1) means that OS1 has been loaded but switched out and that OS2 is running. Since the system has only four valid states, we could use two bits to record the current state of the machine.

When the machine is powered off, the system state is (0,0). After the system is powered on, the BIOS always boots up the trusted OS (OS1) first. When loading the trusted OS from non-volatile memory (e.g., hard disk, USB), BIOS constrains the trusted OS to use only half of the physical RAM. The trusted OS is not aware of the existence of the second half

of the physical memory. The trusted OS can be shut down or can perform sleep/wakeup. Before loading the untrusted OS (OS2), the trusted OS is first suspended. The BIOS then boots up the untrusted OS into the second half of the physical RAM, which has no overlap with the memory used by the trusted OS. At this point, both OSes have been loaded into the memory while the untrusted OS is running and the trusted OS is suspended. When either OS is shut down, the whole system will be shut down.

6.3.2 Secure OS Switching

If a user wants to switch from the untrusted to the trusted OS, the untrusted OS will be suspended first and then the system will wake up the trusted OS. Similarly, the user can switch back from the trusted to the untrusted OS. To save power energy, the user can still sleep and wake up the same OS.

Stateful vs. Stateless Trusted OS

When the system switches into the trusted OS, there are two options for restoring OS context.

- *Stateless mode*: Each time the system switches into the trusted OS, it starts from a pristine state. A copy of the trusted OS in its pristine state is maintained either on the hard disk or in a reserved memory area.
- *Stateful mode*: When the trusted OS is switched in, it resumes from the system context wherein it was last switched out. All states of the trusted OS can be saved in the memory or on the hard disk.

The stateless mode does not save any system state when the OS is switched out. It can mitigate the impacts of network attacks to the trusted OS since it will start from a pristine state that has not been compromised. The drawback is that the user loses the system context, so it cannot resume a previous session or task within the trusted OS. Moreover, an adversary can easily fake a trusted OS environment if it knows the pristine state of the

OS. In a stateful mode, since all of the system states are saved and can be restored, a user may resume sessions and tasks within the trusted OS. However, when the trusted OS has been continuously used for a long time, the risk of being compromised from network attacks increases.

Protecting Control Variables

In Figure 6.2, two variables record the system's current states. The system also uses these as control variables to control the system actions on OS loading and switching. It is critical to protect these control variables from being manipulated by an attacker.

Malicious code may manipulate the two variables to launch a fake OS attack. For instance, when the untrusted OS is running with system state $(-1,1)$, the user wants to switch to the trusted OS. The control variables should first be updated from $(-1,1)$ to $(1,-1)$. To launch the attack, the untrusted OS can keep the control variables unchanged and then suspend itself. When the system reads the control variables, it will simply wake up the untrusted OS, which may create a faked trusted OS environment by installing a virtual machine similar to the trusted OS on the untrusted OS [69]. Since all of these actions may be transparent to the user, the attacker can deceive the user into a fake trusted OS to perform sensitive transactions.

In our prototype, we define two system flags to serve as the control variables. We could prevent the fake OS attacks by using some common hardware components to indicate the value of a system flag. Thus, the system flags can only be manually set by the user and cannot be configured by any software. The design details are described in Section 6.4.2.

Trusted Path

The secure switching consists of two sequential steps: *OS1 Suspend* and *OS2 Wakeup*. In x86 architecture, the suspend step is performed entirely by the operating system without involving the BIOS; the wakeup step is initiated by the BIOS, which then hands over control to the operating system. Since a compromised untrusted OS can gain full control of the

suspend step, it may fake an OS suspend (e.g., power off the monitor) and deceive the user into a fake trusted OS.

To prevent such fake OS attacks, our system must ensure a trusted path that guarantees that the system enter the real trusted OS when it switches to the trusted OS. In our system, the BIOS and some hardware are the only components that we can trust to enforce the trusted path. We must guarantee that one OS will be truly suspended in order to trigger the BIOS to enforce the trusted path.

6.3.3 Secure OS Isolation

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, chipset, 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.

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, our system includes a novel 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 (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. Details regarding this are included in Section 6.4.3.

I/O Device Isolation

Typical I/O devices include a 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 its own *volatile memory* (e.g., NIC, VGA), we must guarantee that the untrusted OS cannot obtain any information remaining 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 the OS is switched in, the device buffer can be reloaded from the RAM or hard disk. 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 information 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/woken. However, this method will increase the switching time dramatically due to costly encryption/decryption operations. Another solution is to use the RAM disk to save temporary sensitive data for the trusted OS because the untrusted OS cannot access the trusted OS's memory. Details regarding this can be found in Section 6.4.3.

6.4 System Design

We combine the BIOS and the standard ACPI S3 sleep to enforce resource isolation between the two OSes. BIOS is the control center and the only trusted computing base to enforce a trusted path during the OS switching process. The integrity of the BIOS can be protected by the Trusted Platform Module (TPM) or by a signed signature from vendors.

The BIOS relies on two flags in the OS loading and switching process. The *OS_FLAG* indicates which OS (and corresponding resources) should be started; the *BOOT_FLAG* indicates whether the untrusted OS is being woken up or loaded.

6.4.1 Loading Two OSes

In the OS loading stage, the system loads both OSes in the RAM. To enforce RAM isolation without a hypervisor, our system requires that the motherboard have at least two DIMMs and two hard disks, and it assigns one DIMM and one disk to each OS. When the computer boots up from a power-off state, the BIOS first loads the trusted OS using only one DIMM. Because BIOS is responsible for detecting and initializing the memory controller, it can enable and report only half of the RAM and hard disk to each OS. The other loading steps are the same as for a normal OS booting.

Our system uses ACPI S3 sleep for both secure switching and the normal OS sleep/wakeup. The BIOS uses the *OS_FLAG* to distinguish the two cases. When the trusted OS is suspended in S3 sleep, the BIOS can either wake up the trusted OS when *OS_FLAG* is set to 0 or wake up the untrusted OS when *OS_FLAG* is set to 1. However, a problem occurs when the BIOS tries to wake up the untrusted OS, which has not been loaded into the RAM at this time. To solve this problem, we use the *BOOT_FLAG* to indicate whether the untrusted OS has been loaded. When the system is powered on, the *BOOT_FLAG* is reset to 0 to reflect that the untrusted OS has not yet been loaded. When the BIOS detects that it is trying to wake up an untrusted OS that has not been loaded, it will load the untrusted OS and then set *BOOT_FLAG* to 1.

One major drawback of this method is that the granularity for memory allocation is the

size of DIMM. When one OS is running, only a portion of the RAM in the system can be used. We consider this the price of enhancing system security.

6.4.2 Switching Between Two OSes

OS switching is conducted by both the operating system and the BIOS. After both OSes have been loaded into the memory, the switching is done to put the currently-running OS into ACPI S3 sleep mode and then to wake up the other OS from ACPI S3 sleep mode. We use ACPI S3 sleep/wakeup because it has defined functionalities to save the CPU context and hardware devices' states. In ACPI S3 sleep mode, the CPU stops executing any instruction, and the CPU context is not maintained. The operating system will flush all dirty cache to RAM and save the CPU context. The Dynamic RAM context is maintained by placing the memory into a low-power self-refresh state. Only those devices that reference power resources are in the ON state. All the other devices (e.g., VGA, NIC) are in the D3 (OFF) state while their states are saved by the OS or the device drivers.

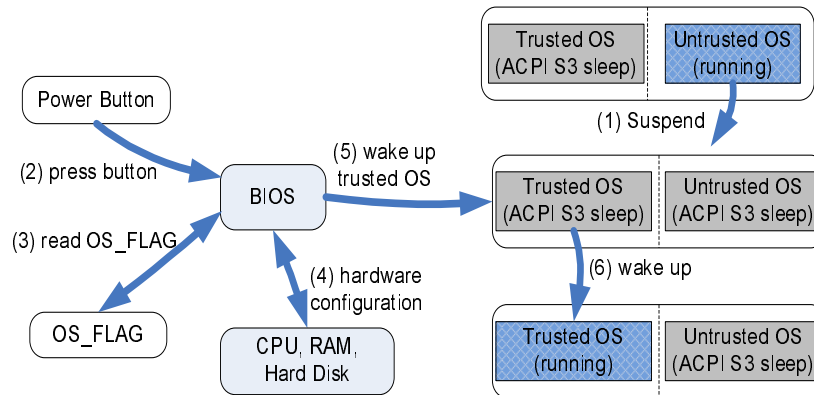


Figure 6.3: Switching Flow from Untrusted OS to Trusted OS.

Figure 6.3 shows the control flow when the system is switching from the untrusted to the trusted OS. The user first suspends the untrusted OS, which is responsible for saving the CPU context and hardware devices' states. Afterwards, both OSes stay in the ACPI sleep

mode. The user then presses the power button to wake up the system. This step is critical to enforce a trusted path by making the system enter the BIOS first. We will discuss this further at a later point. The BIOS can distinguish OS S3 wakeup from OS booting using some register in the southbridge. For instance, in the south bridge VT8237R [70], the three bits of “Sleep Type” in the Power Management Control register is set to 001 for S3 sleep.

Next, the BIOS reads the flag *OS_FLAG* to decide which OS should be woken. According to *OS_FLAG*, the BIOS programs the initial boot configuration of the CPU (e.g., the MSR and MTRR registers), initializes the cache controller, enables the memory controller, and jumps to the waking vector. After the BIOS forwards the system control to the OS, the trusted OS continues to perform the ACPI S3 wakeup and recover its CPU context and device states.

Integrity of System Flags

We must ensure the integrity of the system flags, *OS_FLAG* and *BOOT_FLAG*, to protect the system from fake OS attacks. However, the first challenge is to find a location to save these flags. We cannot simply save the two flags in the RAM because the BIOS uses the flags to enable the memory DIMM(s) and cannot read the flags before the RAM has been enabled. One solution is to save the flags in some unused bytes in CMOS. For example, we could save the *OS_FLAG* at offset 125 bytes of CMOS, and the *BOOT_FLAG* at offset 126 bytes of CMOS.

The second challenge is to ensure the integrity of the flags. Since the OS can also access CMOS, a compromised OS can change the flags in CMOS. For the *BOOT_FLAG* flag, the default value is 1, and it changes to 0 after booting the untrusted OS. If the untrusted OS changes the *BOOT_FLAG* flag to 1, the BIOS will load the untrusted OS again. Thus, the adversary can gain nothing aside from rebooting the untrusted OS, so we save the *BOOT_FLAG* flag in CMOS.

However, CMOS is not secure for the *OS_FLAG* flag. By manipulating the *OS_FLAG* flag from the OS, the adversary can launch the fake OS attack by waking the untrusted

OS instead of the trusted OS and then faking the trusted OS GUI under the control of the untrusted OS.

To protect the *OS_FLAG* flag and support the normal S3 sleep and wakeup, we can use a hardware bit to indicate the value (0 or 1) of *OS_FLAG* flag. This hardware bit can be manually and physically set only by the user, while the BIOS and OS can only read it. For example, we could use the standard parallel port to control the bit. In the D-Type 25-Pin Parallel Port Connector, the Pin Number 15 is used to signal an Error to the computer. The Status Port (base address +1) is a read-only port where Bit 3 reports the Error events. When the user connects Pin 15 (Error) and Pin 25 (the ground pin) with a wire or switch, the bit 3 of the Status port equals 0 and the BIOS will always wake up the trusted OS. When the user disconnects the two pins, the bit 3 of Status port equals to 1 and the BIOS will always wake up the untrusted OS. Many such hardware bits or devices can serve the same purpose in a computer.

Since the user controls which OS should be woken, our system can prevent the fake OS attacks while supporting the normal S3 sleep/wakeup. When users wants to do a normal sleep and wakeup, they do not change the parallel port connection; if they want to switch to another OS, they change the connection. The connection correctly shows the current OS that the user is running, and we can build a small switch with a long wire (so that the user can reach it easily) and put the 'TRUSTED' and 'UNTRUSTED' text on the two ends of the switch.

Trusted Path Enforcement

Since the BIOS is the only component that we can trust to enforce the trusted path from the untrusted to the trusted OS, we use the power button to ensure that the BIOS is entered during the OS wakeup stage, as shown in the second step in Figure 6.3.

Since the *OS_FLAG* alone is not enough to enforce the trusted path, the above-mentioned step is critical to prevent fake OS attacks. Supposing that the untrusted OS could fake the untrusted OS S3 sleep and trusted OS wakeup process on the monitor, the

BIOS and *OS_FLAG* could be totally bypassed. For such an attack, the attacker would also need to disable the power button events so that when the user presses the power button, the OS does not do what it is supposed to. To prevent such an attack, the user can use the system power LED to know the current system mode. The power LED lights up when the user turns on the power and blinks (or changes to another light, depending on the implementation) when the system is in sleep mode. Since the LED is hardware-controlled, the user can trust it to reveal if the untrusted OS has been suspended or not. If not, then this most likely is an attack.

6.4.3 Enforcing System Isolation

Our system depends on the BIOS and the ACPI S3 mode of the trusted OS to enforce resource isolation between the trusted and the untrusted OSes. Modern OSes (e.g., Linux and Windows XP) support ACPI S3 suspend/wakeup mechanisms, which can be used to enforce the isolation on CPU and I/O devices (e.g., VGA and NIC), without any modification of the OS. This dramatically lessens our need to save/recover the CPU context and devices' states. The BIOS must be customized to enforce isolation on RAM and hard disk, which cannot be thoroughly isolated by the OS alone. In the following, we first introduce the isolation capability of the ACPI S3 on CPU, NIC, and video devices. We then present the new mechanism using BIOS and OS to enforce the isolation of RAM and the hard drive.

Isolation based on Trusted OS

CPU Isolation: According to ACPI standards, the CPU context will be lost during the S3 sleep, and the untrusted OS cannot get any CPU context information of the trusted OS. The OS is responsible for saving and restoring the CPU context. In the untrusted OS, an attacker has only two options: either saving the CPU context or not saving it. If the attacker modifies the OS and does not save the CPU context, the untrusted OS cannot be resumed and this becomes a DoS attack. The trusted OS always follows the standard and saves the CPU context.

NIC Isolation: In S3 sleep, most of the devices are put into D3 (a no-power state for devices) state, during which the contexts for these devices are lost. Thus, there is no information leakage during the switching from the trusted OS to the untrusted OS. According to ACPI specifications, a network card may provide Wake-on-LAN functions to wake up the computer when the card stays in D0 or D3 power state. SecureSwitch only supports the network card in D3 state to wake up the computer, since the device in D0 state keeps its context that may be misused by the attacker. Fortunately, most of the current network cards support WOL at the D3 state [?].

Video Device Isolation: In S3 sleep, the content in the video buffer is lost. The ACPI specification does not require the BIOS to reprogram the video hardware or to save the video buffer, so BIOS does not know how to wake up the video card from an unprogrammed state. One easy way around this is to execute code from the video option BIOS in the same way as the system BIOS does. `vbetool` [71] is one such small application that executes code from the video option BIOS. It can run in the user space but may introduce some time delay in S3 sleep and wakeup.

Memory Isolation

Memory isolation is physically enforced by the BIOS. According to the `OS_FLAG` flag, the BIOS knows which OS is going to be booted or woken up, and it then initializes or wakes up the corresponding DIMM for that OS. The other DIMM remains uninitialized or unconfigured (though it may still maintain its data content). Prior to having the memory controller bring system memory out of the self-refresh mode, the BIOS is responsible for restoring the state of the processor's memory controller upon waking up from the S3 sleep state.

DDR2 and DDR3 memories need the BIOS to set the DQS settings for read and write. When the BIOS boots up the system, it searches the best settings for DQS. In addition to setting the memory controller in the north bridge, the BIOS also saves a copy of the setting in non-volatile RAM (NVRAM) of the south bridge. In normal S3 sleep mode, system

power is removed from the memory controller (the north bridge); however, a copy of DQS setting is maintained in NVRAM in the south bridge. During an S3 wakeup, the BIOS copies the DQS settings from the south bridge to the memory controller (the north bridge).

A normal system keeps only one set of the DQS configurations, while our secure-switching system must keep two sets of different DQS configurations to initialize/enable different DIMMs for two OSes. To wake up one OS, the BIOS should reset the DQS setting in the memory controller using the corresponding set of DQS settings.

Since we cannot save two sets of memory controller configurations in the same set of chipset registers, we must store them somewhere during the S3 sleep mode. For custom-designed computers with specific hardware devices, we may save the two settings in the BIOS. However, in many scenarios, the memory control should be dynamically adjusted to achieve optimal performance under different voltages and temperatures. We cannot keep them in the RAM, either, because the RAM is not accessible at that time. Our solution is to save the other set of settings in the CMOS. We save 64 bytes of Data Strobe Signal(DQS) settings, starting from the offset 56 of CMOS, which by default are not used according to the CMOS layout of the motherboard (ASUS M2V-MX SE).

In our system, the memory controller can only be reset by the BIOS, while the untrusted OS cannot initialize/enable the memory controller to access the DIMM for the trusted OS. The DQS settings contain more than one hardware register. For instance, there are 16 registers on AMD K8 and 4 registers on AMD family 10h processors. This means that there is a transient state wherein the system cannot access any DIMM before all the DQS settings are complete. When an attacker exploits a short program to modify the DQS settings, the program cannot obtain the next instruction from the main memory and the system will hang. BIOS can modify the DQS settings because it reads the instructions from the ROM that is not controlled by the DQS settings.

The untrusted OS can modify both DQS settings saved in the south bridge and in the CMOS. However, besides the DQS setting, the BIOS uses a “DIMM Mask” byte to control which DIMM should be enabled. In our system, the BIOS sets “DIMM Mask” according to

the *OS_FLAG* flag. We set the DIMM Mask to 0x01 only to enable the first DIMM, and to 0x10 only to enable the second DIMM. If the Mask conflicts with the DQS setting, the system will hang.

Note that another requirement for our system is that one memory controller should control more than one DIMM. In that case, even if there is more than one memory controller, the BIOS can initialize all of them to use part of the DIMMs connected with the controllers. Then the attacker still cannot modify the memory settings. Otherwise, if memory controllers are mapped one-to-one to the DIMMs, then some memory controllers have to be in an uninitialized (i.e., unconfigured) state that must be used by the trusted OS, and the attacker may try to initialize it by not affecting other memory controllers and DIMMs.

Hard Disk Isolation

The nonvolatile storage, such as hard disks used by the trusted OS, should be completely isolated from the untrusted OS in order to prevent information leakage. One solution is to encrypt a portion or the entirety of the hard disk before sleeping the trusted OS and to decrypt it after waking it up. However, the encryption/decryption operations will increase the switching time, along with the size of the hard disk.

Most motherboards (e.g., ASUS M2V-MX SE, in our implementation) has more than one SATA Channels to support more than one hard disk. When each OS can have its own hard disk, the BIOS can constrain access to the hard disk of the trusted OS. First, some hard disks support disk lock, an optional security feature defined by AT Attachment (ATA) specification [72]. This lock allows the user to set a password to lock a hard disk. Without knowing the password, no one can access the hard disk. The limitation of this method is that not all hard disks are provided with this feature. Second, according to the *OS_FLAG* flag, the payload of BIOS (e.g., SeaBIOS), which is responsible for hard disk initialization, can initialize only one of the two hard disks by setting the SATA Channel enable register (e.g., Bus0, Device15, Function0, offset0x40 on southbridge VT8237r). However, if the attacker knows the southbridge data sheet, it may reset the SATA Channel enable register

and initialize both hard disks.

With the observation that most applications in the trusted OS only require that a small amount of data (e.g., browser cookies) be saved on the hard disk, our system uses RAM disk to store the dynamic sensitive data in the RAM, which can be better protected by the system. In the beginning, we set up a special directory in the RAM disk and ask the user to save sensitive data into this directory. With Linux kernel version 2.6.18, we set the *ramdisk_size* parameter in *memu.list* to initialize about 256MB RAM disk. After booting into the trusted OS, we create a directory called */ramdisk* and mount RAM disk */dev/ram0* to the directory. However, the basic solution is not very user friendly, so we improve upon it by using a stackable file system (e.g., aufs [10,73]) to mount a read-write layer of RAM disk on top of regular directories, which are mounted as read-only. We create a home directory under the */ramdisk* directory and mount */home* to */ramdisk/home*.

The */home* directory is mounted as read-only, and all the files created under the */home* directory will be written into */ramdisk/home*, which is in RAM. Since the RAM is isolated between the trusted and untrusted OSes, the files in the RAM disk cannot be accessed by the attacker. Moreover, the files in RAM disk are lost after a reboot, so an attacker cannot access the sensitive data saved on the RAM disk after rebooting. For other binary program files, the system only needs to protect their integrity. For example, the user can run them from a CD or use other integrity-checking mechanisms to check those files.

6.5 Implementation & Experimental Results

We implement a prototype of the SecureSwitch system using an ASUS M2V-MX_SE motherboard with VIA K8M890 as the northbridge and VIA VT8237R as the southbridge. The CPU is AMD Sempron 64 LE-1300. Two Kingston HyperX 1GB DDR2 memory modules and two Seagate Barracuda 7200 RPM 500GB hard disks are installed. We connect a laptop with the prototype system through a serial port to debug and collect the experimental results. We also use a POST card to display the debugging codes from the BIOS.

We install CentOS 5.5 on one hard disk as the trusted OS, and Windows XP SP3 on another hard disk as the untrusted OS. Our implementation also supports two CentOS 5.5 (or Windows XP) OSes. We use the open-source Coreboot V4 [49] and SeaBIOS [74] as the BIOS.

6.5.1 Trusted Computing Base (TCB)

The BIOS provides the Trusted Computing Base (TCB) of our system. We measure the size of our prototype using `sloccount`¹, and the total lines of code(LOC) we added is just 120. This LOC is significant smaller than the 8471 LOC in Lockdown [25], and it is also smaller than other hypervisor- or microkernel-based methods that rely on an extra software layer in addition to the BIOS.

6.5.2 OS Loading and Switching Latency

We measure twice during SecureSwitch: *system loading time* and *switching latency*. System loading time is the time duration for loading two OSes into the memory. We use the real-time clock (RTC) to measure it. To record the beginning time, we print out the RTC time through the serial port console at the beginning of the BIOS code. For the ending time, we record the time when the “rc.local” file is executed in CentOS or when a startup application is called in Windows XP. The loading times for both OSes are very close within our system: 74 seconds for loading CentOS and 79 seconds for loading Windows XP. The total loading time is 153 seconds. Though the loading time is relatively long, it only occurs once when the user boots up the system. Moreover, the loading time may be reduced by using solid-state drive.

OS Switching latency measures the time duration when switching from one OS to another. It consists of two parts: the time to suspend the current OS and the time to wake up another OS. We use the system’s Time Stamp Counter (TSC) to measure the OS wakeup time. TSC is a 64-bit register that is present on all x86 processors since the Pentium, and

¹<http://www.dwheeler.com/sloccount/>

it counts the number of ticks since reset. After pressing the power button, the TSC is reset to 0. We write a user-level program to obtain the current TSC value continuously. We then calculate the wakeup time as $TSC \cdot (1/\text{CPU frequency})$. TSC can be used to measure the wakeup time for both CentOS and Windows XP. However, it is difficult to use TSC to measure Windows XP's suspension time without the Windows source code. Since the OS suspend does not involve the BIOS, we cannot use the BIOS to read the TSC value either. Instead, we use an Oscilloscope, Tektronix TDS 220, to measure the suspension time. We connect the oscilloscope to the serial port on the motherboard. When we initiate the ACPI S3 sleep, a customized program sends an electrical signal to the serial port to indicate the start of S3 sleep. When the system finishes S3 sleep, the oscilloscope receives a power-off electrical signal from the serial port. We use this method to measure the suspension delay for both CentOS and Windows XP.

Table 6.1: Switching Time

Switching Operation	Secure Switch(s)
Windows XP Suspend	4.41
CentOS Wakeup	1.96
Total	6.37
CentOS Suspend	2.24
Windows XP Wakeup	2.79
Total	5.03

The latency when the system switches from the trusted OS to the untrusted one is different from the latency when the system switches back, as shown in Table 6.1. We can see that switching from Windows XP to CentOS requires 5.03 seconds, which is a little faster than switching from CentOS to the Windows XP. For both OSes, the suspend time is longer than the wakeup time. Windows XP's suspend and wakeup times are longer than those of CentOS.

Table 6.1 only provides a rough latency measurement that is constrained to the specific

hardware and software used in our prototype system. For instance, these measurements will change when we use an external VGA card or execute a large number of processes in the OS. The ASUS motherboard has one integrated VGA card with VIA chip and 256 MB video memory. When we insert an external VGA card with S3 chip and 64 MB memory, the external VGA card needs less suspension time than the integrated one since it has a smaller video memory size. To our surprise, the external VGA card requires a wakeup time that is three times longer than the integrated one due to the fact that coreboot needs to call the option ROM of the external video card, but it encounters a computability problem and dramatically delays the wakeup.

In addition, we run multiple `while(1)` programs on the Linux to see how the CPU intensive processes affect the switching time. When we run five `while(1)` programs at the same time, the switching time is about three times longer. We deduced that most of the increasing is due to the user space suspend and wakeup, while the delay in kernel space does not change much. This leads us to breakdown the operations in BIOS, user space, and kernel space to understand the major contributors for the time delay. Due to the closed-source nature of Windows XP, we only break down the operations on the CentOS 5.5 with Coreboot V4.

Linux Suspend Breakdown

We use Ftrace [75] to trace the suspension function calls in Linux S3 sleep. According to the function call graph generated by Ftrace, we divide the suspend operations into two phases: *user space suspend* and *kernel space suspend*. We use the `pm-suspend` script provided by the OS to trigger the suspend. The script basically notifies the Network Manager to shut down networking and uses `vbetool` [71] to call functions at video option ROM to save VGA states. It then echoes string “mem” to `/sys/power/state`. This jumps to the kernel space and stops the user space. In the kernel space, the suspend code goes through the device tree and calls the device suspend function in each driver. The kernel then powers off these devices. To measure the user space suspend, we record the TSC time stamp in

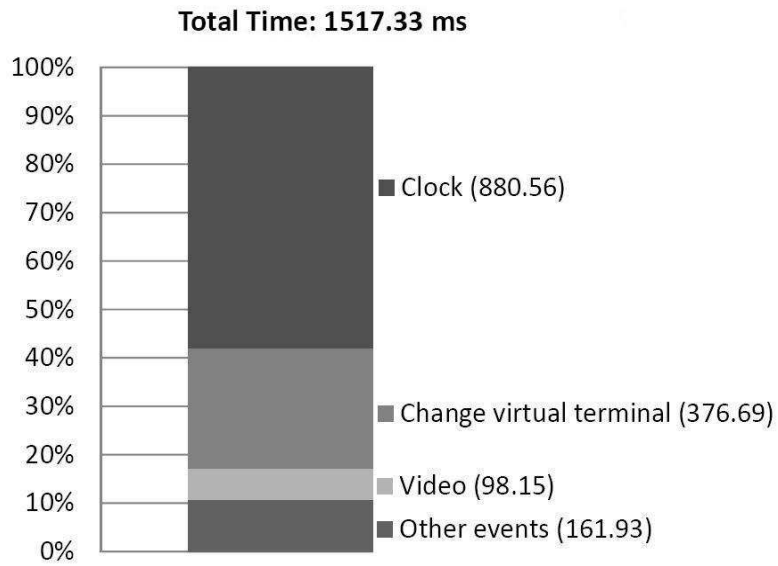


Figure 6.4: User Space Suspend Breakdown

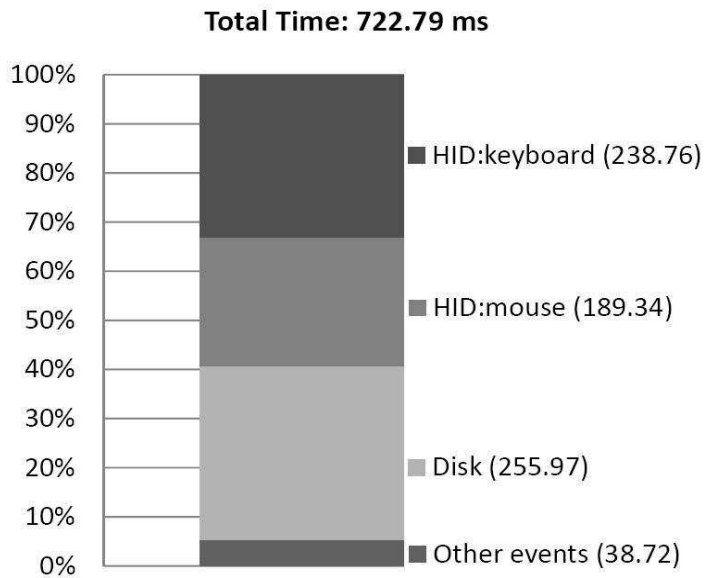


Figure 6.5: Kernel Space Suspend Breakdown

file `/var/log/pm/suspend.log`. For kernel time measurement, we add `printk` statements between various components of the kernel.

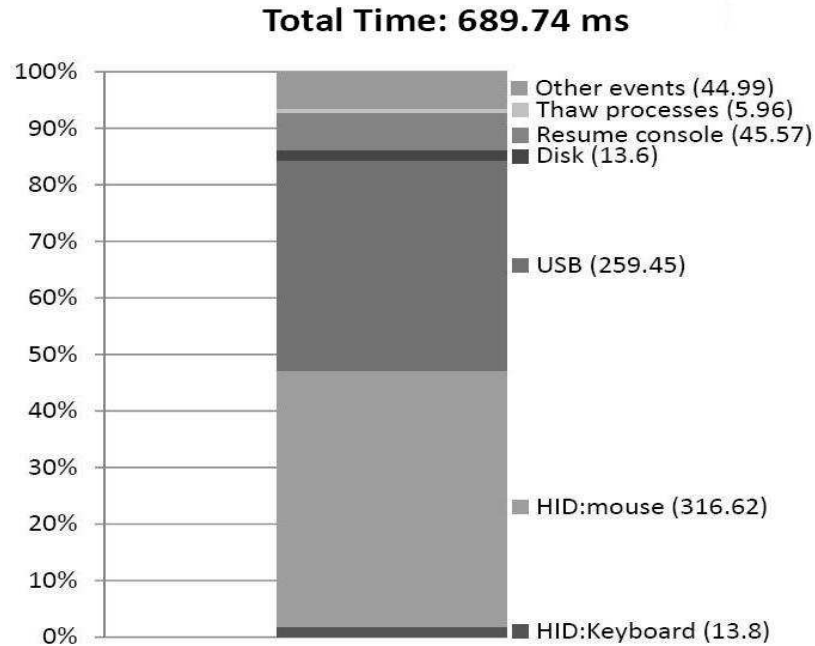


Figure 6.6: Kernel Space Wakeup Breakdown

Figure 6.4 shows the time breakdown for user space suspend, and Figure 6.5 shows the time breakdown for kernel space suspend. The total suspend times for user space and kernel space are 1517.33 ms and 722.79 ms respectively. In the user space, by running command `chvt 63`, the monitor changes the GUI terminal to `/dev/tty63` as the foreground virtual terminal. In the clock operation, the OS stops the Network Time Protocol Daemon and writes the current system time to RTC time in CMOS. For the video operation, the OS uses `vbetool` [71] to save current video state to the `/var/run` directory in memory. Other events include stopping network manager and saving the state of CPU frequency governors, etc. In the kernel space, the most time is consumed by stopping the keyboard, mouse, and hard disks. We use a PS/2 mouse and keyboard in our system. The mouse and keyboard suspend functions in the driver and reset the devices, which causes the delay. For the hard disk,

delay comes from synchronizing the cache. The two hard disks each have 16 MB caches, and cache write is enabled by default for the SATA disk [72]. The OS also needs to stop other devices, such as the USB and serial ports, which takes relatively less time.

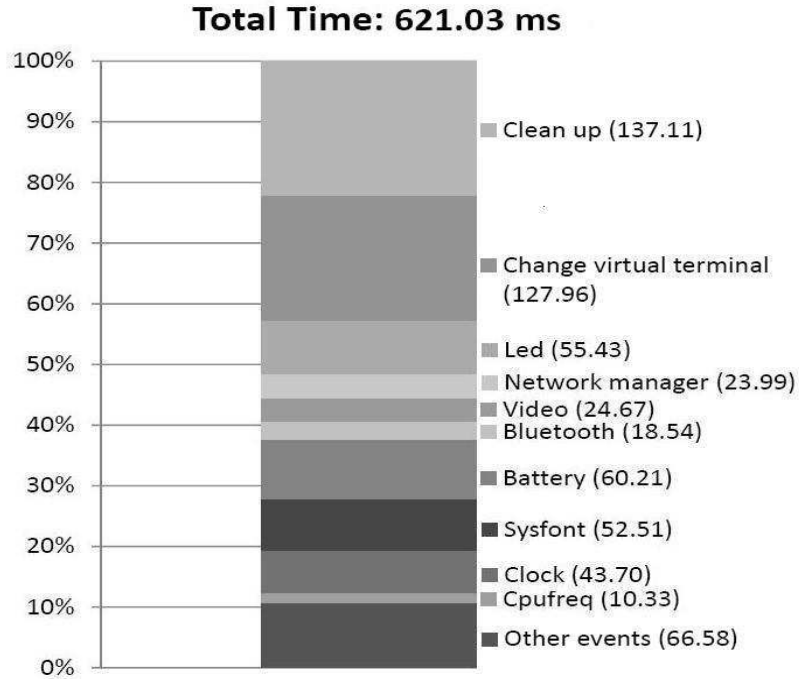


Figure 6.7: User Space Wakeup Breakdown

Linux Wakeup Breakdown

Unlike S3 suspend, S3 wakeup operations are handled by both the BIOS and the OS. The wakeup process starts from a hardware reset. The system enters the BIOS first, then jumps to the OS wakeup vector. The total latency time in BIOS is almost constant and equal to 1259.25 *ms*. Again, the OS wakeup operations can be divided into two parts: kernel space wakeup and user space wakeup. The wakeup latency in kernel space and user space are 698.74 *ms* and 612.04 *ms* respectively. Figure 6.6 shows the time breakdown for the major components in the kernel space. The major delay contributors in kernel space are the USB and the mouse. There are four USB ports on the motherboard. Since coreboot doesn't provide an optimized support for the USB, OS needs to initialize all four of the

Table 6.2: Comparing SecureSwitch with Other systems

	SecureSwitch	Lockdown [25]	TrustVisor [12]	Flicker [13]
Trusted Computing Base	BIOS	BIOS+Hypervisor	BIOS+Hypervisor	BIOS
Switching Time (second)	≈ 6	40	< 1	1
Software Compatibility	High	High	Low	Low
Hardware Dependency	ACPI	ACPI + TPM	TPM (DRTM)	TPM (DRTM)
Memory Overhead	High	Low	Low	Low
Computation Overhead	Low	Median	Low	High

USB ports. The BIOS must initialize the keyboard, but not necessarily the mouse. We discovered that the mouse takes more time than the keyboard in kernel-space wakeup due to the OS initialization of the mouse. Figure 6.7 shows the time breakdown for wakeup in the user space. We can see that cleaning up the files and changing the foreground’s virtual terminal (*chvt 1*) take up most of the time.

6.5.3 Comparison with Other Methods

There is recent research on protecting the execution of security-sensitive code on legacy systems [12, 13, 25]. We contrasted these with SecureSwitch using the following metrics: trusted computing base, switching time, software compatibility, hardware dependency, and performance impact on the system. Table 6.2 presents the comparison results.

The BIOS code is the trusted computing base (TCB) for both SecureSwitch and Flicker [13], while Lockdown [25] and TrustVisor [12] must also ensure the security and integrity of a hypervisor when loading it from the hard disk. Both Flicker and TrustVisor have a very small switching time since they can ensure a hardware-assisted, trusted execution environment without OS sleep/wakeup. The switching delay in SecureSwitch is small and acceptable, while Lockdown requires a relatively long switching time.

In Flicker and TrustVisor, the security code must be custom-compiled or ported to run in the secure environment, while the legacy programs can run directly on both SecureSwitch and Lockdown without any changes. SecureSwitch requires the hardware devices to support

ACPI, which has already been widely supported by hardware manufactures for efficient power management. All of the other solutions depend on TPM to protect the integrity of the hypervisor or to provide the Dynamic Root of Trust Measurement (DRTM) [76] feature. The memory overhead in SecureSwitch is high due to the coarse physical isolation on the DIMMs. The memory overheads in other methods are fairly low.

In SecureSwitch, Lockdown, and Flicker, when a security code is running in the trusted environment, the applications in the untrusted environment are fully stopped. Lockdown requires 15-55% more computation overhead in the trusted environment due to the NPT pages. Flicker incurs significant computation overhead due to its frequent use of hardware support for DRTM. SecureSwitch adds no computation overhead in the trusted environment. TrustVisor can execute the applications in the untrusted environments with little overhead when the security code is running in the trusted environment; however, this requires code modification. Although possible, it would seem to be an engineering challenge to port all existing code to support this, especially for an entire commercial OS.

Chapter 7: Hardware-assisted Forensic

7.1 Introduction

Analyzing volatile state information from a live, working system without disrupting its operation is critical for forensics because it offers a context for the static, non-volatile data. Indeed, by extracting the dynamic context, an analyst is empowered with a wide range of information from process description, including currently-running and stopped programs and their physical memory, CPU and Disk cache, network connections and open files, to operating system state, such as system load, open sockets, and inter-process communication. This global view of the system state is crucial, especially for stealthy malware that maintains a fully-dynamic profile by operating entirely in memory. Code Red [77] and SQL Slammer [78] are two instances of such malware. Therefore, being able to unobtrusively acquire the complete contents of the volatile memory of a computer is a necessary source of evidence for digital forensic analysis [27, 79, 80]. Currently, the majority of forensic analysis tools employ the operating system, employ dedicated hardware to retrieve the running state of a system, or require the analysis of memory traces after shutdown [81].

Nevertheless, it is not trivial to reliably acquire the memory content of a running system without disrupting its operation. Many operating systems (OSes) provide software interfaces for the user or kernel-level programs to read specific areas of memory. However, if the OS itself has already been compromised or if the malware operates on the hypervisor level, then the OS cannot be trusted to provide the correct RAM contents. Rootkits are notorious for hiding process information from user-land and sometimes kernel-level programs that read standard memory structures. Due to these shortcomings, software solutions for memory acquisition are not reliable. To address this limitation, researchers [20, 27] have turned their attention to hardware-based methods. For example, a customized PCI card

can be used to read the physical memory via Direct Memory Access (DMA).

Unfortunately, using specialized PCI cards to access the volatile state still has two unresolved challenges. First, PCI cards can be blocked from access to all physical RAM memory. Both Intel and AMD provide technologies (VT-d [29] and IOMMU [82]) to restrict memory access for peripheral devices as a security precaution. Although the purpose is to protect the privileged software from being compromised by malicious hardware, these technologies also thwart the PCI-dependent tools from monitoring the physical memory. Second, it is difficult to obtain the semantics of a memory dump without knowing the values of the CPU registers at the time that the dump was retrieved. For instance, the interrupt descriptor table register (IDTR) points to the current interrupt descriptor table (IDT). Without knowing the value of the IDTR register, an analyst has to apply heuristic or pattern-matching methods to search for and identify the value of the IDTR. Of course, none of these methods are considered reliable and thus cannot be used to substantiate a robust forensic analysis. The same holds true for a range of other CPU-derived information, including the control register 3 (CR3), which points to the base address of the current page table.

To address some of these challenges, we introduce a firmware-assisted method to reliably acquire the memory and CPU registers. In addition, we propose a memory analysis framework to assist the forensic investigator with the tedious task of analyzing the machine state. To this end, we design the capability of both interactively and automatically examining the contents of memory, CPU registers, and peripheral state remotely. We combine a commercial PCI network card (which is widely available) with the System Management Mode (SMM) to acquire the memory and CPU registers. SMM is a special CPU mode separate from the protected mode and real-address mode [1]. The code of SMM, which resides in the BIOS, can check the CPU registers and leverage the commercial PCI network cards to read the memory. Therefore, we can read and transmit CPU registers, physical memory, and peripheral device states.

The memory and CPU information can either be communicated to a remote server to

be examined off-line or analyzed online (i.e., live forensics). For the live-analysis mode, we can guarantee consistency because the OS enters and remains in suspended state during SMM. While in suspended state, the system view remains untouched while being examined. To investigate the live-memory contents, we propose implementing a GDB-like server and a GDB stub in SMM so that it can be connected with a GDB debugger on another machine via serial console. The benefit of this SMM-based online investigation is that it can examine the physical memory of user-level programs and kernel code. Since the debugger running in SMM is isolated from the OS, it is much more reliable than other OS-based live-forensic methods. Even if the whole OS, including the kernel, is compromised, the SMM is still protected and can reliably perform the investigation tasks.

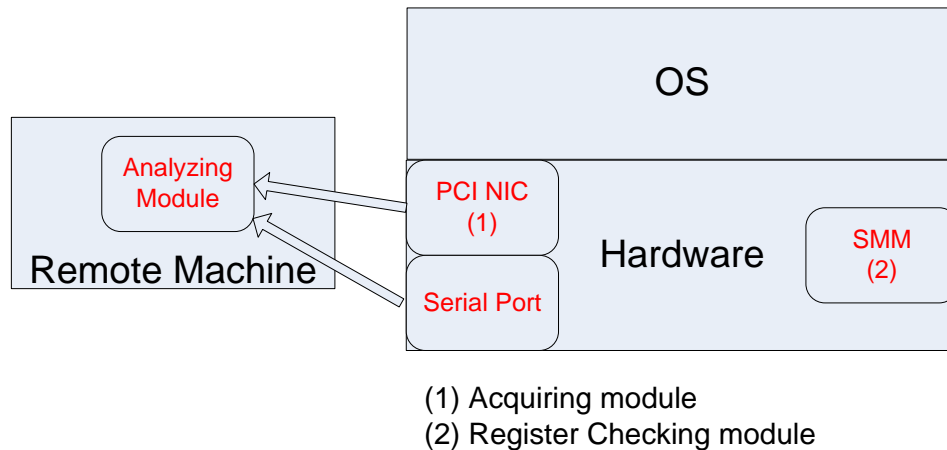


Figure 7.1: The architecture of the memory forensic tool.

7.2 Architecture

The overall architecture of our system is depicted in Figure 7.1. The entire system is comprised of two computers: one used for analysis and the other subjected to inspection. On the right side of the figure is the target machine that is being investigated, which includes a dedicated network card and a serial port. On the left side of the figure is the remote

machine that is used by the forensics investigator. These two machines are connected by a network cable and a serial cable. The network connection is used for off-line investigation, the serial connection for on-line investigation.

7.2.1 Off-line Investigation

The off-line memory investigation collects the content of the physical memory and transfers it to the remote server. The remote investigator can then recover the semantic information (e.g., both user-level processes and kernel modules) from the acquired content. When the remote server receives the acquired content, the OS on the target machine will have resumed its operation. Thus, the current OS context is different from that when the memory content is acquired.

Acquiring Memory

A commercial PCI network card is used to acquire the physical memory of the target machine. PCI network cards can read the physical memory through DMA and then send it out as network packets. However, one problem with using the commercial network cards is that they require a driver. Since the drivers normally reside in the OS, if the OS has been compromised by malware, the driver may be compromised as well and cannot be trusted. Some researchers have overcome this issue by using a customized PCI card, which does not need a driver, to read the physical memory [27]. The drawback to this is that the cost for customized hardware may be high and hard to deploy. We propose using commercial network cards without worrying about their drivers. To solve this issue, we put the drivers of the network cards into the SMM. Since SMM code is trusted and locked, the drivers are not threatened by the malware.

Translating Memory

The PCI network card can only acquire the physical memory of the target machine. To understand the physical memory, the system must translate it to virtual memory used by

the OS and find out the semantics of the memory. For this purpose, two CPU registers are critical: IDTR and CR3. IDTR points to the current IDT, and CR3 points to the base address of the current page table.

CPU registers can be obtained either through a software- or a firmware-based method. We choose the firmware-based (BIOS) method because it is more reliable. We use SMM code, which is a part of the BIOS, to read the CPU registers. As we mentioned in Chapter 3, SMM can obtain the CPU registers used by the OS running in the protected mode because the hardware automatically saves them before switching to SMM.

Challenges

One challenge for off-line investigation is how to perform it reliably on machines with large amounts of memory. For example, current workstations or servers may have 4GB, 8GB, or even higher amounts of physical memory. If we use a 1Gbps network card, then it takes at least 32 seconds to acquire the 4GB memory and 64 seconds for 8GB memory. A serial port is not suitable for transmitting large off-line data due to its low data rate.

There are two ways to acquire the memory: (1) record the whole memory in one operation or (2) record only a portion of the memory, resume the OS, and then repeat the process until all of the memory is recorded. The first method guarantees that the memory content will be consistent because it freezes the OS during the process. However, the OS may hang if it is frozen for a relatively long time, partly because too many hardware interrupts are lost when frozen. The second method does not interfere too much with the OS, but the memory acquired during a relatively long period may not be consistent. To improve upon the first method, we could use advanced network cards, such as 10Gbps cards. To improve upon the second method, we could only capture the difference between the two snapshots of the memory. This would dramatically reduce the amount of data to be recorded. To record the difference, we could mark all of the memory pages as read-only and then every write to the memory will trigger an exception. In the exception handler, the OS records the memory pages to be modified and sends this information to the SMM. This method introduces

some performance impacts and modifications to the OS. In addition, the exception handler should be protected. SMM can be used to watch the integrity of the exception handler.

7.2.2 On-line Investigation

Although the off-line investigation can obtain the complete memory view of the target machine for a given time, some memory content may not be necessary for the investigation. For example, if the investigator only wants to check the OS kernel, then the memory occupied by the user-level programs does not need to be recorded or transmitted. Moreover, if the investigator only wants to check a few variables, then most of the memory dump will be unnecessary. In this situation, it is more convenient and efficient to use an on-line investigation.

In an on-line investigation, the investigator stops the machine, checks a few memory locations or specific registers, and then resumes the machine. We use SMM to stop the machine and obtain the memory contents. Instead of sending all of the memory to the remote machine, SMM simply waits for the command from the investigator at the remote server. Only a small number (i.e., a few bytes) of memory content will be sent out when it is requested by the investigator through online analysis interface. Since this process is similar to a debugging process, we plan to implement a GDB stub and gdbserver in the SMM of the target machine and connect it with a GDB debugger on the remote machine. Gdbserver supports both a TCP connection and a serial line connection. For our system, we plan to use a serial connection due to simplicity.

Triggering SMM

The first step for an online investigation is to trigger SMM to freeze the machine. This can be done in several ways. For example, a program or a kernel module can write to port `0xB2` on an Intel-based motherboard to trigger SMM. The PCI network card can also be used to trigger SMM. From the PCI3.0 specification [83], a PCI card can either trigger a normal interrupt or a System Management Interrupt (SMI) that will enter SMM. Another way to

trigger SMM is to use a hardware timer to periodically trigger SMM. In this case, the SMM code can ask the remote machine whether or not stopping the machine is requested. If not, then the code will exit SMM; otherwise, the code will stop the machine and wait for commands from the remote server.

For server machines, IPMI can also trigger SMI. IPMI stands for the Intelligent Platform Management Interface [84], which is a standardized computer system interface used by system administrators to manage a computer system and monitor its operation. Intel leads the development of IPMI, and more than two hundred computer system vendors support it. IPMI relies on a baseboard management controller (BMC) to manage the interface between system management software and platform hardware. BMC is a specialized microcontroller embedded on the motherboard of a computer, and it is capable of triggering SMI. Therefore, both IPMI and BMC can be used to remotely trigger SMI[19].

Gdbserver and GDB stub

Gdbserver is a control program for Unix-like systems, which allows users to connect their program with a remote GDB without linking in the usual debugging stub [85]. The debugging stub, sometimes referred to as a GDB stub, is also required to support remote debugging. GDB and gdbserver can communicate with each other using the standard GDB remote serial protocol.

We plan to implement gdbserver and GDB stub in the SMM. When a debugging exception occurs, the exception handler will invoke SMM first and then the gdbserver in SMM will communicate with the GDB running on the remote machine. One challenge is how to integrate SMM with gdbserver. Normally, SMM code is one part of the BIOS and is written in assembly language. For some old machines, the SMM is not locked by the BIOS so that third-party developers can write their own code and load it into the SMM. However, for new machines, the SMM is typically locked by the BIOS. To solve this issue, we use coreboot.

Coreboot [49] (formerly known as LinuxBIOS) is an open-source project aimed at replacing the proprietary BIOS (firmware) in the majority of today's computers. It performs

a bit of hardware initialization and then executes a so-called payload, such as SeaBIOS [74]. Coreboot is a modern version of the firmware because it switches to protected mode in a very early stage and is written mostly in C language. Coreboot also initializes the serial port to output some debug messages. For our purposes, we could modify the coreboot source code, find the SMI handler, and implement the GDB remote serial protocol (RSP) there.

The GDB RSP provides a high-level protocol specification, allowing GDB to connect remotely to any target. If a target implements the server side of the RSP protocol, then the debugger will be able to connect remotely to that target. The protocol supports a wide range of connection types: direct serial devices, UDP/IP, TCP/IP, and POSIX pipes.

The RSP is a simple, packet-based scheme. The format for each packet is as follows:

```
$ <data> # CSUM1 CSUM2
```

The packet starts with the \$ symbol, followed by the actual data, then the # symbol, and finally, two hex bits for checksum. GDB provides many commands for debugging. In our prototype, we plan to first implement the minimum commands that are critical for the live forensic applications. Some commands include the following:

1. 'g'. 'g' is used to read the content of all CPU registers. When the GDB server receives this command, it returns the register contents to the client.
2. 'G'. 'G' is used to write the content of the CPU registers. Values to be written are in the same packet with the 'G' command. It requires a reply such as 'ACK' or 'NAK'.
3. 'm'. 'm' command reads the content of a specific memory. When received, the GDB server returns the content to the client.
4. 'M'. 'M' command writes the content to a memory address. Values to be written are in the same packet with the 'M' command. It requires a reply such as 'ACK' or 'NAK'.

5. 'c'. 'c' command resumes the execution. It can be transmitted with or without an address. If an address is specified, then resume the execution from that address. Otherwise, resume from the current address.

One challenge is that the debugger cannot freeze the target OS in the SMM mode for too long; otherwise, the target machine may hang. One solution is to automatically quit the SMM mode after a predetermined time period. Another solution is to determine the reason that the OS hangs and try to modify the OS (e.g., increase the buffer) to support the extendedly-frozen OS situation.

Chapter 8: Security Analysis and Open Problems

8.1 Security Analysis of HyperCheck

HyperCheck aims to detect the modifications to the control data and the codes of the hypervisors or OS kernels. These kinds of attacks are realistic and have a significant impact on the system. HyperCheck can detect these attacks by using an Ethernet card to read the physical memory via DMA and then analyze it. For example, if the attackers control the hypervisor and make some modifications, HyperCheck can detect that change by reading the physical memory directly and comparing it with the previous pristine value.

In addition, HyperCheck also uses SMM to monitor CPU registers, which provides further protection. Some previous research relies only on the symbol table in the symbol file to find the physical address of the kernel code and data. However, there is no binding between the addresses in the symbol table and the actual physical addresses of these symbols [2]. For example, one potential attack is to modify the IDTR register of CPU to point to another address. The malware can then modify the new IDT table, keeping the old one untouched. Another potential attack method is to keep the IDTR register untouched but modify the page tables of the kernel so that the virtual address in the IDTR will actually point to a different physical address. HyperCheck can detect these cases by checking CPU registers in SMM. In SMM, HyperCheck reads the content of the IDTR and CR3 registers used by the operating system. IDTR should never change after booting. If it changes, SMM will send a warning through the Ethernet card to the monitor machine. From CR3, HyperCheck can find the actual physical addresses given the virtual ones. The offset between the virtual and physical addresses should be static. If some offsets have changed, HyperCheck will generate a warning, too. Moreover, PCI devices, including the Ethernet card alone, can be cheated to obtain a different view of the physical memory [68].

With SMM, we could avoid this problem by checking the corresponding settings in SMM.

The network card driver of HyperCheck is put into the SMM code to avoid malicious modifications. In addition, to prevent replay attacks, we use a key to hash a portion of the data randomly and then send them out to the analysis module. Since the key is private and locked in the SMRAM, the attacker cannot get it and generate the same hash. The attacker can still try to disable the Ethernet card or the SMM code, but we can detect this through an out-of-band monitor, such as the Dell remote access controller.

In addition, the attacker may try to launch a fake reboot attack to obtain a private key from the monitor machine. It can mimic the SMM NIC driver and send a request for a new key. For such an event, we have two options. First, we could use Trusted Platform Module (TPM)-based remote attestation to verify the running state of the target machine [13]. We would only need to verify whether the OS has been started or not. If it is already started, then the monitor machine should refuse to send the key. If the target machine does not have a TPM, the second method is to send another reliable reboot signal to the target machine when it asks for the key to make sure the SMM code is running.

8.2 Security Analysis of SecureSwitch

Our system can enforce a trusted path during secure switching and ensure a firmware-assisted resource isolation between two OSes. The untrusted OS cannot steal data from the trusted OS or compromise the integrity of the data in the trusted OS. Our system can prevent Data Exfiltration attacks and the fake OS attacks. We do not prevent DoS attacks because the user can easily notice this type of attack and reboot the machine to recover.

Data Exfiltration Attacks. The untrusted OS cannot steal any data information from the trusted OS using either shared or separated devices. The two OSes have a separated RAM and hard disk. Since the untrusted OS cannot change memory DQS settings without crashing the system, an attacker cannot access the memory DIMM of the trusted OS. To protect the small amount of data saved in the hard disk, we use the RAM disk to save the sensitive data in the memory.

The two OSes share all of the hardware devices aside from the RAM and hard disk. The ACPI S3 sleep guarantees that the trusted OS won't leave any sensitive data on those devices to be accessed by the untrusted OS. First, the CPU context, including registers and caches, will be flushed during S3 sleep. In AMD K8, the north bridge is integrated in the CPU, and its content is flushed as well. The NVRAM in south bridge only records some system configuration data. Second, for the hardware devices with their own buffers, such as the VGA and NIC, all of the content in their buffers will be lost because those devices lose power in S3 sleep.

Suppose that the user has started a sensitive Web transaction in the trusted OS and switches to the untrusted OS before the end of the transaction. If the remote server keeps sending sensitive data without any encryption, the attacker may receive this data. This problem can be solved by (1) protecting all web transactions with encryption, or (2) closing all Web transactions by the OS before the switching occurs.

Fake OS attacks. Our system can prevent fake OS attacks by enforcing a trusted path during OS switching and protecting the system flag *OS_FLAG*. Another promising solution is to employ TPM for remote attestation, wherein a trusted remote server can verify the identify of the running OS.

Network Attacks on Trusted OS. We assume that the trusted OS is secure and can be trusted. However, because an OS contains tens of thousands of lines of code, there exist vulnerabilities that can be misused by attackers from the network. Our system can guarantee that the trusted OS won't be compromised by the untrusted OS. However, if normal users use the trusted OS for a long time, we cannot guarantee that the trusted OS won't be compromised from network attacks. The stateless OS mode can only alleviate this attack by restoring the trusted OS to a pristine state every time it is woken, but it cannot prevent the attack. One promising solution is to employ some TPM- or SMM-based integrity checking mechanisms [41, 76] to detect any OS tampering attempts by comparing the newly-generated OS states with a clean state. However, that is beyond the scope of this thesis.

8.3 Open Problems

8.3.1 Checking Dynamic Data

Currently, HyperCheck can only check the integrity of the static data. One remaining issue is how to check the dynamic data with HyperCheck. In order to do this, the HyperCheck needs to understand the semantics of the physical memory. Some previous research works are attempting to understand the semantics and to enforce the kernel data structure invariants, such as Gibraltar [26] and KOP [86]. Gibraltar uses a CIL [87] module (extractor) to extract the kernel invariant during a training phase and then enforce it during an enforcement phase. They use a PCI network card to acquire the memory snapshot asynchronously, so there is an inconsistency issue. To remedy this, they place an upper bound on the number of objects traversed by the extractor. HyperCheck could use a similar method to enforce the kernel data structure invariants. Since HyperCheck can also obtain the CPU registers, it may produce more accurate results than Gibraltar.

Another problem is how to enforce the data structure invariants, not only for kernels but also for user-level applications. The pages used by an application may be swapped out to the hard disk, and this requires the system to be able to acquire the swapped out pages as well. In addition, if we want to check both the kernel and the applications, the size of the memory snapshot may become very large (e.g., 4GB or larger), and the inconsistency issue may become more serious. Currently, there are no known solutions for this issue. Some systems try to address this issue by using a hypervisor [88], which is not preferred in this thesis.

8.3.2 Memory Isolation Granularity

For the current SecureSwitch system, the granularity of memory isolation is based on the size of one DIMM. If a computer has only one DIMM, then it cannot run SecureSwitch. If a computer has two equal-size DIMMs, then the memory assigned to each OS (trusted or untrusted) must be the same. This granularity is too coarse and may not be desirable.

Another method for memory isolation in SecureSwitch is to modify one OS so that it uses only the second half or an arbitrary amount of the RAM. Suppose that the total physical RAM memory is 2GB. One OS can be configured to use only memory from 0 to 1GB (or X GB), and the other OS would then only use memory from 1GB (or X GB) to 2GB. We tested this method but failed. The challenge is to configure one OS to use only from 1GB to 2GB because an OS always assumes the lowest 1MB physical memory is available for legacy issues.

One solution to address this issue is to employ the memory management component of a hypervisor, such as Xen. However, this requires that another trusted software be running under both OSes. If this layer of software is compromised, then the isolation will fail. Furthermore, suppose that we solve the problem of reserving the first half of the memory for another OS without using a hypervisor-like software. The physical memories of the two OSes would still not be isolated from each other. The OS kernel, if compromised, could access any physical memory, even if the BIOS did not report it. Note that this is different from our DIMMs-based isolation, where the DIMM(s) is physically uninitialized (or unconfigured). Therefore, if we separate the two OSes by modifying the OS itself, then the memory of the trusted OS must be encrypted before switching to the untrusted OS and decrypted before switching back.

Chapter 9: Conclusions and Future Work

9.1 Conclusions

In this thesis, we introduced two hardware-assisted protection systems: HyperCheck and SecureSwitch. HyperCheck is a hardware-assisted tamper detection framework, while SecureSwitch provides a trusted environment for sensitive tasks and allows complete isolation and quick switching between trusted and untrusted environments.

HyperCheck is designed to protect the code integrity of software running on commodity hardware. This includes VMMs and Operating Systems. To achieve this, we rely on the CPU System Managed Mode (SMM) to securely generate and transmit the full state of the protected machine to an external server. HyperCheck does not rely on any software running on the target machine beyond the BIOS. Moreover, HyperCheck is robust against attacks that aim to disable or block its operation.

To demonstrate the feasibility of our approach, we implemented two prototypes: one using QEMU and another using a network card on a commodity x86 machine. Our experimental results indicate that we can successfully identify alterations of the control data and the code on many existing systems. More specifically, we tested our approach in part of the Xen hypervisor, the Domain 0 in Xen, and the control structures of other operating systems, such as Linux and Windows. HyperCheck operation is relatively lightweight; it can produce and communicate a scan of the state of the protected software in less than 40ms. We also analyzed one kind of possible attack to the HyperCheck and other SMM-based protection systems and provided the defense mechanisms for this.

SecureSwitch is designed to provide a context-dependent, trustworthy environment for sensitive tasks without sacrificing the usability provided by the unsensitive applications that have increasing number, size, and complexity, and are running on desktop computers.

Having such environments will enable the user to segregate different activities and lower the attack surface while maintaining system usability. A design tenet of SecureSwitch was the ability to quickly and securely switch between operating environments without extensive code modifications or a need for specialized hardware. At the same time, we wanted to minimize the code attack surface and prevent mutable, non-BIOS code from controlling the switching process. Finally, the system had to offer protection against attacks that aim to deceive the user's perception of the operating environment that he/she is currently in. We believe that the proposed framework achieves all of these goals. In our prototype implementation, the switching process takes approximately six seconds. Moreover, the user can clearly discern the state of the system and seamlessly switch between untrusted and trusted OSes to perform sensitive transactions.

9.2 Future Work

We have designed the framework for hardware-assisted forensic, but as we mentioned in Chapter 7, there are some challenges. One key issue is to discover whether the operating system will hang because of a long-duration SMM execution. If so, what is the reason and how can this be solved? We suspect that the hardware interrupt may cause some problems. If too many interrupts are lost, then the hardware or the driver may fail to work. We will do more experiments to verify whether this is the case or not.

Another future initiative will be to improve the hardware-assisted forensic framework for hardware-assisted debugging. Existing user-level debuggers and kernel debuggers have some problems if the operating system is unreliable or compromised. Hardware is more reliable, as we discussed in this thesis. Therefore, we plan to use a similar technique for debugging purposes. To that end, we plan to support more GDB commands, such as single stepping and setting break points. We could use the hardware debugging support provided by the processor and combine it with SMM and a NIC or a serial port. The benefit of hardware-assisted debugging is that it can support any software running above it, such as an operating system kernel or a hypervisor.

Bibliography

Bibliography

- [1] Intel Corp., “Intel® 64 and IA-32 Architectures Software Developer’s Manual,” June 2010.
- [2] L. Litty, H. A. Lagar-Cavilla, and D. Lie, “Hypervisor support for identifying covertly executing binaries,” in *SS’08: Proceedings of the 17th conference on Security symposium*. Berkeley, CA, USA: USENIX Association, 2008, pp. 243–258.
- [3] B. Payne, M. de Carbone, and W. Lee, “Secure and flexible monitoring of virtual machines,” in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, Dec. 2007, pp. 385–397.
- [4] T. Garfinkel and M. Rosenblum, “A virtual machine introspection based architecture for intrusion detection,” in *In Proc. Network and Distributed Systems Security Symposium*, 2003, pp. 191–206.
- [5] Z. Wang, X. Jiang, W. Cui, and P. Ning, “Countering kernel rootkits with lightweight hook protection,” in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 545–554.
- [6] 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*. ACM, 2007, p. 138.
- [7] 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 twenty-first ACM SIGOPS symposium on Operating systems principles*. ACM, 2007, p. 350.
- [8] R. Riley, X. Jiang, and D. Xu, “Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing,” in *Recent Advances in Intrusion Detection*. Springer, 2008, pp. 1–20.
- [9] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, “Terra: A virtual machine-based platform for trusted computing,” *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 193–206, 2003.
- [10] J. Wang, Y. Huang, and A. Ghosh, “SafeFox: A Safe Lightweight Virtual Browsing Environment,” in *System Sciences (HICSS), 2010 43rd Hawaii International Conference on*. IEEE, 2010, pp. 1–10.
- [11] R. Cox, J. Hansen, S. Gribble, and H. Levy, “A safety-oriented platform for web applications,” in *Security and Privacy, 2006 IEEE Symposium on*. IEEE, 2006, pp. 15–364.

- [12] 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 IEEE Symposium on Security and Privacy*, 2010.
- [13] 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*. ACM, 2008, pp. 315–328.
- [14] Z. Wang and X. Jiang, “Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity,” in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP ’10, 2010, pp. 380–395.
- [15] National Institute of Standards, NIST, “National vulnerability database, <http://nvd.nist.gov>.”
- [16] R. Wojtczuk, “Subverting the Xen hypervisor,” 2008. [Online]. Available: <http://invisiblethingslab.com/resources/misc08/xenfb-adventures-10.pdf>
- [17] Y. Bulygin and D. Samyde, “Chipset based approach to detect virtualization malware a.k.a. DeepWatch,” *Blackhat USA*, 2008.
- [18] D. Murray, G. Milos, and S. Hand, “Improving Xen security through disaggregation,” in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 2008, pp. 151–160.
- [19] 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*, ser. CCS ’10, 2010, pp. 38–49.
- [20] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh, “Copilot - a coprocessor-based kernel runtime integrity monitor,” in *SSYM’04: Proceedings of the 13th conference on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2004, pp. 13–13.
- [21] F. Bellard, “QEMU, a fast and portable dynamic translator,” in *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.
- [22] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer, “Xen and the art of virtualization,” in *In Proceedings of the ACM Symposium on Operating Systems Principles*, 2003.
- [23] “Intel Corp. Intel I/O Controller Hub 9 (ICH9) Family Datasheet (2008) .”
- [24] Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba, “ACPI, <http://www.acpi.info/>.”
- [25] 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.

- [26] A. Baliga, V. Ganapathy, and L. Iftode, “Automatic inference and enforcement of kernel data structure invariants,” in *ACSAC '08: Proceedings of the 2008 Annual Computer Security Applications Conference*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 77–86.
- [27] B. D. Carrier and J. Grand, “A hardware-based memory acquisition procedure for digital investigations,” *Digital Investigation*, vol. 1, no. 1, pp. 50 – 60, 2004. [Online]. Available: <http://www.sciencedirect.com/science/article/B7CW4-4BMXXJS-B/2/74a97eb4c390ef6a2a2f8111c3a59aa5>
- [28] J. Rutkowska and R. Wojtczuk, “Preventing and detecting Xen hypervisor subversions,” *Blackhat Briefings USA*, 2008.
- [29] R. Hiremane, “Intel® Virtualization Technology for Directed I/O (Intel® VT-d),” *Technology© Intel Magazine*, vol. 4, no. 10, 2007.
- [30] G. Coker, “Xen security modules (xsm),” *Xen Summit*, 2006.
- [31] R. Sailer, E. Valdez, T. Jaeger, R. Perez, L. Van Doorn, J. Griffin, and S. Berger, “sHype: Secure hypervisor approach to trusted virtualized systems,” *IBM Research Report RC23511*, 2005.
- [32] F. Wecherowski and core collapse, “A Real SMM Rootkit: Reversing and Hooking BIOS SMI Handlers,” *Phrack Magazine*, 2009.
- [33] BSDaemon, coideloko, and D0nAnd0n, “System Management Mode Hack: Using SMM for ”Other Purposes”,” *Phrack Magazine*, 2008.
- [34] 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*. ACM, 2008, p. 11.
- [35] L. Dufflot, D. Etiemble, and O. Grumelard, “Using CPU System Management Mode to Circumvent Operating System Security Functions,” in *Proceedings of the 7th CanSecWest conference*. Citeseer, 2001.
- [36] B. Lampson, “Privacy and security: Usable security: how to get it,” *Commun. ACM*, vol. 52, pp. 25–27, November 2009. [Online]. Available: <http://doi.acm.org/10.1145/1592761.1592773>
- [37] “Phoenix HyperSpace.” [Online]. Available: http://en.wikipedia.org/wiki/Phoenix_Technologies#HyperSpace
- [38] X. Chen, T. Garfinkel, E. Lewis, P. Subrahmanyam, C. Waldspurger, D. Boneh, J. Dvoskin, and D. Ports, “Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems,” in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*. ACM, 2008, pp. 2–13.
- [39] R. Meushaw and D. Simard, “Nettop-commercial technology in high assurance applications,” *VMware Tech Trend Notes*, 2000.

- [40] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. L. Griffin, and L. van Doorn, "Building a mac-based security architecture for the xen open-source hypervisor," *Computer Security Applications Conference, Annual*, vol. 0, pp. 276–285, 2005.
- [41] J. Wang, A. Stavrou, and A. Ghosh, "HyperCheck: A hardware-assisted integrity monitor," in *Recent Advances in Intrusion Detection*. Springer, 2010, pp. 158–177.
- [42] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," *IEEE Symposium on Security and Privacy*, pp. 233–247, 2008.
- [43] R. Wojtczuk, "Adventures with a certain Xen vulnerability (in the PVFB backend), <http://www.invisiblethingslab.com/resources/misc08/xenfb-adventures-10.pdf>."
- [44] "Hyper-V architecture." [Online]. Available: [http://msdn.microsoft.com/en-us/library/cc768520\(BTS.10\).aspx](http://msdn.microsoft.com/en-us/library/cc768520(BTS.10).aspx)
- [45] D. Chisnall, *The definitive guide to the Xen hypervisor*. Prentice Hall Press Upper Saddle River, NJ, USA, 2007.
- [46] PCI-SIG, "PCI Express 2.0 Frequently Asked Questions." [Online]. Available: http://www.pcisig.com/news_room/faqs/pcie2.0_faq/
- [47] Intel, "Intel® 64 and ia-32 architectures software developers manual volume 1."
- [48] "Unified Extensible Firmware Interface, <http://www.uefi.org/home/>."
- [49] "Coreboot, <http://coreboot.org/>."
- [50] Advanced Micro Devices, Inc., "BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors, April 22, 2010."
- [51] MITRE, "Cve-2007-4993." [Online]. Available: <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-4993>
- [52] 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
- [53] L. Dufлот, D. Etiemble, and O. Grumelard, "Security issues related to pentium system management mode," in *Cansecwest security conference Core06*, 2006.
- [54] L. Dufлот, O. Levillain, B. Morin, and O. Grumelard, "Getting into the SMRAM: SMM Reloaded," *CanSecWest, Vancouver, Canada*, 2009.
- [55] G. Hoglund and J. Butler, *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.
- [56] K. Adamyse, "Handling interrupt descriptor table for fun and profit. Phrack 59," 2002.
- [57] M. Burdach, "Digital forensics of the physical memory," *Warsaw University*, 2005.

- [58] T. Vidas, “The acquisition and analysis of random access memory,” *Journal of Digital Forensic Practice*, vol. 1, no. 4, pp. 315–323, 2006.
- [59] S. Schreiber, *Undocumented Windows 2000 secrets: a programmer’s cookbook*. Addison-Wesley, 2001.
- [60] D. Bovet and M. Cesati, *Understanding the Linux kernel 3rd edition*. O’Reilly Media, 2005.
- [61] J. Masters, “Simple SMI detector,” January 2009. [Online]. Available: <http://lwn.net/Articles/316622/>
- [62] G. Hunt and D. Brubacher, “Detours: Binary interception of Win32 functions,” in *Proceedings of the 3rd conference on USENIX Windows NT Symposium-Volume 3*. USENIX Association, 1999, p. 14.
- [63] H. Shacham, “The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86),” in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, p. 561.
- [64] S. Checkoway, L. Davi, A. Dmitrienko, A. Sadeghi, H. Shacham, and M. Winandy, “Return-oriented programming without returns,” in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, p. 561.
- [65] v14d, “LKL Linux KeyLogger.” [Online]. Available: <http://sourceforge.net/projects/lkl/>
- [66] PENDRAGON., “CaffeineMark 3.0.” [Online]. Available: <http://www.benchmarkhq.ru/cm30/>
- [67] “Mitre cve vulnerability database.” [Online]. Available: <http://cve.mitre.org/>
- [68] J. Rutkowska, “Beyond the CPU: Defeating hardware based RAM acquisition,” *Proceedings of BlackHat DC 2007*, 2007.
- [69] S. King and P. Chen, “SubVirt: implementing malware with virtual machines,” in *Security and Privacy, 2006 IEEE Symposium on*. IEEE, pp. 14–pp.
- [70] I. VIA Technologies, “VT8237R South Bridge, Revision 2.06,” December 2005.
- [71] “vbetool.” [Online]. Available: <http://linux.die.net/man/1/vbetool>
- [72] “AT Attachment specification, <http://www.t13.org/>.”
- [73] J. R. Okajima, “Aufs.” [Online]. Available: <http://aufs.sourceforge.net/>
- [74] “Seabios.” [Online]. Available: <http://www.coreboot.org/SeaBIOS>
- [75] “Ftrace.” [Online]. Available: <http://elinux.org/Ftrace>
- [76] T. C. Group, “Trusted platform module main specification. version 1.2, revision 103, 2007.” [Online]. Available: http://www.trustedcomputinggroup.org/resources/tpm_main_specification

- [77] CAIDA, “Caida analysis of code-red.” [Online]. Available: <http://www.caida.org/research/security/code-red/>
- [78] X-Force, “SQL Slammer worm propagation.” [Online]. Available: <http://xforce.iss.net/xforce/xfdb/11153>
- [79] G. G. Richard, III and V. Roussev, “Next-generation digital forensics,” *Commun. ACM*, vol. 49, pp. 76–80, February 2006. [Online]. Available: <http://doi.acm.org/10.1145/1113034.1113074>
- [80] F. Adelstein, “Live forensics: diagnosing your system without killing it first,” *Commun. ACM*, vol. 49, pp. 63–66, February 2006. [Online]. Available: <http://doi.acm.org/10.1145/1113034.1113070>
- [81] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, “Lest we remember: cold boot attacks on encryption keys,” in *Proceedings of the 17th conference on Security symposium*. Berkeley, CA, USA: USENIX Association, 2008, pp. 45–60. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1496711.1496715>
- [82] AMD, “IOMMU Architectural Specification.”
- [83] PCI-SIG, “PCI Local Bus Specification Revision 3.0,” September 2010. [Online]. Available: <http://www.pcisig.com/specifications/conventional/>
- [84] Intel, HP, NEC and Dell, “IPMI intelligent platform management interface specification second generation v2.0. .”
- [85] “GDB manual.” [Online]. Available: <http://www.gnu.org/software/gdb/documentation/>
- [86] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, “Mapping kernel objects to enable systematic integrity checking,” in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 555–565.
- [87] G. Necula, S. McPeak, S. Rahul, and W. Weimer, “Cil: Intermediate language and tools for analysis and transformation of c programs,” in *Proceedings of the 11th International Conference on Compiler Construction*. Springer-Verlag, 2002, pp. 213–228.
- [88] R. PALEARI, A. FATTORI, L. MARTIGNONI, and L. CAVALLARO, “Live and trustworthy forensic analysis of commodity production systems,” in *Recent advances in intrusion detection: 13th international symposium, RAID 2010, Ottawa, Ontario, Canada, September 15-17, 2010. proceedings*. Springer, 2010.

Curriculum Vitae

Jiang Wang received his Bachelor and Master of Engineering from Zhejiang University in China in 1998 and 2001 respectively. Afterwards, he worked in Synway Digital Information Co. Ltd, Hangzhou, China for approximately three years. He then studied at George Mason University in Fairfax, Virginia, and received his PhD in Information Technology in 2011.