Department of Computer Science
George Mason University
Technical Report Series

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

HyperCheck: A Hardware-Assisted Integrity Monitor

Jiang Wang, Angelos Stavrou and Anup Ghosh
{jwanga, astavrou, aghosh1}@gmu.edu

Technical Report GMU-CS-TR-2010-5

Abstract

Over the past few years, virtualization has been employed to environments ranging from densely populated cloud computing clusters to home desktop computers. Security researchers embraced virtual machine monitors (VMMs) as a new mechanism to guarantee deep isolation of untrusted software components. Unfortunately, their widespread adoption promoted VMMs as a prime target for attackers. In this paper, we present HyperCheck, a hardware-assisted tampering detection framework designed to protect the integrity of VMMs and, for some classes of attacks, the underlying operating system (OS). HyperCheck leverages the CPU System Managed Mode (SMM), present in x86 systems, to securely generate and transmit the full state of the protected machine to an external server. Using HyperCheck, we were able to ferret-out rootkits that targeted the integrity of both the Xen hypervisor and traditional OSes. Moreover, HyperCheck is robust against attacks that aim to disable or block its operation. Our experimental results show that Hypercheck can produce and communicate a scan of the state of the protected software in less than 40ms.

1 Introduction

Hypervisors[1] have become the de facto standard in server consolidation because they decrease the energy footprint and cost of management of modern computing clusters. In addition, hypervisors are increasingly used as components to enforce system security and resilience [23, 27, 18, 37, 21, 35, 30]. This widespread adoption of virtualization has attracted the attention of the attackers towards VMM vulnerabilities. Indeed, recently, there has been a surge in the reported vulnerabilities for commercial and open source hypervisors [4]. Moreover, the number and nature [39, 10] of attacks against the hypervisors are poised to grow.

This increasing attack trend has spurred research towards reducing the hypervisor Trusted Code Base (TCB) of current commercial hypervisors [26]. Others developped new specialized prototype hypervisors [35, 25]. However, having a small code base can only limit the code exposure and thus the attack surface of the hypervisor – it cannot provide strong guarantees about the code integrity of all the hypervisor components.

To address these limitations and to complement the existing protection mechanisms, we designed a hardware-assisted tampering detection framework called HyperCheck. HyperCheck is designed to protect the integrity of VMMs and, for some classes of attacks, the underlying operating system (OS). To achieve that, HyperCheck harnesses the CPU System Managed Mode (SMM) which is present in all x86 commodity systems to create a snapshot view of the current state of the CPU and memory registers of the protected machine. This information is securely and verifiably transmitted using a network card to a remote analysis server. Using that information, the analysis server can identify any tampering by comparing the newly generated view with the one recorded when the machine was initialized. If the two views do not match, a human operator is notified. As shown in Figure 1, HyperCheck works at the BIOS level and can protect the software above it. Our assumptions are that the attacker does not have physical access to the machine and that the SMM BIOS is locked and thus cannot be altered during run. We do not explicitly require trusted boot to initialize HyperCheck [24, 25]. However, having a machine equipped with trusted boot can prevent attacks against HyperCheck that simulate a hardware re-

[1] Also called Virtual Machine Monitors VMMs

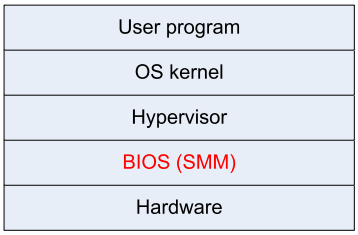

Figure 1: HyperCheck can offer protection to services running above BIOS

set [2]

Unlike previous work [29] that use specialized PCI hardware, we are able to acquire a complete view of the target machine's state including the entire memory and CPU registers. In addition, our approach is able to thwart attacks aimed at disabling, blocking, or even taking over PCI devices. To evaluate the validity and performance of our approach, we implemented two prototypes for HyperCheck. HyperCheck-I uses QEMU [7] – a fully system emulator – to emulate the PCI NIC, while HyperCheck-II is based on an Intel e1000 physical NIC. Using our prototypes, we were able to ferret-out rootkits aimed at Xen [14] hypervisor, Xen Domain 0, Linux, and Windows. Our experimental results indicate that HyperCheck does not cause prohibitive performance overhead requiring only a few milliseconds to completely transmit each snapshot.

In summary, we make the following contributions:

1. Designed a novel hardware-assisted tampering detection framework that creates a complete snapshot of the state of the system with commercial hardware and no modification to the installed software.

2. Implemented two prototypes: one based on QEMU and the other based on the real hardware. The latter has overhead in the order of few milliseconds. Using our prototype, we demonstrate that we can successfully detect rootkits and code integrity attacks against the Xen VMM, Xen Domain 0, Linux, and Windows.

2 Related work

Protecting software from integrity attacks using hardware-assisted techniques is not new: researchers used a special-purpose PCI device to acquire the physical memory either for rootkit detection [29, 6] or for forensic purpose [12] in the past. The closest system to our work is Copilot [29]. Copilot employed a special PCI device to poll the physical memory of the host and send it to an admin 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 state including its registers. Such view is important to prevent copy-and-change attacks that can mislead the PCI card to scan the wrong regions of memory and report erroneously that the system is not affected.

Another closely related work is HyperGuard [32]. Rutkowska *et al.* suggested using SMM of the x86 CPU to monitor the integrity of the hypervisors. Although we have similar goals as the HyperGuard project, the use of a network card allows us to outsource the analysis of the state snapshot. This results in a drastic improvement in the performance of the system reducing the system busy time from seconds to milliseconds. Due to its low performance overhead, HyperCheck can also monitors the code and data of the privileged domain and underlying OSes. Another difference is that the monitoring machine can be used to detect the DoS attacks to the SMM code.

DeepWatch [10] also offers detection of hypervisor rootkits, called virtualization malware in DeepWatch, by using the embedded micro-controller(s) in the chipset. DeepWatch is signature based and used to detect rootkits relying on hardware-assisted virtualization technologies such as Intel VT-d [19]. Contrary, HyperCheck performs anomaly detection and thus can identify a larger class of software rootkits.

Flicker [24] uses a TPM based method to provide a minimum Trusted Code Base (TCB), which can be used to detect the modification to the kernels. Flicker requires advanced hardware features such as Dynamic Root of Trust Measurement (DRTM) and late launch. In contrast, HyperCheck uses the static Platform Configuration Registers (PCRs) to secure the booting process. In addition, by sending out the data, HyperCheck has a lower overhead on the target machine compared to Flicker. To reduce the overhead of Flicker, TrustVisor [25] has a small footprint hypervisor to perform some cryptography operations. However, all the legacy applications should be ported for TrustVisor to work. In addition, TrustVisor requires DRTM.

Another branch of research tries to improve the security of the hypervisor by adding hooks [13] and enforcing security policies between virtual machines [33]. These methods are hypervisor specific and run as the same level as the hypervisor. HyperCheck monitors the hypervisor state from a lower level and thus, is complementary to these methods.

Furthermore, there is a plethora of research aimed towards protecting the Linux kernel [6, 23, 18, 37, 21, 35, 30]. Baliga [6] *et al.* use a PCI device to acquire the memory and automatically derive the kernel invariance.

[2] As we discuss in Section 7, the same can be accomplished using a management interface.

Currently, we discover the kernel invariance manually but we could employ their techniques directly and without modifications. Litty [23] *et al.* developed a technique to discover the address of key data structures that are instantiated during run-time by relying on processor hardware and executable file specifications. But they also rely on the integrity of the underlying hypervisors. HyperCheck first obtains the virtual addresses of those symbols through the symbol file, but 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. Other existing research [37, 21, 35, 30], including work by Jiang *et al.*, depend on the integrity of the hypervisor to protect the kernel. Our work is complementary and can be employed as a meta-protection mechanism to guard the integrity of OS-level defenses. A lot of recent work has gone towards using SMM to generate efficient rootkits [38, 9, 17, 15]. These rootkits can be used either to get root privilege or as a key-stroke loggers. We use SMM to offer integrity protection by monitoring the state of hypervisors and operating systems.

3 Threat model

3.1 Background of System Management Mode

System Management Mode (SMM) was introduced in the Intel386 SL and Intel486 SL processors. It became a standard IA-32 feature in the Pentium [3] processor. SMM is a separate CPU mode besides the protected and real 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 a SMI is received from the advanced programmable interrupt controller (APIC).

In SMM, the processor switches to a separate address space, called system management RAM (SMRAM). In addition, all hardware context of the currently running code is saved in SMRAM. Then, the CPU, being in SMM, executes transparently code that is usually a part of BIOS and resides in SMRAM. The SMRAM can be made inaccessible from other CPU operating modes. Therefore, it can act as trusted storage, sealed from being accessed from any device or even the CPU (while not in SMM mode). In HyperCheck, we modify the SMM code to execute our monitoring functions. This modification of SMM code can be integrated into the BIOS. Another way is to use a trust boot mechanism or a management interface to upload the code to SMM (when SMRAM is not locked) and then lock the SMRAM. Upon returning from SMM, the processor is placed back into its state prior to enter SMM.

3.2 Attacker's capabilities

We assume that the adversary has following capabilities: she is able to exploit vulnerabilities in any software running in the machine after bootup. This includes the VMM and all of its privileged components. For instance, the attacker can compromise a guest domain and escape to the privileged domain. In Xen 3.0.3, pygrub [**?**] 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 [2]. Also, the attacker can modify the hypervisor code or data using any known or zero-day attacks. For instance, the DMA attack [39] hijacks a device driver to perform unauthorized DMA to the hypervisor's code or data.

3.3 General Assumptions

The attacker cannot tamper with or disable the installed PCI hardware without stopping or rebooting the machine. Also, if the SMM code is integrated with BIOS, we assume the SMRAM is properly setup by BIOS upon boot time. If the SMM code is not included in the BIOS, it has to be reliably uploaded to the SMRAM during boot. This can be done by either using trusted boot or using the management interface to bootstrap the computer. In this case, to initialize the SMM code, a trusted bootstrap mechanism has to be employed. The SMRAM is locked once it is properly set up. Once it is locked, we assume it cannot be subverted by the attacker (an assumption supported by current hardware). Attacks that attempt to modify the SMM code [40, 16] are beyond the scope of this paper.

3.4 In-scope Attacks

HyperCheck aims to detect the in-memory, Ring-0 level (hypervisor or general OS) rootkits and rootkits in privileged domains of hypervisors. A rootkit is a set of programs and code that allows a permanent or consistent, undetectable presence on a computer [20]. One kind of rootkits only modifies the memory and/or registers and runs in the kernel level. For example, the idt-hook rootkit [5] modifies the interrupt descriptor table (IDT) in the memory and then gains the control of the complete system. An stealthier version of the idt-hook rootkit 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. When it

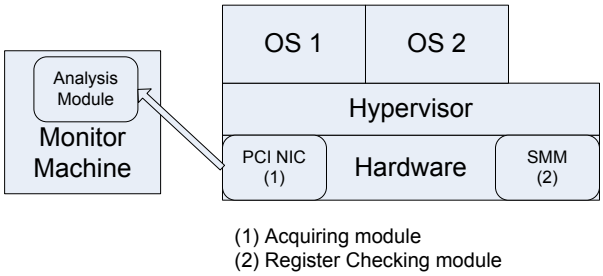

Figure 2: The architecture of HyperCheck

comes to the hypervisor level rootkit, there is yet another kernel: the hypervisor kernel which runs underneath the operating system kernel. There are existing methods to detect in-memory, kernel-level rootkits. We try to bridge this gap by introducing HyperCheck.

3.5 Limitations

Currently, our analysis cannot protect against attacks that modify dynamic data. There are two types of threats: modification to the dynamically generated function pointers and return-oriented attacks. In these attacks, the control flow is redirected to memory location controlled by the attacker. There are techniques to thwart such attacks: the non-executable bit in new CPUs and Address Space Layout Randomization to name a few. HyperCheck can leverage and integrate those techniques to provide full protection but it was not part of our implementation in this paper. Having said that, we can still detect the presence of the malfease if it tries to interfere with the VMM code or statically defined function pointer.

4 System Architecture

HyperCheck is composed of three key components: the physical memory acquiring module, the analysis module and the CPU register checking module. The memory acquiring module reads the contents of the physical memory of the protected machine and sends them to the analysis module. Then, the analysis module checks the memory contents and verifies if anything is altered. The CPU register checking module reads the registers and validates their integrity. The overall architecture of HyperCheck is shown in Figure 2. Before introducing the key components, we first describe our design principles.

Our main design principle is that HyperCheck should not rely on any software running on the machine except the boot loader. Since the software may be compromised, one cannot trust even the hypervisor. Therefore, we use hardware – a PCI Ethernet card – as a memory acquiring module and SMM to read the CPU registers. Usually, Ethernet cards are PCI devices with bus master mode enabled and are able to read the physical memory through DMA, which does not need help from CPU. SMM is an independent operating mode and could be made inaccessible from protected mode which is what the hypervisor and privileged domains run in.

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

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

4.1 Acquiring the physical memory

In general, there are two ways to acquire the physical memory: a software method and a hardware one. The former uses the interface provided by the OS or the hypervisor to access the physical memory, such as `/dev/kmem` on Linux [11] or `\\.\Device\PhysicalMemory` on Windows [36]. This method relies on the integrity of the underlying operating system or the hypervisor. If the operating system or the hypervisor is compromised, the malware may provide a false view of the physical memory. Moreover,

these interface to access memory can be disabled in future versions of the operating systems. In contrast, the hardware method uses a PCI device [12, 29] or other kinds of hardware [10]. The hardware method is more reliable because it depends less on the integrity of the operating system or the hypervisor.

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

Another class of attacks is denial of service attacks. Such attacks aim to stop or disable the device. For instance, according to ACPI [1] specification, every PCI device supports D3 state. This means that an ACPI-compatible device can be suspended by an attacker who takes over the operating system: ACPI was designed to allow the operating system to control the state of the devices. Of course, the OS is not a trusted component in our threat model. Therefore, one possible attack is to selectively stop the NIC without stopping any other hardware. To prevent ACPI DoS attacks, we need an out-of-band mechanism to verify that the PCI card is not disabled. The remote server that receives the state snapshots plays that role.

4.2 Translating the physical memory

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

The current analysis module depends on three properties of the kernel memory: linear mapping, static nature and persistence. Linear mapping means the kernel (OS or hypervisor) memory is linearly mapped to physical memory and the physical addresses are fixed. For example, on x86 architecture, the virtual memory of Xen hypervisor is linearly mapped into the physical memory. Therefore, in order to traslate the physical address to a given virtual address in Xen, we have to subtract the virtual address from an offset. In addition, Domain 0 of Xen is also linear mapped to the physical memory. The offset for Domain 0 is different on different machines but remains the same on a given machine. Moreover, other operating system kernels, such as Windows [34], Linux [8] or OpenBSD [15], also have this property when they are running directly on the real hardware.

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

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

4.3 Reading and verifying the CPU registers

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

We choose to use SMM in x86 CPU which is similar to a hardware method. As we mentioned earlier, SMM is a different CPU mode from the protected mode which the hypervisor or the operating system reside in. When CPU switches to SMM, it saves the register context in the SMRAM. The default SMRAM size is 64K Bytes beginning at a base physical address in physical memory called the SMBASE. The SMBASE default value following a hardware reset is $0x30000$. The processor looks for the first instruction of the SMI handler at the address [SMBASE + 0x8000]. It stores the processor's state in the area from [SMBASE + 0xFE00] to [SMBASE + 0xFFFF]. In SMM, if SMI handler issues `rsm` instruction, the proces-

sor will switch back to the previous mode (usually it is protected mode). In addition, the SMI handler can still access I/O devices. HyperCheck verifies the registers in SMM and reports the result by sending it via the Ethernet card to the monitor machine. HyperCheck focuses on monitoring two registers: IDTR and CR3. IDTR should never change after system initialization. For CR3, SMM code can use it to translate the physical addresses of the hypervisor kernel code and data. The offsets between physical addresses and virtual ones should never change as we discussed in Section 4.2.

5 Implementation

We implemented two prototypes for HyperCheck. HyperCheck-I is based on QEMU full system emulation while HyperCheck-II is based on a physical machine. HyperCheck-I is first developed for quick prototyping and debugging. To measure the actual performance, we then developed HyperCheck-II. Both of them utilize Intel e1000 Ethernet card as the acquiring module.

We first introduce HyperCheck-I. The target machine runs as a virtual machine in QEMU. The analysis module runs on the host operating system of QEMU. For the acquiring module, we put a small NIC driver into SMM of the target machine. It drives the NIC to transmit the contents of physical memory as an Ethernet frame. On the monitoring machine, an analysis module receives the packet from the network. The analysis module then compares contents of the physical memory with the original (initial) versions. If a new snapshot of the memory contents is different from the original one, the module will report the event and the administrator can then make a decision. Moreover, the small program runs in the SMM checks the CPU registers and reports the result via the Ethernet card.

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

5.1 Memory Acquiring module

As mentioned earlier, we used QEMU full system emulation mode for HyperCheck-I. QEMU is suitable for debugging problems. But it also has two drawbacks. First, the throughput of a QEMU network card is much lower than a real device. For our QEMU based prototype, the network card throughput is just about 10MB/s, although Gigabit Ethernet cards are common in real world. Second, the performance measurement on QEMU may not reflect the real world performance. HyperCheck-II overcomes these problems.

The main task to implement the acquiring module is to port the e1000 network card driver into SMM to scan the memory and send it out. Normally, SMM code is one part of BIOS. Since we don't have the source code of the BIOS, we used the method similar to the one mentioned in [9] to modify the default SMM code. Basically, it writes the SMM code in 16bit assembly and uses a user level program to open the SMRAM and copy the assembly code to the SMRAM.

To overcome the limitations of [9], we split the e1000 driver into two parts: initialization and data transfer. The initialization part is complex and very similar to the Linux driver. The transferring part is simpler and different from the Linux driver. Therefore, we modified the existing Linux e1000 driver to initialize the network card and only program the transferring part in assembly. The e1000 driver on Linux is changed to only initialize the NIC but does not send out any packet. The assembly code is compiled to an ELF object file. Next, we wrote a small loader which can parse the ELF object file and load the code and data to the SMM.

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

To prevent replay attacks, a secret key is transferred from the monitor machine to the target machine when it is booting. The key is used to create a random seed to selectively hash a portion of the data. If we hash the entire data, the performance impact may be high. To reduce the overhead, we use the secret key as a seed to generate one big random number used for one-time pad encryption and another set of serial random numbers. The serial of random numbers are used as the indexes of the positions of the memory being scanned. Then the content at these positions are XORed with the one-time pad with the same length before starting NIC DMA. After the transmission is done, the memory content is XORed

again to restore the original value.

The NIC driver also checks the loop back setting of the NIC before sending the packet. To further guarantee the data integrity ,the SMM NIC driver stays in the SMM until all the packet is written to the internal FIFO of the NIC, and add 64KB more data to the end to flush the internal FIFO of the NIC. Therefore, the attacker cannot use loopback mode to get the secret key or peek into the internal NIC buffer through debugging registers of the NIC.

5.2 Analysis module

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

Currently, the analysis module can check the integrity of the control data and code. The control data includes IDT table, hypercall table and exception table of Xen, and the code is the code part of Xen hypervisor. To find out the physical address of these control tables, we use `Xen.map` symbol file. First, we find the virtual addresses of `idt_table`, `hypercall_table` and exception_table. Then the physical address of these symbols is virtual_address − 0xff00,0000 on x86-32 architecture with PAE. The address of Xen hypervisor code is between `_stext` and `_etext`. HyperCheck can also monitor the control data and codes of Domain 0. This includes the system call table and the code part of Domain 0 (a modified Linux 2.6.18 kernel). The kernel of Domain 0 is also linearly mapped to the physical memory. We use a kernel module running in Domain 0 to compute the exact offset. On our test machine, the offset is 0x83000000. Note that, there is no IDT table for Domain 0, because there is only one such table in the hypervisor. We input these parameters to the acquiring module to improve the scan efficiency.

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

5.3 CPU register checking module

HyperCheck uses SMM code to acquire and verify CPU registers. In a product, the SMI handler should be integrated into BIOS. Or it can be set up during the system boot time. This requires the bootstrap to be protected by some trusted bootstrap mechanism. In addition, most chipsets provide a function to lock the SMRAM. Once it is locked, SMM handler cannot be changed until reboot. Therefore, the SMRAM should be locked once it is set up. In our prototype, we used the method mentioned in Section 5.1 to modify the default SMM code.

There are three steps for CPU register checking: 1) triggering SMI to enter SMM; 2) checking the registers in SMM; 3) reporting the result. SMI is a hardware interrupt and can only be triggered by hardware. Normally, I/O Controller Hub (ICH), also called Southbridge, defines the events to trigger SMI. For HyperCheck-I, the QEMU emulates Intel 82371SB chip as the Southbridge. It supports some device idle events to generate SMI. SMI is often used for power management, and Southbridge provides some timers to monitor the state of a device. If that device remains idle for a long time, it will trigger SMI to turn off that device. The resolutions of these timers are typically one second. However, on different motherboard, the method to generate the SMI may be different. Therefore, we employ the Ethernet card to trigger the SMI event.

For the register checking, HyperCheck monitors IDTR and CR3 registers. The contents of IDTR should never change after system boot. The SMM code just reads this register by `sidt` instruction. HyperCheck uses CR3 to find out the physical addresses of hypervisor kernel code and data given their virtual addresses. Essentially, it walks through all the page tables as a hardware Memory Management Unit (MMU) does. Note that offset between the virtual address and the physical address of hypervisor kernel code and data should never change. For example, it is 0xff000000 for Xen 32bit with PAE. The Domain 0 has the same property. The SMM code requires the virtual address range as the input (this can be obtained through the symbol file and send to the SMM in the boot time) and then check their physical addresses. If any physical address is not equal to virtual address – off-

set, then it is a possible attack. The SMM code reports the result of this checking via the Ethernet card. The assembly code of it is just 67 LOC.

As we already mentioned, the SMM code uses the Ethernet card to report the result. Without the ethernet card, it is difficult to send the report reliably without stopping the whole system. For example, the SMM code could write the result to a fixed address of physical memory. But according to our threat model, the attacker has access to that physical memory and can easily modify the result. Or the SMM code could write it to the hard disk. Again, this can be altered by the attacker too. Since security cannot relies on the obscurity, the only way left without a network card is to stay in the SMM mode and put the warning message on the screen. This is reliable, but the system in the protected mode becomes completely frozen. Sometimes, it may not be desirable, and could be abused by the attacker to launch Denial of Service attacks.

5.4 HyperCheck-II

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

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

In addition, we built a debugging interface for the SMM code on the physical machine. We use the reserved physical memory to pass the information between the SMM code and the normal OS. This interface is also used to measure the performance of the SMM code as we will discuss in Section 6.

6 Evaluation

To validate the correct operation of HyperCheck, we first verified the properties that need to hold for us to be able to protect the underlying code as we discussed in Section 4.2. Then, we tested the detection for hypervisor rootkits and measured the operational overhead of our approach. We have two testbeds: testbed 1 is mainly used for HyperCheck-I and also used as the monitor machine for HyperCheck-II. Testbed 2 uses HyperCheck-II to produce the plotted performance overhead on the real hardware. Testbed 1 was equipped with a Dell Precision 690 with 8GB RAM and one 3.0GHz Intel Xeon CPU with two cores. The host operating system was CentOS 5.3 64bit. The QEMU version was 0.10.2 (without kqemu). The Xen version was 3.3.1 and Domain 0 was CentOS 5.3 32bit with PAE. Testbed 2 was a Dell Optilex GX 260 with one 2.0GHz Intel Pentium 4 CPU and 512MB memory. Xen 3.1 and Linux 2.6.18 was installed on the physical machine and the Domain 0 is CentOS 5.4.

6.1 Verifying the static property

An important assumption is that the control data and respective code are statically mapped into the physical memory. We used a monitoring module designed to detect legitimate control data and code modifications throughout the experiments. This enabled us to test our approach against data changes and self-modifying code in the Xen hypervisor and Domain 0. We also tested the static properties of Linux 2.6 and Windows XP 32bit kernels. In all these tests, the hypervisor and the operating systems are booted into a minimal state. The symbols used in the experiments are shown in Table 1. During the tests, we found out that during boot the control data and the code changes. For example, the physical memory of IDT is all 0 when the system first boots up. But after several seconds, it becomes non-zero and static. The reason is that the IDT table is initialized later in the boot process.

6.2 Detection

To verify whether HyperCheck can detect attacks against the hypervisor, we implemented DMA attacks [39] on Xen hypervisor and then tested HyperCheck-I's response on testbed 1. We ported the HDD DMA attacks to modify the Xen hypervisor and Domain 0. There are four

Table 1: Symbols for Xen hypervisor, Domain 0, Linux and Windows

System	Symbol	Use
Xen	idt_table	Hypervisor's Interrupt Descriptor Table
	Hypercall_table	Hypervisor's Hypercall Table
	exception_table	Hypervisor's Exception Table
	_stext	Beginning of hypervisor code
	_etext	Bnd of hypervisor code
Dom0	sys_call_table	Domain 0's System Call Table
	_text	Beginning of Domain 0's kernel code
	_etext	End of Domain 0's kernel code
Linux	idt_table	Kernel's Interrupt Descriptor Table
	sys_call_table	kernel's System Call Table
	_text	Beginning of kernel code
	_etext	End of kernel code
Windows	PCR→idt	Kernel's Interrupt Descriptor Table
	KiServiceTable	Kernel's System Call Table

attacks to Xen hypervisor and two attacks to Domain 0. We also modified the pcnet network card in QEMU to perform the DMA attack from the hardware directly. The modified pcnet NIC is used to attack Linux and Windows operating systems. There are three attacks to Linux 2.6.18 kernel and two attacks to Windows XP SP2 kernel, each targeting one control table or the code. They can modify the IDT table and other tables of the kernel. HyperCheck-I correctly detected all these attacks by reporting the contents of memory in the target machine are changed.

6.3 Monitoring overhead

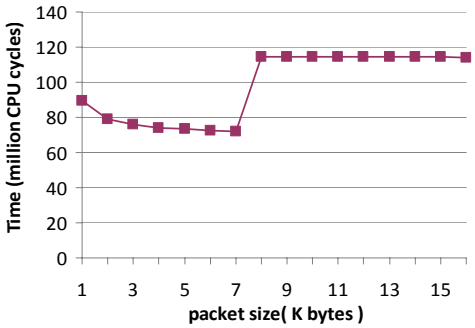

Figure 3: Network overhead for variable packet size.

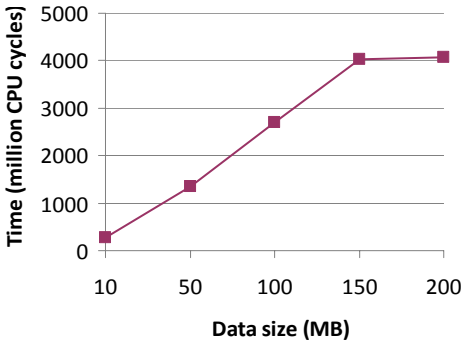

Figure 4: Network overhead for variable data size.

The primary source of overhead is coming from the transmission of the memory contents to the external monitoring machine. In addition, to ensure the memory contents have not been tampered with, HyperCheck needs to remain in SMM and wait until the NIC finished. Otherwise, the attacker may control the OS and modify the memory contents or the transmit descriptor in the main memory while transmitting. Initially, we measured the time to transmit a single packet varying its payload size. The packet flushed out when the Transmit Descriptor Head register (TDH) is equal to Transmit Descriptor Tail register (TDT). We calculated the elapsed time using the `rdtsc` instruction to read the time stamp before

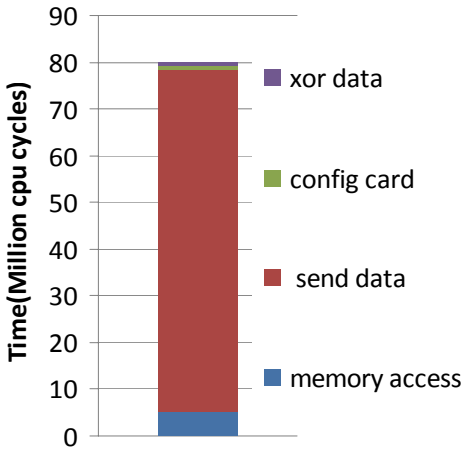

Figure 5: Overhead of the operations in SMM

and after each operation. As expected, the time linearly increases as the size of the packet increases.

Next, we measured the bandwidth by using different packet sizes to send out a fixed amount of data: 2881 KB memory (the size of Xen code plus Domain 0 code). The result is depicted in the Figure 3: when the packet size is less than 7 KB, the time required to send the data similar to a constant value. When the packet size becomes 8KB, the overhead increases dramatically and it remains high. The reason is that the internal NIC transfer FIFO is 16KB. Therefore, when the packet size becomes 8KB or larger, the NIC cannot hold two packets in the FIFO at the same time and this introduces delay.

Since HyperCheck can be used to monitor different sized hypervisors and OSes, we measured the time required to send different amount of data and the results are in Figure 4. In this set of experiments, we use 7KB as the packet size since it introduced shortest delay in our testbed. We can see that the time also nearly linearly increased with the amount of memory. In addition to PCI scanning, HyperCheck also triggers SMI interrupt every one second and checks the registers in SMM. To measure the overall overhead of entering SMM, executing SMM code and return from SMM, we wrote a kernel module running in Domain 0.

The tests were conducted on testbed 2 (HyperCheck-II) and each test is repeated many times. Here we present the average of the results. The overall time is composed of four parts. First, the time taken to XOR the data with the secret key. Second, the time to access the memory. Third, the time to configure the card and switch from protected mode to SMM and back. Finally, the time to send out the data through the NIC. To find out how much time was spent in each part, we wrote two more test programs. One is a dummy SMM code which does nothing but just returns from SMM to CPU protected mode. The other one does not access the main memory but just

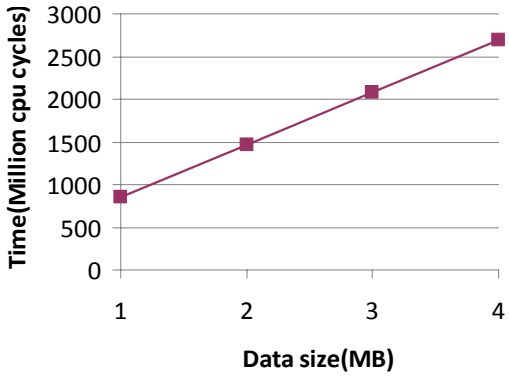

Figure 6: Overhead of the XOR data in SMM

		Execution Time(ms)		
Code base	Size(MB)	HC	SMM	TPM
Linux	2.0	31	736	1022
Xen+Dom0	2.7	40	981	>1022
Window XP	1.8	28	709	> 972
Hyper-V	2.4	36	876	>1022
VMWare ESXi	2.2	33	825	>1022

Table 2: Time overhead of HyperCheck and other methods

use the registers to simulate the verification of IDTR and CR3. Then we tested the running time for these two SMM codes. From the first one, we can get the time for switching between protected mode and SMM and then switch back. From the second one, we can get the time for the CPU computation part of the verification of IDTR and CR3.

The results are presented in Figure 5. The most of the time is spent in sending the data, which is 73 Million cycles. Next is the time to accessing the main memory : 5.28 Million cycles. Others took a very small portion. The total time is 80 Million cycles. Since the CPU of the testbed 2 is 2 GHz. Therefore, the SMM code consumes 6.5% of the CPU cycles, and takes 40 ms.

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

In contrast, previous research suggests using SMM to read the memory and hashing it on the target machine. We call this SMM only method. To compare our approach with SMM only method, we wrote a program to XOR the memory in SMM with different sizes. The result is shown in Figure 6.

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

Table 3: Comparison between HyperCheck and other methods

The time for XOR data is linearly increased with the amount of data and typically uses thousands of CPU cycles. Also, we compare our approach with a TPM based approach [24] which can also be used to monitor the integrity of the kernels. The result is shown in the Table 2. HC stands for HyperCheck. We can see that the overhead of HyperCheck is one magnitude lower than SMM-only and TPM based method. For SMM-only, it has to hash the entire data to check its integrity, while HyperCheck only hashes a random portion of the data and then sends the entire data out using an Ethernet card. For TPM based method, the most expensive operation is TPM quote, which alone took 972 ms. Note that the test machine of TPM based method is better than our testbed 2. An overall comparison between HyperCheck and other methods is shown in Table 3. We can see that only HyperCheck can monitor both memory and registers with low overhead.

7 Security Analysis & Limitations

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

In addition, HyperCheck also uses SMM to monitor CPU registers, which provides further protection. Some pervious research works only rely on the symbol table in the symbol file to find the physical address of the kernel code and data. Nonetheless, there is no binding between the addresses in the symbol table and the actual physical address of these symbols [23]. For example, one potential attack is to modify the IDTR register of CPU to point to another address. Then the malware can modify the new IDT table, keeping 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 addresses in the IDTR will actually point to different physical addresses. HyperCheck can detect these cases by checking CPU registers in SMM. In SMM, HyperCheck read the content of IDTR and CR3 registers used by the operating system. IDTR should never change after booting. If it changed, SMM will send a warning through the Ethernet card to the monitor machine. From CR3, HyperCheck can find the actual physical address given the virtual ones. The offset between the virtual addresses and the physical addresses should be static. If some offsets changed, HyperCheck will generate a warning too. Moreover, PCI devices including the Ethernet card alone can be cheated to get a different view of the physical memory [31]. With SMM, we could avoid this problem by checking the corresponding settings in SMM.

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

In addition, the attacker may try to launch a fake reboot attack to get a private key from the monitor machine. It can mimic the SMM NIC driver and send a request for a new key. For this event, we have two options: first, we could use Trusted Platform Module (TPM) based remote attestation to verify the running state of the target machine [24]. 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 not have a TPM, the second method is to send another reliable reboot signal to the target machine when it asks for the key to make sure the SMM code is running.

However, HyperCheck also has its limitations. It cannot detect the changes which happen between the two consecutive memory and register scans. Although the time window between the scans is just one second in the current prototype, malware can still potentially make some changes in the time interval and restore it before the next scan. To address this problem, we could randomize the scan interval to increase the chances for detection. In addition, we could use high bandwidth devices, such as PCI Express, which is able to reach 5GT/s transfer rate [28], to minimize the scan interval.

In addition, if the memory mappings of the hypervisor do not hold the three properties (linear mapping, persistence and static nature), the current version of HyperCheck cannot deal with it. We will try to address these problems in the future.

8 Conclusions

In this paper, we introduced a hardware-assisted tampering detection framework designed to protect the software code integrity of VMMs. We rely on the CPU System Managed Mode (SMM) to securely generate and transmit the full state of the protected machine to an external server. Moreover, we implemented two prototypes to demonstrate the performance and feasibility of our approach: using an Ethernet card and a commodity x86 machine.

Our experiments show we can successfully identify tampering to the control data and the code part of the Xen hypervisor. In addition, we also used HyperCheck to protect Domain 0 in Xen and other operating systems, such as Linux and Windows. HyperCheck does not rely on the software running on the target machine. Moreover, HyperCheck is robust against attacks that aim to disable or block its operation are relatively lightweight: it can produce and communicate a scan of the state of the protected software in less than 40ms.

References

[1] ACPI http://www.acpi.info/.

[2] Cve-2007-4993.

[3] Intel® 64 and ia-32 architectures software developers manual volume 1.

[4] National vulnerability database, http://nvd.nist.gov/.

[5] K. Adamyse. Handling interrupt descriptor table for fun and profit. Phrack 59, 2002.

[6] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *ACSAC '08: Proceedings of the 2008 Annual Computer Security Applications Conference*, pages 77–86, Washington, DC, USA, 2008. IEEE Computer Society.

[7] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[8] D. Bovet and M. Cesati. *Understanding the Linux kernel 3rd edition*. O'Reilly Media, 2005.

[9] BSDaemon, coideloko, and D0nAnd0n. System Management Mode Hack: Using SMM for "Other Purposes". *Phrack Magazine*, 2008.

[10] Y. Bulygin and D. Samyde. Chipset based approach to detect virtualization malware a.k.a. DeepWatch. *Blackhat USA*, 2008.

[11] M. Burdach. Digital forensics of the physical memory. *Warsaw University*, 2005.

[12] B. D. Carrier and J. Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1(1):50 – 60, 2004.

[13] G. Coker. Xen security modules (xsm). *Xen Summit*, 2006.

[14] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *In Proceedings of the ACM Symposium on Operating Systems Principles*, 2003.

[15] L. Duflot, D. Etiemble, and O. Grumelard. Using CPU System Management Mode to Circumvent Operating System Security Functions. In *Proceedings of the 7th CanSecWest conference*. Citeseer, 2001.

[16] L. Duflot, D. Etiemble, and O. Grumelard. Security issues related to pentium system management mode. In *Cansecwest security conference Core06*, 2006.

[17] S. Embleton, S. Sparks, and C. Zou. SMM rootkits: a new breed of OS independent malware. In *Proceedings of the 4th international conference on Security and privacy in communication netowrks*, page 11. ACM, 2008.

[18] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *In Proc. Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[19] R. Hiremane. Intel® Virtualization Technology for Directed I/O (Intel® VT-d). *Technology© Intel Magazine*, 4(10), 2007.

[20] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.

[21] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and communications security*, page 138. ACM, 2007.

[22] G. H. Kim and E. H. Spafford. The design and implementation of tripwire: a file system integrity checker. In *CCS '94: Proceedings of the 2nd ACM Conference on Computer and communications security*, pages 18–29, New York, NY, USA, 1994. ACM.

[23] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In *SS'08: Proceedings of the 17th conference on Security symposium*, pages 243–258, Berkeley, CA, USA, 2008. USENIX Association.

[24] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 315–328. ACM, 2008.

[25] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2010.

[26] D. Murray, G. Milos, and S. Hand. Improving Xen security through disaggregation. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 151–160. ACM, 2008.

[27] B. Payne, M. de Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 385–397, Dec. 2007.

[28] PCI-SIG. PCI Express 2.0 Frequently Asked Questions.

[29] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 13–13, Berkeley, CA, USA, 2004. USENIX Association.

[30] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Recent Advances in Intrusion Detection*, pages 1–20. Springer.

[31] J. Rutkowska. Beyond the CPU: Defeating hardware based RAM acquisition. *Proceedings of BlackHat DC 2007*, 2007.

[32] J. Rutkowska and R. Wojtczuk. Preventing and detecting Xen hypervisor subversions. *Blackhat Briefings USA*, 2008.

[33] 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.

[34] S. Schreiber. *Undocumented Windows 2000 secrets: a programmer's cookbook*. Addison-Wesley, 2001.

[35] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, page 350. ACM, 2007.

[36] T. Vidas. The acquisition and analysis of random access memory. *Journal of Digital Forensic Practice*, 1(4):315–323, 2006.

[37] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 545–554. ACM, 2009.

[38] F. Wecherowski and core collapse. A Real SMM Rootkit: Reversing and Hooking BIOS SMI Handlers. *Phrack Magazine*, 2009.

[39] R. Wojtczuk. Subverting the Xen hypervisor, 2008.

[40] R. Wojtczuk and J. Rutkowska. Attacking SMM Memory via Intel® CPU Cache Poisoning.