Department of Computer Science
George Mason University Technical Reports

4400 University Drive MS#4A5
Fairfax, VA 22030-4444 USA
http://cs.gmu.edu/ 703-993-1530

An Analysis of System Management Mode (SMM)-based Integrity Checking Systems and Evasion Attacks

Jiang Wang jwanga@gmu.edu
Kun Sun ksun3@gmu.edu
Angelos Stavrou astavrou@gmu.edu

Technical Report GMU-CS-TR-2011-8

Abstract

System Management Mode (SMM) is an x86 processor feature designed to assist debugging for hardware manufacturers. Recent research has shown that SMM can also be used to protect the run-time integrity of software by invoking SMM to periodically check current system state and compare it with known pristine or trusted software states. Researchers and practitioners have claimed that any unauthorized state modification can be detected with an SMM-based system integrity-checking mechanism.

In this paper, we demonstrate that all hardware-based, periodic integrity mechanisms can be defeated by a new class of attacks, which we refer to as "evasion attacks." Such attacks use a compromised software stack to remove any attack traces before the integrity checks begin and to continue the execution of the malicious code after the integrity checks are completed. We detail two categories of evasion attacks: directly-intercepting System Management Interrupt (SMI) and indirectly-deriving SMI invocations. Finally, we measure the performance impact of our proof-of-concept prototypes for all of the attacks and present countermeasures for these attacks.

1 Introduction

System Management Mode (SMM) is an x86 CPU mode present on all modern processors that creates an *isolated* and *trusted* environment where the developer can safely execute pre-stored BIOS code. System Management RAM (SMRAM), the memory used by SMM, can be locked so that even the privileged software in the protected mode of the CPU cannot access it. SMM is entered through a special interrupt called System Management Interrupt (SMI). SMI can be triggered by the SMI interrupt pin on the processor or through Advanced Programmable Interrupt Controller (APIC) [10].

SMI is the highest hardware interrupt on an x86 system, higher than both normal interrupt and non-maskable interrupt (NMI). Therefore, SMM code holds a unique position to monitor the software running on top of it, whether it is an operating system or a hypervisor[1]. Recently, many efforts, including HyperGuard [18], HyperCheck [24], and HyperSentry [1], utilize the SMM to monitor the integrity of the hypervisors. 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 paper, we demonstrate that the SMI triggering and memory validation mechanisms can 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 also clean up the attack trace each time before the integrity measurement starts. Furthermore, it can reload itself to the system after the integrity measurement ends or at a later point in time. This class of attacks is feasible because, when in SMM, the CPU halts the execution of regular programs. Therefore, the protection mechanisms must be carefully crafted to consume little 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 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 hyper-

[1] Also called Virtual Machine Monitors (VMMs)

visor (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 [11] 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.

In summary, we make the following contributions:

1. Provide a systematic analysis of different classes of evasion attacks.

2. Implement prototypes that demonstrate various evasion attacks and show the effectiveness of these attacks.

3. Develop defense mechanisms to curtail evasion attacks.

4. Measure the overhead and detection rate of different evasion attacks on commodity hardware.

2 Background

System Management Mode (SMM) was first introduced in the Intel386 SL and Intel486 SL processors. It became a standard IA-32 feature in the Pentium processor [10]. SMM is a separate x86 processor mode from protected mode or real-address mode. The original purpose of SMM was to provide 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 APIC.

In SMM, the processor switches to a separate address space, referred to as System Management RAM (SMRAM). All hardware context of the current processor running code is saved in SMRAM. The CPU, being in SMM, then transparently executes code that is usually a part of BIOS and resides in SMRAM. The SMRAM can be rendered inaccessible within other CPU operating modes. Therefore, it can act as trusted storage that cannot be accessed by any device, or even by the CPU when it is not in SMM mode.

SMM cannot be accessed or modified by hypervisors or by operating systems that run in protected mode; SMM is currently used for hypervisor integrity checking, which can verify that the code of the hypervisor or of the operating system software has not been compromised by malicious code [24, 1]. Figure 1 presents a typical model for SMM-based integrity checking. When the processor receives a System Management Interrupt (SMI) event, it will switch from Protected Mode to System Management Mode (SMM). The integrity checking code is then run to verify that the hypervisor or operating system has not been compromised. When *rsm* instruction is executed, the processor switches from SMM to Protected Mode, where the processor will resume its execution path.

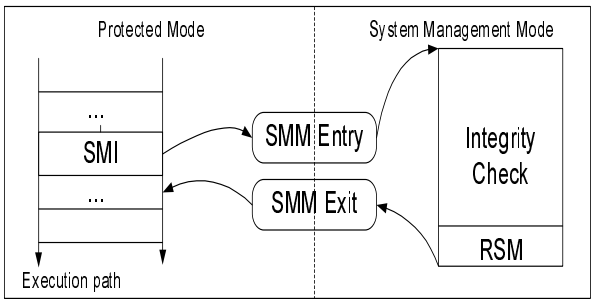

Figure 1: SMM-based System Integrity Checking: the System State is Inspected Only While in SMM.

Although each CPU chip has only one SMI pin, many hardware events can trigger an SMI. The concrete reason for entering SMM is normally defined by I/O Controller Hub (ICH) for Intel chipsets and reported by hardware. For example, in ICH4, there are **39** events that can trigger SMI [9]. These include power management events, USB events, Total Cost of Ownership (TCO) events, writing to 0xB2 port, periodic timer expiration, and SMBus-related events. Newer ICH chips, such as ICH5, ICH9, and ICH10, have similar functions. To enable an SMI event, both a global bit and an individual control bit on registers of ICH4 need to be set. Most of the SMI events have an individual enable bit. However, for ICH4, nine SMI events cannot be individually disabled. Writing to an 0xB2 port is one such event. Moreover, there are registers on ICH to record the reason for entering SMI.

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, since the operating system is suspended during SMM, they must run in a short time, and then exit from 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. **HyperGuard** [18] 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. **HyperCheck** [24] 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. **HyperSentry** [1] 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.

3 Threat Model

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.

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 runs only once for several seconds, and that compromising the hypervisor may take just a few instructions. Evasion attack can be accomplished using the following two mechanisms: either *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. Moreover, we discuss how to launch evasion attacks to randomized SMM-checking mechanisms.

3.1 Type I: Directly Intercepting SMI

Figure 2 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 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.

3.1.1 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 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.

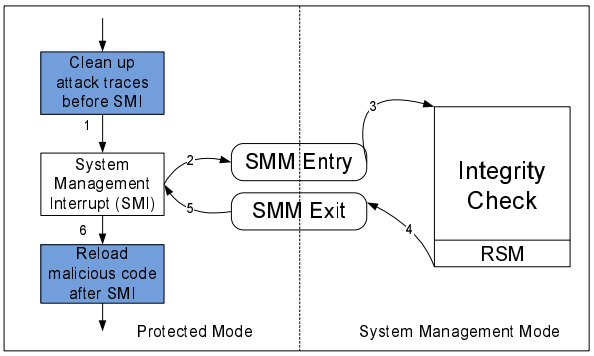

Figure 2: Directly Intercepting SMI. The attacker inserts a code preamble before the SMM-based integrity checks can occur.

***Scenario 1:* An attacker cannot reissue the same SMI triggering event as the intercepted one.**
In HyperCheck [24], the SMI-based integrity checking is triggered by a PCI network card, 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 5.2.1. 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 [1].

***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

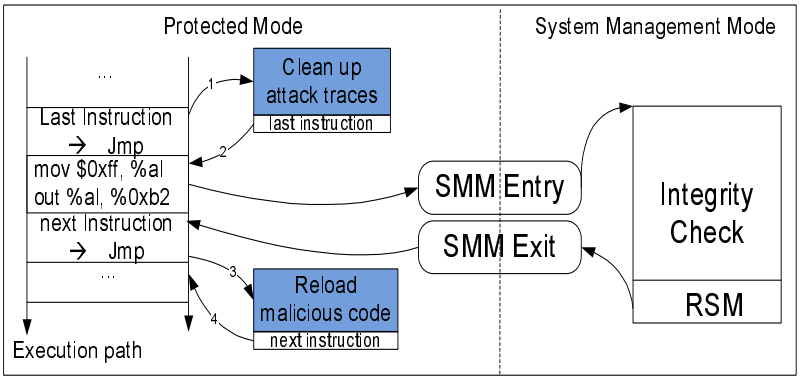

Figure 3: Intercept SMI triggering through write to port 0xB2. This attack requires the scanning of memory to identify the call location for 0xB2

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 3. 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, %al
e6 b2 out %al, %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 [8].

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 4. 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 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

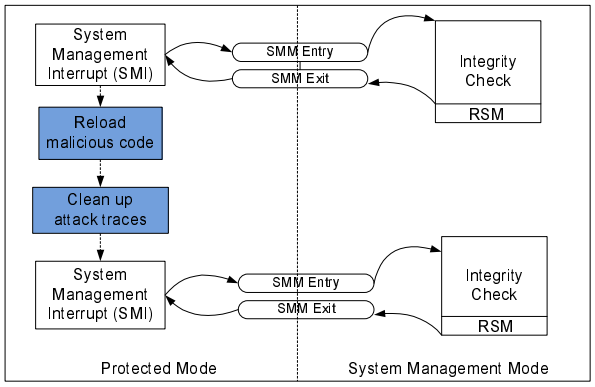

Figure 4: Launch of attacks between two SMIs.

event. We refer to the component that derives SMI time information as the *SMI detector*.

3.2.1 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 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 [11].

An SMI detector may detect the existence of SMI in many ways. Figure 5 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

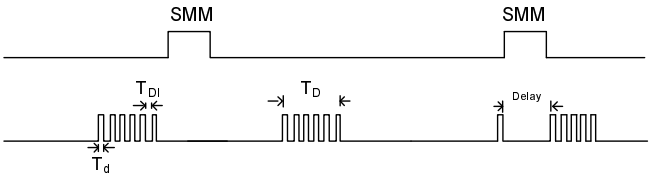

Figure 5: Detecting the invocation of SMI: T_D denotes the detection duration while we check the Time Stamp Counter every $T_d + T_{DI}$, where T_d is the duration time for one counter checking process and T_{DI} is the time interval between two counter checking processes.

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.

3.2.2 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.

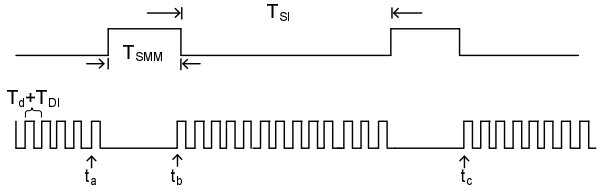

Figure 6: Measuring the SMM duration(T_{SMM}): T_{SI} is the time interval between two SMIs. $T_d + T_{DI}$ is the time delay between two counter polling processes when the system is not in SMM mode. t_a is the last polling time before the system enters SMM, and t_b is the first polling time after the system exits SMM.

Method 1: Continuous SMI Detector

Figure 6 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 6 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. 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.

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 4.2.

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.

4 Defense Strategies

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.

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.

4.1.1 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.

4.1.2 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.

4.1.3 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. Since attackers may disable or reroute SMI, the defender should be alerted when some SMI events are missed.

When the defender carefully chooses the pseudo-random number generator (PRNG) functions and parameters for the minimum and maximum time intervals, he/she can help to alleviate Type III evasion attacks. The smaller the minimum interval is, the more difficult it becomes for the attacker to launch Type III attacks.

4.2 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 ensure 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. Type II and Type III attacks require the help of SMI detectors, which reside in the memory at the beginning of 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.

4.2.1 Checking SMI Events

HyperCheck [24] 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 Scenario 1 of the Type I attacks.

To prevent such an attack, the SMM code should check the SMI triggering events to ensure that it can only be triggered by a specific SMI event. HyperSentry [1] 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.

4.2.2 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.

4.2.3 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.

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 3.

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.

5.2 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.

5.2.1 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.

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 [10], 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.

5.2.2 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 μs and 18 μs, so we set the threshold as 20 μs.

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.

5.2.3 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 [21, 4] 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 [21, 4], 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 [23]. After running 100 times, the average loading time was 1.232 μs, and the average unloading time was 1.637 μs. In total, it was almost 3 μs. The actual running time depends on the attacking code.

6 Evaluation

Our experimental results show that the evasion attacks are effective at evading the existing SMM-based integrity checking solutions, such as HyperCheck [24] and HyperSentry [1]. 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 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 [6] 3.1 and Linux 2.6.18 were installed on the physical machine, and the Domain 0 is CentOS 5.4.

6.1 Performance Analysis

6.1.1 System Overhead of SMI Detector

The source code (in C) 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. 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 [15], which 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 amount of time taken to execute the test. Because CaffeineMark uses an internal scoring metric, it is useful only for relative comparisons.

Figure 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.

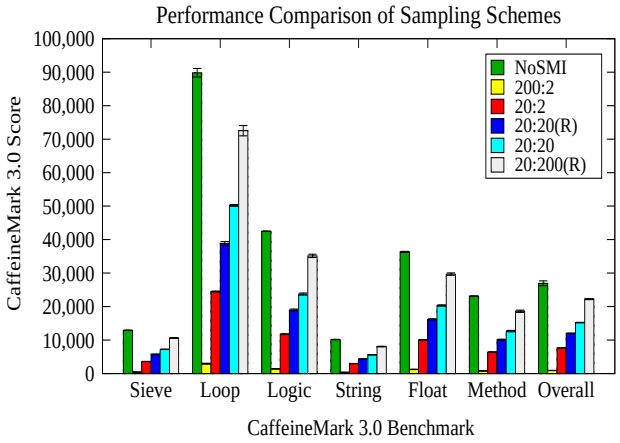

Figure 7: CaffeineMark 3.0 Micro-benchmarks of System Performance. The legend notation is "sample duration:check interval" in milliseconds.

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 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 the 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 7 indicate 95% confidence intervals. System performance is at its best when no SMI detector is running; it goes down with each increase of the SMI sampling duration, and it goes up with each increase of the SMI sampling time interval. 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 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.

The system overhead decreases 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 "2,000:2" in our experiments, but the system halted.

Table 1: System Overhead of SMI Detector

Sampling	2:2	2:20	2:20(R)	2:200(R)	2:2,000(R)	No SMI
Overhead(s)	2.379	2.389	2.343	2.381	2.347	2.332

Table 1 shows the results when we set the SMI sampling time to 2 ms and the SMI sampling interval to 2, 20, 200, and 2,000 ms. The overhead was not significantly affected

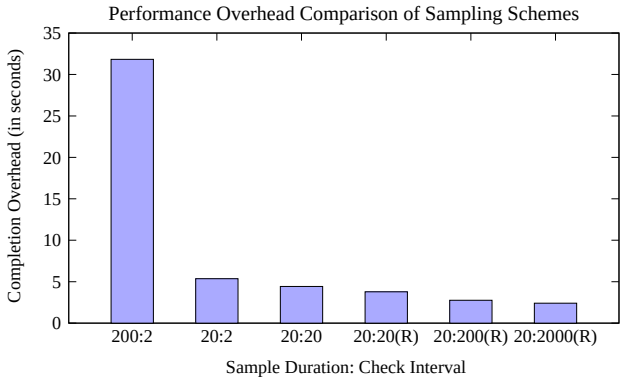

Figure 8: System Overhead of SMI Detector

as we decreased the checking intervals between SMI samples. This means that if an attacker reduces the SMI sampling time to a small number, this won't have an impact on system performance. However, increasing the checking interval will dramatically decrease the detection probability of the SMI events.

6.1.2 Detection Probability of SMI Detector

Figure 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.

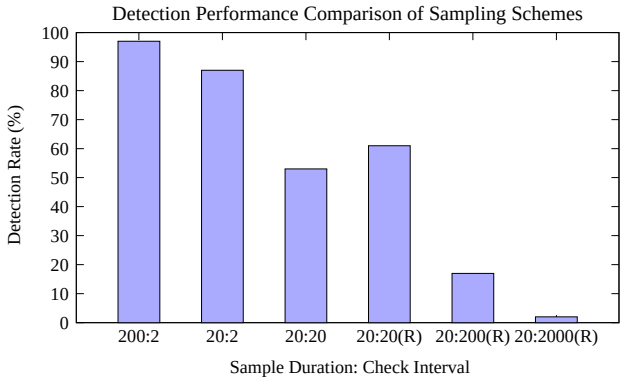

Figure 9: SMI Detector's Detection Probability.

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)."

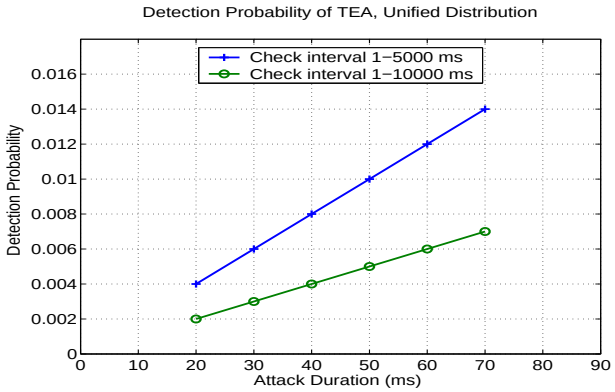

Figure 10: Detection Probability of TEA, uniform distribution.

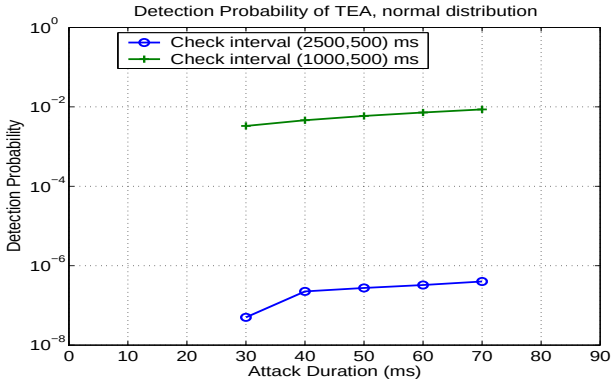

Figure 11: Detection Probability of TEA, normal distribution

We simulated these two attack types by using two programs written in Matlab. The results for TEA are shown in Figure 10 and Figure 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 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 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 2 and Table 3. From Table 2, we can see that the detection probability is mostly determined by the SMM checking intervals. The attack intervals do not affect the detection probability too much. From Table 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 2: Detection probability of NTEA: intervals are **uniformly distributed.**

SMM interval(ms)		1,000		
Attack interval(ms)	500	1,000	2,000	3,000
Detection probability	5.40%	6.00%	5.80%	6.20%

Table 3: Detection probability of NTEA: intervals are **normally distributed.**

SMM(ms)	1,000	2,500	5,000	10,000
Detection probability	2.40%	1.40%	0.80%	0.25%

7 Related Work

A plethora of existing research proposes various methods of isolating and protecting the integrity of software, including the operating system kernels or hypervisors. Some earlier approaches, including Copilot [16] and Gibraltar [2], employed PCI devices to directly examine the physical memory. However, the PCI device-based method is no more reliable than the SMM-based method since the PCI devices can be manipulated to obtain a different view. They are also vulnerable to evasion attacks. Another approach is to introduce in-hypervisor hooks [5, 25] and enforce security policies between virtual machines [19], which are hypervisor-specific and run at the same level as the hypervisor.

Furthermore, there have been many attempts to protect against attacks by minimizing the code footprint and relying on the Trusted Computing Base (TCB) for current commercial hypervisors [13, 12, 7, 14, 20, 22]. These approaches aim to provide a minimal layer, thus limiting the code exposure and subsequent attack surface for the hypervisor code. However, due to third-party driver code, they cannot offer strong guarantees regarding the code integrity of all hypervisor components.

In addition to the software-based mechanisms, researchers have proposed employing commodity and specialized hardware components to assist in the protection of software integrity [26, 3, 18, 24, 1]. Among the hardware-based solutions, many (e.g., HyperGuard [18], HyperCheck1.0 [24] and HyperSentry [1]) depend on the System Management Mode (SMM) that exists on the current x86 CPU to provide an isolated and trusted environment. SMM is a separated CPU mode whose memory (called SMRAM) can be locked so that even the privileged software in the protected or real-address mode cannot access it.

More specifically, HyperGuard [18] has suggested using the SMM of the x86 CPU to monitor the integrity of the hypervisors. It used a hardware timer to periodically trigger SMI and checked the reason for SMI [17]. On the other hand, HyperCheck [24] used SMM and a network card to detect attacks against the hypervisor kernel. SMM was used to obtain the CPU register context and transmit it to a network-located console for further validation. However, HyperCheck does not examine the cause of an SMI. Therefore, it is vulnerable to direct SMI interception attacks and indirect evasion attacks. In HyperSentry [1], the authors coined the term "scrubbing attack," which attempts to masquerade a valid SMI by triggering a software SMI. Such attacks are easy to prevent through the use of IPMI and BMC to trigger SMI and make it difficult for the attacker to trigger the same SMI. Unfortunately, as our analysis in section 3 shows, HyperSentry is still vulnerable to indirect evasion attacks.

8 Conclusions

We presented a systematic analysis of evasion attacks for SMM-based integrity monitors. In such attacks, the adversary can circumvent the defense mechanisms by interposing before and after the SMI invocation mechanism. We point out that evasion attacks can be accomplished either directly (by intercepting SMI) or indirectly (by measuring the behavior hardware timers). Furthermore, we implemented these attacks on unmodified commodity x86 software and hardware components. Our study shows that evasion attacks are both realistic and easy to mount. Moreover, through experimentation on our prototype implementation, we quantified the performance overhead and detection capabilities of evasion attacks. Finally, we discussed potential countermeasures to mitigate the introduced threats, highlighting the advantages and caveats that would influence the future design of hardware-assisted monitors.

References

[1] AZAB, A., NING, P., WANG, Z., JIANG, X., AND

ZHANG, X. HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS 2010)* (2010).

[2] BALIGA, A., GANAPATHY, V., AND IFTODE, L. Automatic inference and enforcement of kernel data structure invariants. In *ACSAC '08: Proceedings of the 2008 Annual Computer Security Applications Conference* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 77–86.

[3] BULYGIN, Y., AND SAMYDE, D. Chipset based approach to detect virtualization malware a.k.a. DeepWatch. *Blackhat USA* (2008).

[4] CHECKOWAY, S., DAVI, L., DMITRIENKO, A., SADEGHI, A., SHACHAM, H., AND WINANDY, M. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security* (2010), ACM, p. 561.

[5] COKER, G. Xen security modules (xsm). *Xen Summit* (2006).

[6] DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., PRATT, I., WARFIELD, A., BARHAM, P., AND NEUGEBAUER, R. Xen and the art of virtualization. In *In Proceedings of the ACM Symposium on Operating Systems Principles* (2003).

[7] HEISER, G., AND LESLIE, B. The OKL4 Microvisor: Convergence point of microkernels and hypervisors. In *Proceedings of the first ACM asia-pacific workshop on Workshop on systems* (2010), ACM, pp. 19–24.

[8] HUNT, G., AND BRUBACHER, D. Detours: Binary interception of Win32 functions. In *Proceedings of the 3rd conference on USENIX Windows NT Symposium-Volume 3* (1999), USENIX Association, p. 14.

[9] INTEL CORP. Intel® 82801DB I/O Controller Hub 4 (ICH4) Datasheet, May 2002.

[10] INTEL CORP. Intel® 64 and IA-32 Architectures Software Developer's Manual, June 2010.

[11] MASTERS, J. Simple smi detector, http://lwn.net/articles/316622/, January 2009.

[12] MCCUNE, J., PARNO, B., PERRIG, A., REITER, M., AND ISOZAKI, H. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008* (2008), ACM, pp. 315–328.

[13] MCCUNE, J. M., LI, Y., QU, N., ZHOU, Z., DATTA, A., GLIGOR, V., AND PERRIG, A. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2010).

[14] MURRAY, D., MILOS, G., AND HAND, S. Improving Xen security through disaggregation. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (2008), ACM, pp. 151–160.

[15] PENDRAGON. CaffeineMark 3.0, http://www.benchmarkhq.ru/cm30/.

[16] PETRONI, JR., N. L., FRASER, T., MOLINA, J., AND ARBAUGH, W. A. Copilot - a coprocessor-based kernel runtime integrity monitor. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium* (Berkeley, CA, USA, 2004), USENIX Association, pp. 13–13.

[17] PHOENIX. Hypervisor security using smm, http://www.faqs.org/patents/app/20100057982.

[18] RUTKOWSKA, J., AND WOJTCZUK, R. Preventing and detecting Xen hypervisor subversions. *Blackhat Briefings USA* (2008).

[19] SAILER, R., VALDEZ, E., JAEGER, T., PEREZ, R., VAN DOORN, L., GRIFFIN, J., AND BERGER, S. sHype: Secure hypervisor approach to trusted virtualized systems. *IBM Research Report RC23511* (2005).

[20] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. 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* (2007), ACM, p. 350.

[21] SHACHAM, H. 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* (2007), ACM, p. 561.

[22] SHINAGAWA, T., EIRAKU, H., TANIMOTO, K., OMOTE, K., HASEGAWA, S., HORIE, T., HIRANO, M., KOURAI, K., OYAMA, Y., KAWAI, E., ET AL. BitVisor: a thin hypervisor for enforcing i/o device security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (2009), ACM, pp. 121–130.

[23] V14D. LKL Linux KeyLogger, http://sourceforge.net/projects/lkl/.

[24] WANG, J., STAVROU, A., AND GHOSH, A. K. HyperCheck: A Hardware-Assisted Integrity Monitor. In *Proceedings of the 13th International Symposium on Recent Advances in Intrusion Detection (RAID 2010)* (2010).

[25] WANG, Z., AND JIANG, X. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Proceedings of the IEEE Symposium on Security and Privacy* (2010).

[26] WOJTCZUK, R. Subverting the Xen hypervisor. *Black-Hat USA* (2008).