HyperCheck: A Hardware-Assisted Integrity Monitor

Fengwei Zhang, Jiang Wang, Kun Sun, and Angelos Stavrou

Abstract—The advent of cloud computing and inexpensive multi-core desktop architectures has led to the widespread adoption of virtualization technologies. Furthermore, security researchers embraced virtual machine monitors (VMMs) as a new mechanism to guarantee deep isolation of untrusted software components, which, coupled with their popularity, promoted VMMs as a prime target for exploitation. In this paper, we present HyperCheck, a hardware-assisted tampering detection framework designed to protect the integrity of hypervisors and operating systems. Our approach leverages System Management Mode (SMM), a CPU mode in $\times 86$ architecture, to transparently and securely acquire and transmit the full state of a protected machine to a remote server. We have implement two prototypes based on our framework design: *HyperCheck-I* and *HyperCheck-II*, that vary in their security assumptions and OS code dependence. In our experiments, we are able to identify rootkits that target the integrity of both hypervisors and operating systems. We show that HyperCheck can defend against attacks that attempt to evade our system. In terms of performance, we measured that HyperCheck can communicate the entire static code of Xen hypervisor and CPU register states in less than 90 million CPU cycles, or 90 ms on a 1 GHz CPU.

Index Terms—Hypervisor, system management mode, kernel, Coreboot

1 Introduction

Virtualization technologies have become the de facto standard in server consolidation because they decrease the energy footprint and cost of management of modern computing clusters. Additionally, hypervisors are increasingly used as components to enforce system security and resilience [1], [2], [3], [4], [5], [6], [7].

Due to their widespread adoption, hypervisors have attracted the attention of attackers. National vulnerability data [8] shows that there are 73 security vulnerabilities in Xen and 30 vulnerabilities in VMWare ESX. Moreover, a number of virtual machine escape attacks [9], [10], [11] and hypervisor rootkits [12] are widely deployed. Security researchers have noticed this problem and have begun to improve hypervisor security [13], [14], [15].

The increase of vulnerabilities and the observed attack trends have spurred research towards reducing the Trusted Code Base (TCB) of current commercial hypervisors [16]. Others developed new specialized prototypes of hypervisors [6], [17]. However, having a small code base can only limit the code exposure and reduce the attack surface of the hypervisor—it cannot provide a strong guarantee about the code integrity of all the hypervisor components.

To address these limitations and to complement the existing protection mechanisms, we design a hardware-assisted tampering detection framework called HyperCheck to protect the integrity of hypervisors or operating systems. To achieve that, HyperCheck harnesses the CPU System Management Mode (SMM) present in all x86 commodity systems to create a snapshot view of the current states of the CPU and memory of the protected machine. This information is securely transmitted using a network card to a remote monitor machine. Then, the monitor machine can identify any tampering by comparing the newly generated snapshot with the one recorded when the machine was initialized. If the two views do not match, a human operator is notified for further investigation. As shown in Fig. 1, HyperCheck works at the Basic Input-Output System (BIOS) level and can protect the software above. We assume that the attacker does not have physical access to the machine and the SMM is locked so that it cannot be modified after booting. We will discuss more BIOS attacks in Section 7.

Unlike previous work [18] with specialized PCI hardware, we are able to acquire a complete view of the target machine, including the entire physical memory and CPU registers using SMM and a generic PCI hardware device. In addition, our approach is able to thwart attacks aimed at disabling and blocking the PCI device because the monitor machine can be used to detect a denial-of-service attack. To evaluate the security and performance of our framework, we implement two prototypes of HyperCheck framework, *HyperCheck-I* and *HyperCheck-II*, on real, physical machines. They are implemented and deployed on two different machines depending on the availability of the BIOS source code. In HyperCheck-I where the BIOS is closed source, we perform reverse engineering on BIOS to make changes and rely on a kernel module to setup network packets that are transmitted by SMM. To reduce the development

- *F. Zhang and A. Stavrou are with the Department of Computer Science, George Mason University, 4400 University Drive, Room 437, Research Hall, Fairfax, VA 22030. E-mail: {fzhang4, astavrou}@gmu.edu.*
- *J. Wang is with the Riverbed Technology, 46864 Winema Common, Fremont, CA 94539. E-mail: jwangzju@gmail.com.*
- *K. Sun is with the Center for Secure Information Systems, George Mason University, 4400 University Drive, Room 417, Research Hall, Fairfax, VA 22030. E-mail: ksun3@gmu.edu.*

Manuscript received 5 Feb. 2013; revised 3 Nov. 2013; accepted 1 Dec. 2013. Date of publication 11 Dec. 2013; date of current version 16 July 2014.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TDSC.2013.53

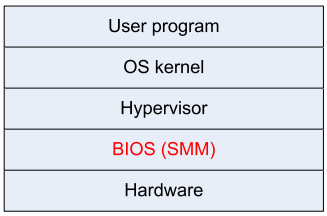

Fig. 1. HyperCheck can offer protection to services running above the BIOS.

complexity and overcome the hardware limitation, we leverage an open-source BIOS called Coreboot [19] in HyperCheck-II to add a trusted network driver and secure packet transmission in SMM.

Our prototypes are able to expose rootkits aimed at Xen hypervisor, Xen Domain 0, Linux and Windows. In addition, HyperCheck is able to defend evasion attacks existing in the polling-based systems. We use a PCI network card to randomly trigger SMI and check the triggering reason in SMM. Furthermore, we randomize the time spent in SMM so that the attacker cannot accurately predict when SMM exits. Section 5.5 details our methods.

Our approach is agnostic to the underlying system and it is straightforward to extend our system to protect any other hypervisors (e.g., VMWare ESX) or OSes. Our experimental results indicate that HyperCheck introduces a reasonable overhead and only requires 90 million CPU cycles to completely transmit 2.8 MB of Xen code and Domain 0 code. In addition, we analyze the transmitting packet size to optimize the network delay. To measure the overall system impact of HyperCheck, we employ a popular benchmark to test the system overhead by varying the sampling intervals.

In summary, we make the following contributions:

1. Design a novel hardware-assisted tampering detection framework that creates a complete snapshot of the states of the system with commercial hardware and no modifications to the installed software.
2. Demonstrate that HyperCheck can securely and transparently transmit the CPU registers, static code and control data of hypervisors or OSes to a remote server. HyperCheck is able to thwart evasion attacks against SMM-based polling systems.
3. Implement two HyperCheck prototypes. HyperCheck-I is implemented on real hardware with closed-source BIOS; HyperCheck-II improves the security and reduces the development complexity of HyperCheck-I by using an open-source BIOS called Coreboot.
4. Both HyperCheck-I and HyperCheck-II introduce a reasonable overhead. Both systems can successfully detect rootkits and code integrity attacks against the Xen hypervisor, Xen Domain 0, Linux and Windows.

This paper is an expanded version of our previous work [14] published in RAID 2010 and is organized as follows. Section 2 provides background information on SMM and BIOS. Section 3 discusses the threat model and assumptions. Section 4 presents the HyperCheck framework. Section 5 details the two prototype implementations, and we evaluate them in Section 6. Section 7 provides the security analysis and discusses the limitations. Section 8 discusses related work, and Section 9 concludes this paper.

2 Background

2.1 System Management Mode

System Management Mode is a separate CPU mode from the protected mode and real mode. It provides a transparent mechanism for implementing system-control functions, such as power management and system security. SMM is implemented by the Basic Input-Output System. SMM is entered via the system management interrupt (SMI) when the SMM interrupt pin is asserted. The microprocessor automatically saves its entire state in a separate address space known as system management ram (SMRAM) and enters SMM to execute an SMI handler. The program executes the rsm instruction to exit SMM. The SMRAM is inaccessible from other CPU modes (while not in SMM); therefore, it can act as a trusted storage space.

2.2 BIOS, UEFI and Coreboot

BIOS is an indispensable component for all computers. The main function of the BIOS is to initialize the hardware devices, including the processor, main memory, chipsets, hard disk, and other necessary IO devices. BIOS code is normally stored on a non-volatile ROM chip on the motherboard. In recent years, a new generation of BIOS, referred to as unified extensible firmware interface (UEFI) [20], has become increasingly popular in the market. UEFI is a specification that defines the new software interface between OS and firmware. One goal of UEFI is to ease the development by switching to the protected mode in an early stage and writing most of the code in C language. A portion of the Intel UEFI frame (named Tiano Core) is open source; however, the main function of the UEFI (to initialize the hardware devices) is still closed source. Coreboot (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. Similar to the UEFI-based BIOS, Coreboot also switches to protected mode in a very early stage and is written mostly in the C language. Our HyperCheck-II prototype implementation is based on Coreboot V4. We choose to use Coreboot rather than UEFI because Coreboot does all of the hardware initializations, whereas we would need to implement UEFI firmware from scratch, including obtaining all of the data sheets for our motherboard and other devices.

3 Threat Model and Assumptions

3.1 Attacker Capabilities

The adversary is able to exploit vulnerabilities in any software running in the machine after booting. The software includes the VMM and all of its privileged components. For instance, the attacker can compromise a guest domain and escape to the privileged domain. When using PCI passthrough on Intel VT-d chipsets that do not have interrupt remapping, Xen 4.1 and 4.0 allow the guest OS to gain host OS privileges by using DMA to generate malicious MSIs [11]. In Xen 3.0.3, pygrub [21] 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 [22]. In addition, the attacker can modify the hypervisor code or data using any known or zero-day attacks. For instance, the DMA attack [23] hijacks a device driver to perform unauthorized DMA accesses to the hypervisor's code and data.

HyperCheck aims to detect OS rootkits or hypervisor rootkits. One kind of rootkit only modifies the memory and/or registers and runs in the kernel level. For instance, the IDT-hook rootkit [24] modifies the interrupt descriptor table (IDT) in the memory and then gains control of the complete system. A stealthier version of the IDT-hook rootkit (we call it a copy-and-change attack) could keep the original IDT unchanged by copying it to a new location and altering it. Next, the attacker could change the IDTR register to point to the new location. Thus, a malicious interrupt handler would be executed when an interrupt occurs [25]. Our system could detect rootkits in an OS running on bare metal and rootkits in a native hypervisor.

3.2 General Assumptions

First of all, we assume BIOS is trusted. Since SMM code is loaded into SMRAM from the BIOS, we assume the SMRAM is properly set up by the BIOS while booting. To secure the BIOS code [26], [27], we can use a signed-BIOS mechanism to prevent any modification of the BIOS code, but this method requires that the BIOS updating process is securely implemented and trusted. An alternative way to secure the BIOS is to use Static Root of Trust Measurement (SRTM) to perform a trusted boot, and it requires that the Core Root of Trust Measurement (CRTM) is trusted and secure. The SMRAM is locked after booting into the OS. Once it is locked, we assume it cannot be subverted by the attacker (an assumption supported by current hardware), and we will discuss more SMM attacks in Section 7. Furthermore, we assume attackers do not have physical access to our system.

Currently, our system cannot protect against attacks that modify dynamic data, such as modification of dynamically generated function pointers and return-oriented programming attacks. In these attacks, the control flow is redirected to a memory location controlled by the attackers. HyperCheck can leverage existing solutions (e.g., Address Space Layout Randomization (ASLR) [28], [29]) to prevent or mitigate such attacks; however, it is not the focus of this paper. Section 7 provides a further discussion on the limitations of our system.

4 The HyperCheck Framework

HyperCheck is composed of three key components: the physical memory acquisition module, the analysis module, and the CPU register checking module. Both the physical memory acquisition module and CPU register checking module are on the target machine, and the analysis module is on the monitor machine. The memory acquisition module reads the memory contents of the protected machine and sends it to the analysis module, which then checks the memory contents for any malicious alterations. The CPU register checking module reads the CPU registers and validates their values. The overall architecture of HyperCheck is shown in Fig. 2. Before introducing the three key components, we first describe our design principles.

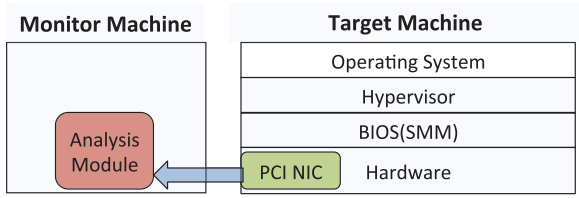

Fig. 2. The architecture of HyperCheck.

4.1 Design Principle

Our main design principle is that HyperCheck should not rely on any software running on the machine except the BIOS. Because SMM code resides in the BIOS, the BIOS and the monitor machine should be the only Trusted Computing Base (TCB) in our system. Therefore, we could use SMM to read the CPU registers and memory contents, and then use a PCI Ethernet card to send out this information to the monitor machine. Usually, Ethernet cards are PCI devices with bus master mode enabled and can read the physical memory through DMA, which does not need help from the CPU. SMM is an independent operating mode and can be made inaccessible from other CPU modes where hypervisors and privileged domains run.

Previous researchers only use PCI devices to read the physical memory [18]; however, CPU registers (e.g., CR3 and IDTR registers) are also important for rootkit detection because they define locations of active memory used by the hypervisor or the OS kernel. Without protecting these registers, an attacker can launch a copy-and-change attack by updating the registers to point to a new memory location controlled by the attacker. For instance, the attacker can prepare a malicious copy of the IDT table and change the IDTR register, pointing to the new IDT table [25]. Since PCI devices cannot read the CPU registers, it fails to detect this attack. By using SMM, HyperCheck can examine the CPU registers and report suspicious modifications.

4.2 HyperCheck Components

4.2.1 *Acquiring Physical Memory*

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

the SMRAM. The key is used as a random seed to selectively hash a small portion of the data to avoid data replay attacks.

Since paging is not enabled in SMM, HyperCheck uses the CR3 register to translate the virtual memory addresses used by the OS kernel to the physical memory addresses used by the SMM. Since the acquisition module relies on physical addresses to read the memory contents, HyperCheck needs to find the physical address of the protected hypervisor and privileged domain. One method is to use the `system.map` file to get the virtual addresses of symbols. HyperCheck uses this method to obtain the virtual addresses of monitoring symbols. However, we believe there are many other ways to obtain these addresses. For instance, the system call table address can be found by using the interrupt vector of the INT 0x80 system call [30]. From the symbol files, HyperCheck first reads the virtual addresses of the target memory and then utilizes CR3 register to find the physical addresses corresponding to the virtual ones. Another possible way to get the physical addresses without using page table translation is to scan the entire physical memory and use pattern matching to find all potential targets. However, this method is not efficient because hypervisors and OS kernels have a small memory footprint.

Furthermore, HyperCheck should be able to check the integrity of any software above the BIOS. Although we focus on the Xen hypervisor in this paper, HyperCheck also can be used to check KVM or other hypervisors. Some operating systems use Address Space Layout Randomization in kernel booting (e.g., Windows 7 [28]), which adds a fixed offset when setting up virtual address space. For example, Kernel Processor Control Region (KPCR) is located at a fixed virtual address 0xffdff000 in Windows XP and Windows 2000. In Windows 7, KPCR structure is no longer at a fixed address. However, researchers have demonstrated that the KPCR structure can be acquired by conditional searching of physical memory [31]. After obtaining the KPCR structure, we are able to bridge the semantic gap [32] in the physical memory and identify the targeting memory contents.

4.2.2 Analyzing Memory Content

In practice, there is a semantic gap between the physical memory addresses in SMM that we monitor and the virtual memory addresses used by the hypervisor or the OS. To verify the memory contents, the analysis module must be aware of the semantics of the memory layout, which depends on the specific hypervisor or the OS we monitor. The current analysis module depends on three properties of the kernel (OS or hypervisor) memory: linearity, stability, and perpetuity.

The linearity means that the kernel virtual memory is linearly mapped to physical memory and the offset is fixed. For instance, on x86 architecture, the virtual memory of Xen hypervisor is linearly mapped into the physical memory. In order to translate the virtual address to a physical address in Xen, we need only to subtract the virtual address from an offset. In addition, Domain 0 of Xen is also linearly mapped to the physical memory. The offset for Domain 0 is machine dependent but remains the same on a given machine. Moreover, other OS kernels, such as Windows [33], Linux [34] and OpenBSD [35], also have this property when they are directly running on bare metal.

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

The perpetuity property means 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 or match any content by only reading the physical memory. We would have to read the swapped files on the hard disk. For instance, Windows kernel code can be swapped to a disk. For this case, we have two solutions to read these swapped pages in the HyperCheck system. One method is to port a small disk driver in SMM to enable disk access. Then, we can use page table information to locate these pages on disk and send them to the monitor machine for integrity checking. The other solution is simply to wait for the swapped pages to swap back into memory. Since HyperCheck enters SMM periodically, we can check the page table information to see if the pages have been swapped in. After these pages are present in the memory, we send them out in SMM. Additionally, we can force these pages to be swapped back into memory by accessing them to generate page faults.

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

4.2.3 Reading and Verifying CPU Registers

Since the PCI NIC card cannot read the CPU registers, we must use another method to read them. Fortunately, SMM can read and verify the CPU registers. When the CPU switches to SMM, it saves the register context in the SMRAM. The default SMRAM size is 64 KB. The processor fetches the first instruction of the SMI handler at the address [SMBASE + 0x8000], and it stores the CPU states in the area from [SMBASE + 0xFE00] to [SMBASE + 0xFFFF] [36]. The default value of SMBASE is 0x30000. HyperCheck verifies the registers in SMM and reports the result 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 uses it for memory address translation of the hypervisor kernel code and data. The offsets between physical addresses and virtual ones should never change as we discussed in Section 4.2.2.

5 Implementation

We implement two prototypes for HyperCheck on two physical machines: HyperCheck-I uses an original closed-source BIOS, and HyperCheck-II uses an open-source BIOS called Coreboot [19]. We first develop HyperCheck-I for quick prototyping and debugging. After that, we implement HyperCheck-II as an improved version of our previous prototype in terms of security and scalability.

HyperCheck-I follows the HyperCheck framework shown in Fig. 2 to use two physical machines: one as the target machine and the other as the monitor machine. On the

target machine, we install Xen 3.1 natively and use Intel e1000 Ethernet card as the acquisition module. We modify the default SMM code in the original Dell BIOS on the target machine to transfer system states to the monitor machine. Since we use original BIOS with closed source code, we need to apply reverse engineering methods to change the default SMI handler code [37] on the target machine. However, HyperCheck-I comes with two drawbacks. First, it relies on an unlocked SMRAM to inject the customized SMI handler code while most of machines today have locked their SMRAM. The other drawback of the HyperCheck-I design is the high development complexity. Due to the time-consuming reverse engineering required and the usage of assembly language, it is difficult to add new functions into the BIOS. For instance, we have to use a kernel module to prepare the network transmit descriptors, instead of implementing all the functions in the BIOS. Therefore, we implement another HyperCheck prototype called HyperCheck-II using Coreboot, an open-source BIOS.

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

5.1 Memory Acquisition Module

HyperCheck uses a dedicated PCI network card to transfer the memory contents. In our prototype, we have two network interfaces on the target machine. We use an Intel e1000 network card to transfer the system states and the integrated network card for the normal traffic. When we implement the acquisition module, the main task is to port the e1000 network card driver into SMM to scan the memory and send it out to the monitor machine. Since HyperCheck-I does not have the source code of the BIOS, we use a similar method mentioned in [37] to modify the default SMM code in the BIOS. It writes the SMM code in 16-bit assembly code, uses a user-level program to open the SMRAM, and then copies the assembly code to the SMRAM. While HyperCheck-II uses the open source Coreboot as the BIOS, we have full control over the BIOS code. Thus, we can write C code to port the e1000 NIC driver into SMI handler of HyperCheck-II.

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

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

To prevent replay attacks, a secret key is transferred from the monitor machine to the target machine during the booting of the target machine. The key is used to create a random seed to selectively hash the data. If we hash the entire data stream, the performance impact may be high. To reduce the overhead, we use the secret key as a seed to generate one big, random number used for a one-time pad encryption and another set of serial random numbers. The serial random numbers are used as the indices of the positions of the memory. Then, the contents at these positions are XORed with the big, random number before starting NIC DMA. After the transmission is done, the memory contents received by the monitor machine 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 the data integrity, the NIC driver stays in the SMM until all of the packets have been written to the internal FIFO of the NIC. Then, it adds an extra 16 KB data to the end to flush the internal 16 KB FIFO buffer of the NIC. Thus, the attacker cannot use loop-back mode to get the secret key or peek into the internal NIC buffer through debugging registers of the NIC.

5.2 Analysis Module

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

time of the system, the contents of those control data and code may change.

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

5.3 CPU Register Checking Module

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

5.4 From HyperCheck-I to HyperCheck-II

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

Similar to HyperCheck-I, HyperCheck-II reserves a small portion of memory by adding the boot parameter `mem=2,000M` to the Xen hypervisor or Linux kernel. Since the total memory size is 2,048 MB, it saves 48 MB of memory to store the TX descriptor ring.

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

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

5.5 Evasion Attack Defense

Most polling-based integrity checking systems (e.g., [14], [15], [38]) are vulnerable to evasion attacks. If the polling time is detectable, an attacker can clean the attacking trace before the polling and reload the malicious code after. As discussed in our previous work [39], there are three types of evasion attacks for polling-based systems. The first type (Type-I) is to directly intercept the triggering events by detouring SMI invocation. The software-based SMI triggering writes to a specific port, and the attacker can locate the code (OUT %AL, %0XB2) in memory to detour the program execution before triggering SMI. The hardware-based SMI triggering method can be rerouted to normal interrupt and reissue another SMI triggering. The second type (Type-II) of evasion attack is to indirectly derive periodicity of the polling time by using an SMI detector. After knowing the polling time, an attacker can launch the evasion attack between two checks. The last type (Type-III) of evasion attack is to avoid random polling by running a small, malicious task after exiting SMM. This type of evasion attack assumes the attacker knows when the SMM exits.

HyperCheck-I may suffer from evasion attacks; however, HyperCheck-II can prevent all three types of evasion attacks. We use Message Signaled Interrupts (MSI) in a PCI network card to invoke SMI. The delivery mode of MSI register in PCI configuration space can be set as the SMI mode. However, if the hypervisor or OS has been compromised, the attacker can modify MSI register to trigger a normal interrupt controlled by the attacker and then reissue an SMI (Type-I evasion attack). To trigger the same SMI, the attacker needs to know the MAC address of the dedicated NIC card and use another machine to send an authenticated packet to reissue the same SMI. Since the secret key is saved in SMRAM, the attacker cannot send the authenticated packet without knowing the secret key. Furthermore, the attacker may use other methods to trigger SMI, but we can prevent it by checking the reason for the triggered SMI, which has also been implemented in HyperSentry [15].

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

Type-III evasion attack runs a small, malicious task after SMM exits to avoid the random polling. However, this type

TABLE 1
Symbols for Xen Hypervisor, Domain 0, Linux and Windows

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

of evasion attack needs to know when SMM exits. In HyperCheck-II, we include a random delay function in the SMI handler, so the SMI handler will take various amounts of time between two checkings. Thus, the attacker cannot accurately predict when SMM exits.

6 Evaluation

We evaluate the HyperCheck system on two different testbeds for HyperCheck-I and HyperCheck-II. The monitor machine is the same for both HyperCheck-I and HyperCheck-II. It is a Dell Precision 690 with 8 GB RAM and one 3.0 GHz Intel Xeon CPU with two cores. The host operating system is 64-bit CentOS 5.3. The target machine in HyperCheck-I is implemented on a Dell Optiplex GX 260 with one 2.0 GHz Intel Pentium 4 CPU and 512 MB memory. Xen 3.1 and Linux 2.6.18 is installed on the physical machine, and the Domain 0 is CentOS 5.4. The Dell BIOS version A09 is closed source. The target machine in HyperCheck-II uses an ASUS M2V-MX_SE motherboard with 2.2 GHz AMD Sempron LE-1250 CPU and 2 GB memory. We install CentOS 5.5 as the operating system. We replace the original BIOS with the open source Coreboot V4.

6.1 Code Size

In the HyperCheck-I implementation, we inject about 100 lines of assembly code into the original BIOS. Since HyperCheck-II uses the open source Coreboot, we add only 200 lines of C code into Coreboot source tree. The code base of Coreboot V4 is 232,315 lines of code, and its payload Seabios has 21,576 lines of code, which are measured using sloccount [40].

6.2 Verifying the Static Property

We verify an important system assumption that the control data and respective code are statically mapped into the physical memory. We use a monitoring module designed to detect legitimate control data and code modifications throughout the experiments. It enables us to test our approach against data changes and self-modifying code in the Xen hypervisor and Domain 0. We also test the static properties of Linux 2.6 and Windows XP 32-bit kernels. In all these tests, the hypervisor and the OSes are booted into a minimal state. The symbols used in the experiments are shown in Table 1. During system booting time, we find that the control data and the code may change. For example, the physical memory of IDT is all 0s when the system first boots up, but after several seconds, it becomes non-zero and static. The reason is that the IDT table is initialized later in the booting process.

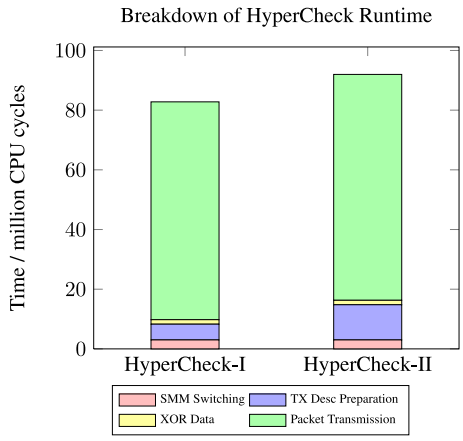

Fig. 3. Breakdown of HyperCheck runtime.

6.3 Integrity Attacks Detection

To verify HyperCheck's capability of detecting attacks against the hypervisor, we implement DMA attacks [23] on the Xen hypervisor. Firstly, we port the HDD DMA attacks to modify the Xen hypervisor and Domain 0. In this experiment, there are four attacks against the Xen hypervisor (modifying IDT table, hypercall table, exception table and Xen code) and two attacks against Domain 0 (modifying system call table and Domain 0 code). In another experiment, we modify the pcnet network card to perform the DMA attack from the hardware directly. The modified pcnet NIC is used to attack Linux and Windows operating systems. This experiment includes three attacks against Linux 2.6.18 kernel (modifying IDT table, system call table and kernel code) and two attacks to Windows XP SP2 kernel (modifying IDT table and system call table). In our experiments, HyperCheck-I and HyperCheck-II correctly detect all of these attacks and report the memory content changes on the target machine.

6.4 Breakdown of HyperCheck

To quantify how much time is required to execute each step in the system, we breakdown the HyperCheck into four logical operations: 1) SMM context switch; 2) TX descriptors preparation; 3) XOR data; and 4) packet transmission. To measure the time for each operation, we use `rdtsc` instruction to print out the TSC counter value. This experiment is conducted on both HyperCheck-I and HyperCheck-II. The sending data size is about 2.8 MB including Xen code and Domain 0 code; we also add an extra 16 KB of data at the end to flush the NIC internal buffer. In addition, we use 7 KB as the packet size because it introduces the lowest network delay; and more details of network delay can be found in Section 6.5.

Fig. 3 shows the observed times of each breakdown operation in HyperCheck-I and HyperCheck-II. We can see that

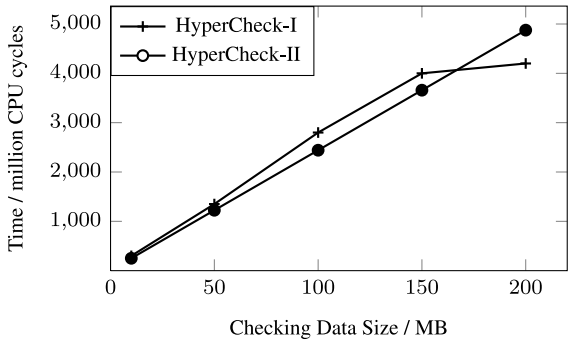

Fig. 4. Network time delay for variable data size in HyperCheck.

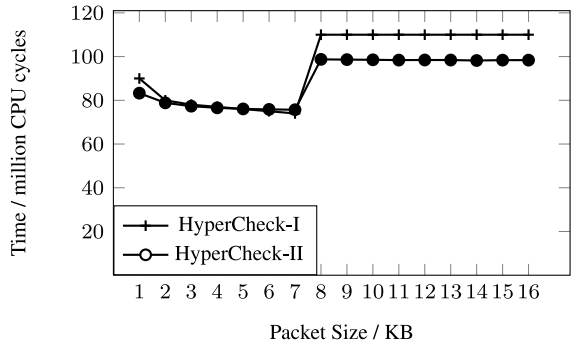

Fig. 5. Network time delay for variable packet size in HyperCheck.

packet transmission time is the majority. Additionally, HyperCheck-II spends more CPU cycles for preparing TX descriptors because the same amount of code running in SMM takes more time than in normal protected mode. This is mainly due to the fact that 1) SMM operates in 32-bit mode while a normal OS runs in 64-bit protected mode, and 2) SMM physical memory needs to be uncacheable to avoid cache poisoning attacks [41], [42].

The sizes of different hypervisors and OSes may vary (e.g., Linux running with KVM). HyperCheck is scalable to measure other hypervisors and OSes, but it should expect more performance overhead when measuring larger code base systems. We measure the time delay for sending different sizes of data in both HyperCheck-I and -II where the packet size is 7 KB. The results are shown in Fig. 4. We can see that the time increases almost linearly along with the size of memory in both prototypes.

6.5 Network Packet Size Analysis

To optimize the network time delay for our system, we measure the packet transmission time by varying the packet size for sending a fixed amount of memory. The memory size is about 2.8 MB including Xen code and Domain 0 code. We range the packet size from 1 to 16 KB on both HyperCheck-I and HyperCheck-II. Fig. 5 shows the results. When the packet size is less than 7 KB, the transmission time is about constant. However, when the packet size increases to 8 KB, the overhead increases dramatically and remains constant after that. The reason is that the internal NIC transfer FIFO buffer size is 16 KB for our network card. Therefore, when the packet size becomes 8 KB or larger, the buffer cannot hold two packets at the same time, which introduces the delay.

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

6.6 System Overhead

We also measure the overall system overhead incurred by different sampling intervals of HyperCheck-II. In this experiment, we run the UnixBench [43] suite without our system in place. Next, we run the benchmark with HyperCheck-II enabled at several different time intervals ranging from 0.0625 to 5 seconds using Global Standby Timer (GP0) on the southbridge to periodically trigger an SMI. We then calculate the overhead as a ratio with and without the system in place. In this experiment, we transfer Xen and domain 0 code (2,881 KB) and use 7 KB as the packet size. Fig. 6 shows the result of the overhead. In general, HyperCheck-II introduces a low overhead. It causes 2 percent overhead when trigging SMI every 5 seconds and 11 percent overhead with a 1-second sampling interval.

6.7 Comparison with Other Methods

HyperGuard [38] suggests using SMM to read the memory and hash it on the target machine. Flicker [44] is a TPM-based approach that can be used to monitor the integrity of the kernels. We compare our method with them, and Table 3 shows the results. We can see that the overhead of HyperCheck is one order of magnitude lower than HyperGuard and the TPM-based method. In HyperGuard, it must hash the entire data to check its

TABLE 2
Time Measurements for Variable Packet Sizes in HyperCheck-II

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

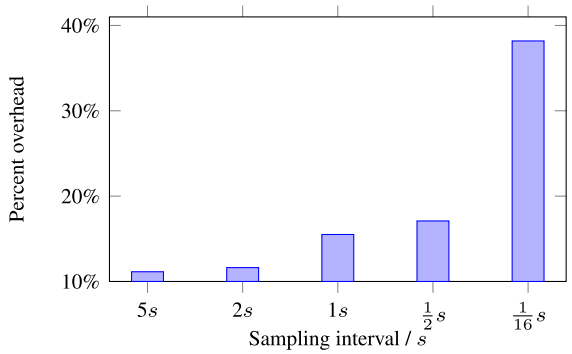

Fig. 6. Overhead introduced in HyperCheck-II with different sampling intervals.

integrity, while HyperCheck only hashes a random portion of the data and then sends the entire data out using an Ethernet card. For the TPM-based method, the most expensive operation is the TPM quote, which takes 972 ms. Although HyperCheck needs TPM in SRTM process to secure the BIOS, it does not require TPM at runtime. Once the SMM is securely set up, HyperCheck leverages SMM to perform its integrity checking, while Flicker requires TPM operation for each check. Additionally, an overall comparison between HyperCheck and other methods is shown in Table 4. In summary, HyperCheck can monitor both memory and registers with a lower overhead.

7 Security Analysis and Limitations

HyperCheck aims to detect modifications to the static code and control data of the hypervisors or OS kernels. It uses the SMM and a dedicated NIC card to read the physical memory via DMA and then validates the results on a monitor machine.

7.1 Address Binding Validation

Some previous works rely on the symbol table to find the virtual address of the kernel code and data. Nonetheless, there is no binding between the virtual addresses in the symbol table and the actual physical addresses of these symbols [3]. However, HyperCheck validates this binding by checking CPU registers. In order to explain how HyperCheck overcomes this problem, we use the IDT table as an example. One potential attack is to modify the IDTR register and point to another address (copy-and-change attack). The malware can then modify the new IDT table and keep the old one untouched. Another potential attack 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.

TABLE 3
Comparison on Time Overhead

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

TABLE 4
Comparison on Capability and Overhead

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

HyperCheck can detect these attacks by checking CPU registers in SMM. The value of the IDTR register should never change after booting; otherwise, SMM sends a warning via the Ethernet card to the monitor machine. Thus, HyperCheck can detect copy-and-change attacks by checking IDTR value. Furthermore, we use the CR3 register to detect page table changing attacks. Using the CR3 register, HyperCheck can find the actual physical address for a given virtual address. The offset between the virtual address and the physical address should always be static. If the offset changes, it means the CR3 or page tables have been altered, and HyperCheck sends a warning message to the monitor machine. Moreover, PCI devices, including the NIC card itself, can be cheated and redirected to get a different mapping of the physical memory [45]. We can prevent these attacks by storing a correct PCI configuration space in SMRAM and then comparing it to current PCI space while running in SMM.

7.2 Network Card Attacks

Though the network driver is running in the protected SMM, the firmware of NIC could be malicious, too. For instance, an attacker may flash the NIC firmware with a malicious one after compromising the OS. After that, the malicious NIC firmware can modify the packets to avoid detection. In addition, Duflot et al. [46] demonstrated that attackers can take full control of the OS by exploiting a vulnerability in NIC firmware. In order to defeat these attacks, we can store a hash of original NIC firmware in the SMRAM and verify its integrity in the SMI handler before using the NIC. It would require us to update the hash value for new benign NIC firmware patches.

We do not use IOMMU in HyperCheck because our testbeds do not support it. However, the design of HyperCheck is compatible with IOMMU. We can configure IOMMU to allow the dedicated NIC to access the whole physical memory in Coreboot during system booting. To prevent malicious attacks that misuse IOMMU to manipulate or disable NIC, we can check the current state of IOMMU settings in SMM, similar to checking PCI configuration space.

Moreover, to prevent replay attacks, we use a key to randomly hash a portion of the data and then send them out to the analysis module. Since the secret key is locked in the SMRAM, the attacker cannot get to it to generate the same hash. Attackers can still disable the Ethernet card or the SMM code in a denial-of-service attack; however, the monitor machine can easily detect such an attack.

An attacker may try to launch a fake reboot attack to get the secret key from the monitor machine. It can mimic the SMM NIC driver and send a request for a new key. We have two solutions to prevent this attack. First, we could use dynamic root of trust for measurement (DRTM)-based remote attestation to verify the running state of the target

machine [27]. We only need to verify whether the OS has been started or not. If it is already started, the monitor machine should refuse to send the key. If the target machine does support the DRTM, the second method is to send another reliable reboot signal to the target machine to make sure the SMM code is running whenever the target machine asks for the key.

7.3 SMM and BIOS

HyperCheck relies on SMM to check system integrity, and it assumes SMM code is trusted. Fortunately, to the best of our knowledge, there is no generic way to bypass SMM protection with locked SMRAM. Once the D_LCK bit is set, the SMM code cannot be modified until system reboot. SMM rootkits proposed by Embleton et al. [47] can have ring -2 privilege and serve as a keystroke logger, but it only works with unlocked SMRAM. However, all of the post-2006 machines have locked the D_LCK bit during system booting, and it is a major reason for us to develop HyperCheck-II, which locks SMRAM after booting. In addition, Wojtczuk and Rutkowska [41] demonstrate how to modify SMM memory via Intel CPU cache poisoning due to a hardware design flaw. By manipulating CPU MTRR registers, the attacker can make the CPU execute the SMM code from cache instead of DRAM. Thus, the attacker can bypass the SMRAM protection and successfully run the code in cache with SMM privilege. The same vulnerability was also discovered by Duflot et al. [42]. Fortunately, Intel has cooperated with BIOS vendors and fixed this architectural vulnerability.

Though SMM is not designed for security purposes, we leverage SMM to assist the integrity monitoring. We want to emphasize that SMM is the mechanism that essentially provides an isolated computing fabric and the hardware support for meeting HyperCheck's needs. Our prototype leverages the isolation principles currently implemented by SMM, but this does not mean that the HyperCheck architecture must use SMM. In terms of future hardware support, we are essentially demanding a CPU that provides a dedicated mode for security purposes.

Since SMM code is loaded from the BIOS, we need to protect the BIOS from malicious modification. Signed BIOS update enforcement can protect against malicious writes to the flash chip. Intel chipsets provide hardware registers to support this mechanism (Protected Range Registers and BIOS_CNTL register). Furthermore, this method requires that the BIOS update routine is securely implemented. However, researchers [48] have demonstrated that this access control can be bypassed by exploiting a vulnerability of the BIOS updating process. To further protect the BIOS code, we can use Static Root of Trust for Measurement to check the integrity of the BIOS when system boots up [27]. As suggested by Butterworth et al. [48], this method requires that Core Root of Trust for Measurement (CRTM) is trusted so that it can perform the self-measurement of the BIOS.

7.4 Multi-Core Platforms

Although HyperCheck is implemented on a single core platform, SMM is able to deal with multi-core processors as well. Since each core has its own MSR registers to define SMRAM, each core can have its own SMI handler. When an SMI is generated, all of the cores on the platform are switched to the SMM. However, multi-core mode introduces a security concern in the HyperCheck system. When one core transmits the network packets, the other ones may be able to manipulate the memory or NIC card to hide attack traces. One solution is to let other cores remain in SMM until the core executing HyperCheck code finishes. Another more efficient way is to let one core execute HyperCheck code while other cores resume their normal operations. The latter approach needs to carefully handle inter-core communication and prevent memory modification between cores. SICE [49] has demonstrate this method on AMD processors.

When the memory mapping of the hypervisor does not hold the three properties (linearity, stability, perpetuity), the current version of HyperCheck cannot work correctly. Moreover, it is still a challenge to check the integrity of the dynamic components of the Hypervisor or OS kernel. We will address these problems in the future.

8 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 [18], [50] or for forensic purposes [51], [52] in the past. Copilot [18] employs a special PCI device to poll the physical memory of the host and send it to an administrator, station periodically. 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 register states. 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 compromised. Similar to HyperCheck, SMMDumper [52] leverages SMM and the PCI network card to perform acquisition of volatile memory of the running system. The purpose of SMMDumper is forensic analysis, and it needs to transmit the whole physical memory of the running system, while HyperCheck only transfers the critical memory sections for integrity checking.

Another closely related work is HyperGuard [38], which suggests using SMM in the x86 CPU to monitor the integrity of the hypervisors. HyperCheck has a similar goal, but it can outsource the state snapshot by using a network card. This results in a drastic performance improvement of the system, reducing the system busy time from seconds to milliseconds. Another difference is that the monitor machine in the HyperCheck system can be used to detect the DoS attacks against the SMM code.

HyperSentry [15] uses an out-of-band channel (Intelligent Platform Management Interface (IPMI) commonly supported on server platforms) to trigger SMI, and adopts SMM to protect its base code on critical data. HyperCheck does not need IPMI support and could work on both desktops and servers. Another difference in HyperSentry is the location of the analysis module, which is on the target machine, while HyperCheck performs integrity analysis on a separate machine, which can reduce the workload of the target machine.

Vigilare [53] is a system bus traffic monitor on a system-on-a-chip (SoC) platform for checking kernel

integrity. It can effectively prevent the evasion attack in polling systems by using event-driven monitoring mechanisms. However, Vigilare only works on the SoC platform and requires extra hardware support (i.e., to duplicate system bus traffic to the verifier) in desktops and servers, while HyperCheck leverages an existing hardware feature (SMM) in the x86 CPUs.

HyperWall [54] architecture achieves hypervisor-secure virtualization by using hardware to isolate a VM's memory, protecting guest VMs from untrusted hypervisors. Although HyperCheck and HyperWall are both hardware-assisted systems, they have different threat models. HyperCheck provides integrity checking of both the hypervisor and the operating system, while HyperWall only protects the guest VMs from a malicious hypervisor. In addition, HyperWall requires new hardware components in the microprocessor, while HyperCheck requires no hardware modifications.

Flicker [44] uses a TPM to ensure a minimum Trusted Code Base, which can be used to detect modifications to the kernels. Flicker requires hardware features such as Dynamic Root of Trust Measurement and late launch, which have been widely supported in new computers. HyperCheck uses the SRTM to secure the booting process, but it does not need any TPM operation at runtime. To reduce the overhead of Flicker, TrustVisor [17] has a small footprint hypervisor to perform cryptography operations. However, all of the legacy applications should be recompiled and ported to work on TrustVisor. TrustVisor also requires the DRTM support. SecureSwitch [55] is a BIOS-assisted system for secure instantiation and management of trusted execution environments. It has the same TCB (BIOS and hardware devices) with HyperCheck, but it is designed for different purposes.

Another branch of research focuses on improving the security of the hypervisor by adding hooks [13], [56], [57] and enforcing security policies in virtual machines [54], [58]. 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 mechanisms.

Furthermore, there is a plethora of research towards protecting the Linux kernel [6], [59]. Baliga et al. [50] use a PCI device to acquire the memory and automatically derive the kernel invariance. Currently, we discover the kernel invariance manually, but we could integrate their techniques into our system smoothly. Litty et al. [3] develop a technique to discover the address of key data structures that are instantiated during runtime by relying on processor hardware and executable file specifications. It assumes that the underlying hypervisor can be trusted. Instead, HyperCheck first obtains the virtual addresses of those symbols through the symbol file, and then calculates the physical addresses through CPU registers. Therefore, HyperCheck can get the correct view of the system memory even if the underlying OS or hypervisor is compromised and page tables are altered. Some other researchers [1], [2], [60] also depend on the integrity of the hypervisor to protect the kernel. Our work is complementary to those mechanisms and can be employed as a meta-protection mechanism to guard the integrity of OS-level defenses. AppCheck [61] is a follow-up work on HyperCheck to check the integrity of application code.

9 Conclusions

In this paper, we introduce HyperCheck, a hardware-assisted tampering detection framework, that aims to protect the code integrity of software running on commodity hardware. HyperCheck relies on the CPU System Management Mode to securely generate and transmit the state of the protected machine to an external machine. HyperCheck does not rely on any software running on the target machine beyond a trusted BIOS. Moreover, HyperCheck is robust against evasion attacks and DoS attacks.

To demonstrate the feasibility of our approach, we implemented two prototypes on two testbeds. HyperCheck-I is implemented on real hardware with an original closed-source BIOS, and HyperCheck-II is based on a physical machine with open-source Coreboot. Our experimental results indicate that we can successfully identify alterations of the control data and code on protected systems. Overall, HyperCheck operation is lightweight, and it can complete one round of integrity checking in less than 90 million CPU cycles.

Acknowledgments

The authors would like to thank all of the anonymous reviewers for their valuable comments and suggestions. This work was partly supported by the US National Science Foundation under Grants CT 0915291 and CNS 1205453 and United States Air Force Research Laboratory (AFRL) through Contract FA8650-10-C-7024. Opinions, findings, conclusions and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US Government, National science Foundation (NSF), or the Air Force.

References

[1] Y. Fu and Z. Lin, "Space Traveling across VM: Automatically Bridging the Semantic gap in Virtual Machine Introspection via Online Kernel Data Redirection," *Proc. 33rd IEEE Symp. Security and Privacy*, 2012.

[2] T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection," *Proc. 32nd IEEE Symp. Security and Privacy*, 2011.

[3] L. Litty, H.A. Lagar-Cavilla, and D. Lie, "Hypervisor Support for Identifying Covertly Executing Binaries," *Proc. 17th Conf. Security Symposium (SS '08)*, pp. 243-258, 2008.

[4] B. Payne, M. de Carbone, and W. Lee, "Secure and Flexible Monitoring of Virtual Machines," *Proc. 23rd Ann. Computer Security Applications Conf. (ACSAC)*, pp. 385-397, Dec. 2007.

[5] T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection based Architecture for Intrusion Detection," *Proc. Network and Distributed Systems Security Symp.*, 2003.

[6] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes," *Proc. 21st ACM SIGOPS Symp. Operating Systems Principles (SOSP '07)*, 2007.

[7] M. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-VM Monitoring using Hardware Virtualization," *Proc. 16th ACM Conf. Computer and Comm. Security (CCS '09)*, 2009.

[8] National Institute of Standards, NIST, National Vulnerability Database, http://nvd.nist.gov, 2014.

[9] N. Elhage, "Virtunoid: Breaking Out of KVM," https://nelhage.com/talks/kvm-defcon-2011.pdf, 2011.

[10] K. Kortchinsky, "CLOUDBURST: A VMware Guest to Host Escape Story," *Blackhat USA*, 2009.

[11] R. Wojtczuk, J. Rutkowska, "Following the White Rabbit: Software Attacks against IntelÂ® VT-d," http://invisiblethingslab.com/itl/Resources.html, 2011.

[12] S. King and P. Chen, "Subvirt: Implementing Malware with Virtual Machines," *Proc. IEEE Symp. Security and Privacy*, pp. 314-327, May 2006.
[13] Z. Wang, C. Wu, M. Grace, and X. Jiang, "Isolating Commodity Hosted Hypervisors with Hyperlock," *Proc. Seventh ACM SIGOPS/EuroSys European Conf. Computer Systems (EuroSys '12)*, 2012.
[14] J. Wang, A. Stavrou, and A. Ghosh, "HyperCheck: A Hardware-Assisted Integrity Monitor," *Proc. 13th Int'l Symp. Recent Advances in Intrusion Detection*, 2010.
[15] A.M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N.C. Skalsky, "HyperSentry: Enabling Stealthy in-Context Measurement of Hypervisor Integrity," *Proc. 17th ACM Conf. Computer and Comm. Security*, 2010.
[16] D. Murray, G. Milos, and S. Hand, "Improving Xen Security through Disaggregation," *Proc. Fourth ACM SIGPLAN/SIGOPS Int'l Conf. Virtual Execution Environments*, pp. 151-160, 2008.
[17] J.M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "TrustVisor: Efficient TCB Reduction and Attestation," *Proc. 31st IEEE Symp. Security and Privacy*, 2010.
[18] N.L. Petroni Jr., T. Fraser, J. Molina, and W.A. Arbaugh, "Copilot - A Coprocessor-Based Kernel Runtime Integrity Monitor," *Proc. 13th Conf. USENIX Security Symp. (SSYM '04)*, p. 13, 2004.
[19] Coreboot, "Open Source BIOS," http://www.coreboot.org/, 2014.
[20] UEFI, "Unified Extensible Firmware Interface," http://www.uefi.org/home/, 2014.
[21] D. Chisnall, *The Definitive Guide to the Xen Hypervisor*. Prentice Hall, 2007.
[22] MITRE, "Cve-2007-4993," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-4993, 2014.
[23] R. Wojtczuk, "Subverting the Xen Hypervisor," http://invisiblethingslab.com/resources/misc08/xenfb-adventures-10.pdf, 2008.
[24] K. Adamyse, "Handling Interrupt Descriptor Table for Fun and Profit. Phrack 59," 2002.
[25] B. ProchÃazka, T. Vojnar, and M. Drahansk, "Hijacking the Linux Kernel," http://drops.dagstuhl.de/opus/volltexte/2011/3063/pdf/7.pdf, http://drops.dagstuhl.de/opus/volltexte/2011/3063/pdf/7.pdf, 2011.
[26] A. Regenscheid, K. Scarfone, "BIOS Integrity Measurement Guidelines (Draft), NIST-800-155," http://csrc.nist.gov/publications/drafts/800-155/draft-SP800-155_Dec2011.pdf, Dec. 2011.
[27] "TCG PC Client Specific Implementation Specification for Conventional BIOS," http://www.trustedcomputinggroup.org/files/resource_files/CB0B2BFA-1A4B-B294-D0C3B9075B5AFF17/TCG_PCClientImplementation_1-21_1_00.pdf, Feb. 2012.
[28] M. Howard, M. Miller, J. Lambert, and M. Thomlinson, "Windows ISV Software Security Defenses," Dec. 2010.
[29] PaX Team, "http://pax.grsecurity.net/," http://pax.grsecurity.net/, 2014.
[30] D. SD, "Linux on-the-Fly Kernel Patching without LKM," *Phrack Magazine*, 2001.
[31] S. Zhang, L. Wang, R. Zhang, and Q. Guo, "Exploratory Study on Memory Analysis of Windows 7 Operating System," *Proc. Third Int'l Conf. Advanced Computer Theory and Eng. (ICACTE)*, vol. 6, pp. V6–373-V6–377, 2010.
[32] F. Zhang, K. Leach, K. Sun, and A. Stavrou, "SPECTRE: A Dependable Intropsection Framework via System Management Mode," *Proc. the 43rd Ann. IEEE/IFIP Int'l Conf. Dependable Systems and Networks (DSN '13)*, 2013.
[33] S. Schreiber, *Undocumented Windows 2000 Secrets: A Programmer's Cookbook*. Addison-Wesley, 2001.
[34] D. Bovet and M. Cesati, *Understanding the Linux kernel*. third ed., O'Reilly Media, 2005.
[35] L. Duflot, D. Etiemble, and O. Grumelard, "Using CPU System Management Mode to Circumvent Operating System Security Functions," *Proc. Seventh CanSecWest Conf.*, 2004.
[36] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual," http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html, 2014.
[37] BSDaemon, coideloko, D0nAnd0n, "System Management Mode Hack: Using SMM for 'Other Purposes'," Phrack Magazine, 2008.
[38] J. Rutkowska and R. Wojtczuk, "Preventing and Detecting Xen Hypervisor Subversions," *Blackhat Briefings USA*, 2008.
[39] J. Wang, K. Sun, and A. Stavrou, "A Dependability Analysis of Hardware-Assisted Polling Integrity Checking Systems," *Proc. 42nd Ann. IEEE/IFIP Int'l Conf. Dependable Systems and Networks (DSN '12)*, 2012.
[40] sloccount, "Sloccount," http://www.dwheeler.com/sloccount/, 2014.
[41] R. Wojtczuk, and J. Rutkowska, "Attacking SMM Memory via Intel CPU Cache Poisoning," http://invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf, 2009.
[42] L. Duflot, O. Levillain, B. Morin, and O. Grumelard, "Getting into the SMRAM: SMM Reloaded," *Proc. 10th CanSecWest Conf.*, 2009.
[43] Unixbench, http://code.google.com/p/byte-unixbench/, 2014.
[44] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki, "Flicker: An Execution Infrastructure for TCB Minimization," *Proc. Third ACM SIGOPS/EuroSys European Conf. Computer Systems*, 2008.
[45] J. Rutkowska, "Beyond the CPU: Defeating Hardware Based RAM Acquisition," *Proc. BlackHat DC Conf.*, 2007.
[46] L. Duflot, Y.-A. Perez, and B. Morin, "What If You Can't Trust Your Network Card?" *Proc. 14th Int'l Conf. Recent Advances in Intrusion Detection (RIAD '11)*, pp. 378-397, 2011.
[47] S. Embleton, S. Sparks, and C. Zou, "SMM Rootkits: A New Breed of OS Independent Malware," *Proc. Fourth Int'l Conf. Security and Privacy in Comm. Networks*, 2008.
[48] J. Butterworth, C. Kallenberg, and X. Kovah, "Bios Chronomancy: Fixing the Core root of Trust for Measurement," *Proc. 20th ACM Conf. Computer and Comm. Security (CCS '13)*, 2013.
[49] A.M. Azab, P. Ning, and X. Zhang, "SICE: A Hardware-Level Strongly Isolated Computing Environment for x86 Multi-Core Platforms," *Proc. 18th ACM Conf. Computer and Comm. Security*, 2011.
[50] A. Baliga, V. Ganapathy, and L. Iftode, "Automatic Inference and Enforcement of Kernel Data Structure Invariants," *Proc. Ann. Computer Security Applications Conf. (ACSAC '08)*, pp. 77-86, 2008.
[51] J. Wang, F. Zhang, K. Sun, and A. Stavrou, "Firmware-Assisted Memory Acquisition and Analysis Tools for Digital Forensic," *Proc. IEEE Sixth Int'l Workshop Systematic Approaches to Digital Forensic Eng. (SADFE '11)*, 2011.
[52] A. Reina, A. Fattori, A. Pagani, L. Cavallaro, and D. Bruschi, "When Hardware Meets Software: A Bulletproof Solution to Forensic Memory Acquisition," *Proc. Ann. Computer Security Applications Conf. (ACSAC '12)*, 2012.
[53] H. Moon, H. Lee, J. Lee, L. Kim, P.Y., and K.B., "Vigilare: Toward Snoop-Based Kernel Integrity Monitor," *Proc. 19th ACM Conf. Computer and Comm. Security (CCS '12)*, 2012.
[54] J. Szefer and R.B. Lee, "Architectural Support for Hypervisor-Secure Virtualization," *Proc. 17th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*, 2012.
[55] K. Sun, J. Wang, F. Zhang, and A. Stavrou, "SecureSwitch: BIOS-Assisted Isolation and Switch between Trusted and Untrusted Commodity OSes," *Proc. 19th Ann. Network & Distributed System Security Symp. (NDSS '12)*, 2012.
[56] G. Coker, "Xen Security Modules (XSM)," *Proc. Xen Summit*, 2006.
[57] Z. Wang and X. Jiang, "HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity," *Proc. IEEE Symp. Security and Privacy (SP '10)*, pp. 380-395, 2010.
[58] 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.
[59] X. Jiang, X. Wang, and D. Xu, "Stealthy Malware Detection Through VMM-Based Out-of-the-Box Semantic View Reconstruction," *Proc. 14th ACM Conf. Computer and Comm. Security*, 2007.
[60] R. Riley, X. Jiang, and D. Xu, "Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing," *Proc. 11th Int'l Symp. Recent Advances in Intrusion Detection*, 2008.
[61] J. Wang, K. Sun, and A. Stavrou, "Hardware-Assisted Application Integrity Monitor," *Proc. IEEE Hawaii Int'l Conf. System Sciences (HICSS45)*, 2012.

Fengwei Zhang received a master's degree in computer science from Columbia University in 2010. He is currently working toward a PhD at the Department of Computer Science of George Mason University. His research interests include system security, system integrity checking and malware debugging.

Jiang Wang received his PhD in information technology from George Mason University in 2011. He is a member of the technical staff in Riverbed Technology. His research interests include operating system security and browser security.

Kun Sun received his PhD in computer science from North Carolina State University in 2006. He is a research professor at the Center for Secure Information Systems at George Mason University. His research interests include system security, cloud security, moving targeting defense, and security in MANET and wireless sensor networks.

Angelos Stavrou received his PhD in computer science from Columbia University in 2007. He is an associate professor at the Department of Computer Science at George Mason University. His research interests include large systems security & survivability, intrusion detection systems, privacy & anonymity, and security for MANETs and mobile devices.