Nighthawk: Transparent System Introspection from Ring -3

Lei Zhou[1,2], Jidong Xiao[3], Kevin Leach[4], Westley Weimer[4], Fengwei Zhang[2,5](✉), and Guojun Wang[6]

[1] Central South University, Changsha, China
[2] Wayne State University, Detroit, USA
{gn6392,fengwei}@wayne.edu
[3] Boise State University, Boise, USA
jidongxiao@boisestate.edu
[4] University of Michigan, Ann Arbor, USA
{kjleach,weimerw}@umich.edu
[5] SUSTech, Shenzhen, China
[6] Guangzhou University, Guangzhou, China
csgjwang@gzhu.edu.cn

Abstract. During the past decade, virtualization-based (e.g., virtual machine introspection) and hardware-assisted approaches (e.g., x86 SMM and ARM TrustZone) have been used to defend against low-level malware such as rootkits. However, these approaches either require a large Trusted Computing Base (TCB) or they must share CPU time with the operating system, disrupting normal execution. In this paper, we propose an introspection framework called Nighthawk that transparently checks system integrity at runtime. Nighthawk leverages the Intel Management Engine (IME), a co-processor that runs in isolation from the main CPU. By using the IME, our approach has a minimal TCB and incurs negligible overhead on the host system on a suite of indicative benchmarks. We use Nighthawk to check the integrity of the system software and firmware of a host system at runtime. The experimental results show that Nighthawk can detect real-world attacks against the OS, hypervisors, and System Management Mode while mitigating several classes of evasive attacks.

1 Introduction

Security vulnerabilities [28] that enable unauthorized access to computer systems are discovered and reported on a regular basis. Upon gaining access, attackers frequently install various low-level malware or rootkits [2] on the system to retain control and hide malicious activities. While many solutions target different specific threats, the key ideas are similar: the defensive technique or analysis

L. Zhou—Work was done while visiting COMPASS lab at Wayne State University.

K. Sako et al. (Eds.): ESORICS 2019, LNCS 11736, pp. 217–238, 2019.
https://doi.org/10.1007/978-3-030-29962-0_11

gains an advantage over the attacker by executing in a more privileged context. More specifically, to detect low-level malware, virtualization-based defensive approaches [20,21] and hardware-assisted defensive approaches [5,27,32,46] have been proposed. However, both approaches come with inherent limitations.

Limitations in Virtualization. Virtualization-based approaches require an additional software layer (i.e., the hypervisor) to be introduced into the system, resulting in two problems. First, virtualization can incur significant performance overhead. While CPU vendors and hypervisor developers have worked to improve the performance of CPU and memory virtualization, the cost of I/O virtualization remains high [25]. Second, and more importantly, mainstream hypervisors have a large trusted computing base (TCB). Hypervisors such as Xen or KVM contain many thousands of lines of code in addition to the millions of lines present in the control domain. Thus, while virtualization has facilitated significant defensive advances in monitoring the integrity of a target operating system, attackers in such systems can target the hypervisor itself. By exploiting vulnerabilities in the large TCB of the hypervisor, attackers can escape the virtualized environment and wreak havoc on the underlying system.

Limitations in Hardware. Hardware-assisted approaches are not burdened by large TCBs. However, to provide a trustworthy execution environment, hardware-assisted approaches typically require either (1) an external monitoring device or (2) specialized CPU support for examining state such as Intel System Management Mode (SMM). The former, seen in Copilot [32], Vigilare [27], and LO-PHI [37], typically use a co-processor (on a PCI card or an SoC) that runs outside of the main CPU. Such a requirement increases costs and precludes large-scale deployment. The latter, seen in HyperSentry [5], HyperCheck [48] runs code in SMM and monitors the target host system. While it does not require any external devices, code running in SMM can disrupt the flow of execution in the system. Running code in SMM requires the CPU to perform an expensive context switch from the OS environment to SMM. This switch suspends the OS and application execution until the SMM code completes, that is benefit for static analyzing the current host running state. But this suspension of execution results in abnormalities (e.g., lost clock cycles) that are detectable from the OS context. Attackers can measure and exploit such abnormalities so as to escape detection or hide malicious activities.

To address the limitations of current approaches, we present Nighthawk, a framework leveraging the Intel Management Engine (IME). While the IME is intended as an advanced system management feature (e.g., for remote system administration of power and state), in this work, we leverage the IME to construct a system introspection framework which is capable of efficiently checking the integrity of critical kernel and hypervisor structures and system firmware. To the best of our knowledge, this is the first paper to consider using the IME for such application. Our proposed framework offers the following advantages in comparison to previous work:

- **No extra hardware required.** The IME has been integrated in virtually every Intel processor chipset since 2008. Therefore, the proposed framework can be deployed in most current commercially-available Intel-based computer systems without requiring external peripheral support.
- **High privilege.** As a co-processor running independently from the main CPU, the IME has a high privilege level in a computer system.[1] The IME has unrestricted access to the host system's resources, making it suitable for analyzing the integrity of the underlying operating system, hypervisor, or firmware.
- **Small TCB.** The IME runs a small independent Minix 3 OS distribution. As Minix 3 uses a microkernel architecture, it contains only thousands of lines of kernel code (cf. millions of lines of code in modern hybrid architecture systems like Linux or Windows). The reduced size of code results in a decreased trusted code base.
- **Low overhead.** Since the IME runs in an isolated co-processor, executing code in the IME does not disrupt the normally-executing tasks on the main CPU and does not compete for resources with the underlying OS. Thus, code executing in the IME incurs very little overhead on the target system.[2]
- **Transparency.** In addition to low overhead, the isolation of the IME means that the host OS are not aware of code executing in the IME. This allows transparent analysis of the host system from the IME.

We apply our prototype to several indicative experiments in which we verify the integrity of (1) kernel code, (2) virtualization system core code, and (3) System Management RAM. Our experimental results show that Nighthawk is able to detect real-world rootkits, including kernel-level rootkits and SMM rootkits, and incurs minimum performance overhead on the target system. Our main contributions are:

- We present Nighthawk, a novel introspection framework that transparently checks the integrity of the host system at runtime. We leverage the Intel Management Engine, an extant co-processor that runs alongside the main CPU, enabling a minimal TCB and detection of low-level system software attacks while incurring negligible overhead.
- We demonstrate a prototype of Nighthawk that can detect real-world attacks against operating system kernels, Xen and KVM hypervisors, and System Management RAM. Furthermore, Nighthawk is robust against page table manipulation attacks and transient attacks.
- Nighthawk causes low latency to verify the integrity of critical data structures. Our results show that Nighthawk takes 0.502 s to verify the integrity of the system call table (4 KB) of the host operating system. This low latency results in a small system overhead on the host.

[1] Expanding on Intel's privilege rings, userspace applications are said to have ring 3 privilege while the kernel has ring 0 privilege. The IME is said to have ring -3 privilege [12,40].

[2] Cache contention and bus bandwidth limits may incur overhead.

2 Background

We introduce the Intel Management Engine and System Management Mode.

Intel Management Engine: The Intel Management Engine is a subsystem which includes a separate microprocessor, its own memory, and an isolated operating system [13]. The IME has been integrated into Intel x86 motherboards since 2008 and was frequently used for remote system administration. Once the system is powered on, the IME runs in isolation, and its execution is not influenced by the host system on the same physical machine. To contact with isolated IME from host system, Intel designed the Host Embedded Controller Interface (HECI, also called Management Engine Interface) to secure exchange data between host memory and IME. Note that some other chipsets integrated co-processors, like the Intel Innovation Engine [16], also have the similar features, but are designed for special platforms (e.g., Data Center Servers) rather than for ordinary computers. Thus, in this paper, we build our introspection framework based on the IME rather than the Innovation Engine.

System Management Mode: System Management Mode (SMM) is a highly privileged execution mode included in all current x86 devices since the 386. It is used to handle system-wide functions such as power management or vendor-specific system control. SMM is used by the system firmware, but not by applications or normal system software. The code and data used in SMM are stored in a hardware-protected memory region named SMRAM. Under normal operation, SMRAM is inaccessible from outside of SMM unless configured otherwise (i.e., if SMRAM is unlocked). SMM code is executed by the CPU upon receiving a system management interrupt (SMI), causing the CPU to switch modes to SMM (e.g., from protected mode). The hardware automatically saves the CPU state, including control registers like CR3, in a dedicated region in SMRAM. After executing SMM code, the CPU state is restored and it resumes execution as normal. We use SMM in tandem with the IME to transparently gather accurate data from a system, even when it is compromised.

3 Threat Model and Assumptions

In this work, we assume the operating system, the hypervisor, and even SMM are not trusted. In contrast, due to its isolation and small TCB, we favor deploying security-critical software in the IME. We use this environment to run our code and introspect activities occurring in the operating system, the hypervisor, and SMM. Additionally, we assume an attacker does not have physical access to the machine. We assume that we start with a trustworthy firmware image (i.e., BIOS) so that we can reliably insert our IME introspection code. We assume the booting process of the Intel TXT [18] is trusted. We assume SMM could be compromised via a software vulnerability at runtime. However, attacks against SMM due to architectural bugs like cache poisoning [44] are out of scope because such attacks can be mitigated with official patch [45]. We assume the hardware can be trusted to function normally (e.g., hardware trojans are out of scope).

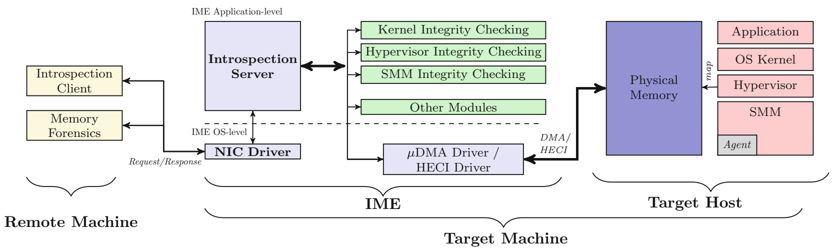

Fig. 1. High level architecture of the Nighthawk. The user operates a Remote Machine to interact with the Target Machine. We place custom IME code on the Target Machine, consisting of an Introspection server and several Integrity Checking Modules. When the user invokes an integrity checking command, the server dispatches the corresponding Integrity Checking Module, which in turn creates a communication channel with the Target Host's physical memory using either μDMA or HECI. We place custom SMM "Agent" code on the Target Host. The SMM Agent is capable of basic introspection to recover critical data structures, which can be transmitted to the IME using the same μDMA/HECI channel. The Introspection Server can transmit the resulting data back to the Remote Machine for analysis or forensics.

4 System Architecture

In Nighthawk, we leverage the IME to transparently monitor the integrity of the target system's memory (i.e., code and data) belonging to the kernel, any hypervisor present, and SMRAM. When an integrity violation is detected, our IME code asserts that an attack has occurred. Our system consists of a *Target Machine*, which we seek to protect, and a *Remote Machine*, which is used to interact with the Target Machine. An overview of Nighthawk is shown in Fig. 1, we then describe those key roles in more detail.

Target Machine: It represents the potentially vulnerable system we want to analyze and protect. The Target Machine contains both the IME and an underlying Target Host (e.g., operating system or hypervisor). We use the IME as the key component in Nighthawk to transparently introspect the Target Machine's physical memory. An Introspection Client, which is deployed on the Remote Machine, allows the user to send introspection commands to the Target Machine's IME. An Introspection Server on the Target Machine's IME then processes these commands. The Introspection Server invokes an analysis module on behalf of the Remote Machine. In this paper, we implemented three integrity checking modules: (1) kernel, (2) hypervisor, and (3) SMM. Each module corresponds to a particular class of attack that may occur against the Target Machine.

When the Introspection Server processes a command from the Client, we initialize the corresponding module and acquire the Target Host's memory. We use μDMA to access the host's memory. By design, μDMA only understands physical addresses, so we bridge the semantic gap to understand the Host's high-level abstractions (i.e., virtual memory addresses). We perform some initial reconnaissance on the Target Host's memory—we collect virtual memory addresses of some critical kernel/hypervisor data structures to derive a mapping to physical addresses. In SMM, we first build a SMRAM static configuration map for comparison at runtime. This map allows us to retrieve virtual memory addresses from the physical memory regions we acquire via μDMA. Next, we create a communication channel between the Target Machine's physical memory space and the IME's external memory space by using μDMA and HECI. This channel enables transferring critical data structures (e.g., the system call table, a hypervisor's kernel text, and saved architectural state) to the IME. Afterwards, each integrity checking module is able to locate relevant data structures in the IME's external memory space and perform integrity checking.

Remote Machine: It serves as a way for a user to remotely access the Target Machine and assess its integrity transparently. More specifically, the Remote Machine implements a simple Introspection Client that allows access to the Target Machine's IME remotely. Users can issue commands using the Introspection Client, and receive results from the Target Machine's IME. We implement several commands that are usable by the Introspection Client, including fetching segments of kernel memory for verification. We also implement a Memory Forensics Helper for dumping memory images to the Remote Machine for offline analysis. Due to the resource-constrained nature of the IME processor, it is more efficient to dump memory from the Target Machine and use the Remote Machine to perform more computationally-expensive analyses. Users can develop more complex memory forensic analysis helper based on their needs.

Both the Introspection Client and the Memory Forensics Helper work in tandem to communicate with the IME on the Target Machine. Rather than developing a custom communication protocol, we rely on the existing IME remote management protocol [42], which is a RESTful HTTPS protocol for remote management tasks. We reverse-engineered the protocol to augment it with custom commands used by our integrity checking code.

Integration: To summarize, we seek to protect a Target Machine from malicious attacks using the IME. We use custom IME code to implement integrity checking for the Target Machine's kernel, hypervisor, and SMM code and data. A user can interact with an Introspection Client to perform various integrity checking tasks. Because the IME enables transparent and low-overhead access to the Target Machine's physical memory, we can detect the presence of advanced attacks by leveraging a combination of integrity checks and introspection.

5 Implementation

In this section, we describe implementation details pertinent to our prototype of Nighthawk. We embedded custom IME firmware on the Target Machine to transparently acquire the Target Host memory with low overhead. Loosely, there are two main parts of the implementation: (1) preparing the Target Machine with custom IME firmware, and (2) interacting with the Target Machine's IME at runtime (Fig. 2).

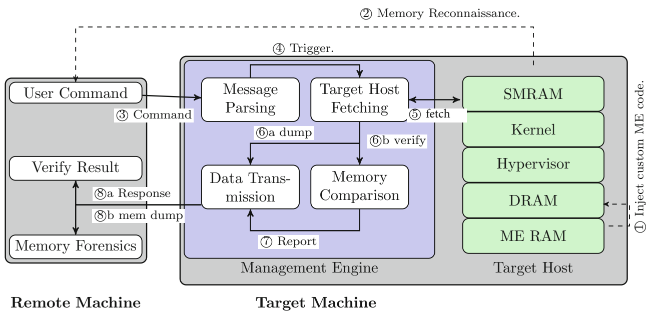

Fig. 2. High-level overview of the implementation. Following the numbered arrows, we (1) inject custom code into the IME on the Target Machine, and (2) acquire physical addresses of critical data structures. Next, the user (3) issues commands to the Introspection Server, which (4) triggers the corresponding command. (5) the IME uses μDMA and a modified HECI channel to fetch the target data from the Target Host and SMM memory. Depending on the command, the resulting memory is either (6a) dumped to the Remote Machine or (6b) checked locally for integrity. If applicable, (7) the integrity is checked with respect to a clean version of memory. Finally, the result is (8) transmitted back to the Remote Machine.

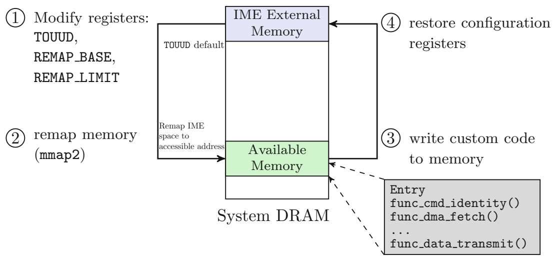
① Modify registers: TOUUD, REMAP_BASE, REMAP_LIMIT
② remap memory (mmap2)
TOUUD default
Remap IME space to accessible address
IME External Memory
Available Memory
System DRAM
④ restore configuration registers
③ write custom code to memory
Entry
func_cmd_identity()
func_dma_fetch()
...
func_data_transmit()

Fig. 3. Custom IME code injection. First, we configure system registers (`TOUUD`, top of upper usable memory, `REMAP_BASE`, and `REMAP_LIMIT` in step 1) to map the IME external memory to a userspace-accessible region of memory (step 2). We write custom instructions to that region (step 3), then restore the configuration registers (step 4).

5.1 Preparing the Target Machine

The Intel Management Engine is a secret system developed by Intel, one except vendors expanding IME functions should be a hard challenge. With several previous ME related research works [12,31,40], we adopt the memory-remapping approach taken by Tereshkin and Wojtczuk [40]: essentially, the external IME RAM is configured to be accessible by the Target Host by configuring several system registers that influence memory mapping. The workflow is shown in Fig. 3. In practice, developers can work with vendors to deploy custom IME code that does not require such a workaround. SMM can be protected in a similar way.

Since we directly get the runtime IME memory data but not open-source of IME code, we first reverse engineer the ME code with assembly instruction set (chipset-dependent, and ARCompact [39] in testbed). Next, we trigger the remote command to run related thread in the target machine's IME, we then debug each corresponding functions and analyze the branch instructions (i.e., *bl*) to address the suitable functions and positions for introspection. Finally, we insert introspection code while maintaining the original functions.

However, with kernel-level access, it is possible to reuse those memory control registers to remap and subsequently alter the IME-reserved memory region and SMRAM. This could potentially allow attackers to compromise the Nighthawk. To close the injection vector after we insert the introspection code into the IME and SMRAM, we implement a lock mechanism on those memory control register by leveraging Intel TXT [18] with the follow operations.

1. We pre-install Trusted Boot [41] (TBoot), a booting module based on Intel TXT Technology to perform a measured and verified launch of an OS kernel/VMM. We can configure the TBoot to lock the memory control registers.
2. We configure the bootloader to use TBoot to boot the Linux kernel, then restart the target machine from remote server with an IME based remotely reboot instruction.

After rebooting, the custom IME and SMM code remains intact because booting into TXT mode prevents memory control registers from being modified.

5.2 Target Host Reconnaissance

In this subsection, we describe challenges associated with verifying the integrity of kernel, hypervisor, and SMM code and data, the solutions we chose, and how these solutions mitigate certain attacks.

Static Kernel Integrity Checking. The static kernel segments include both OS and Hypervisor kernel code and data. Typically, kernel code and several key data structures such as the system call table and the interrupt descriptor table do not change during runtime, but attackers might modify these structures, violating the kernel's integrity. To monitor kernel integrity, we use the system symbol table like `System.map` to gather crucial virtual addresses. `System.map` is a map from kernel symbols to virtual addresses. We can then obtain that symbol's physical address, which resides at a fixed offset away from its virtual address.[3] We use this approach to find physical addresses of several critical structures, including the system call table, the interrupt descriptor table, the kernel code and data segments, and (when applicable) hypervisor modules.

SMM Integrity Checking. Unlike the kernel or the hypervisor, accessing SMM memory is less straightforward. SMM code is stored in and executes from the System Management RAM (SMRAM), which is an isolated address space. This isolation feature can be locked or unlocked through configuring special register in the BIOS to protect access after booting. If SMRAM is unlocked, we can measure the integrity directly via the μDMA channel. However, even if SMRAM

[3] While this offset can be system-dependent, in most Linux setups, kernel virtual addresses are `0xc0000000` bytes from the corresponding physical address.

is locked, we implement a secure communication channel between the IME and SMM. Since HECI is an unique interface designed to communicate between the IME and host, we reuse the related HECI registers to create a channel between the IME and SMM. Atop this channel, we add code to check the integrity of both SMM-related code and register values. We can communicate this information from SMM over the HECI channel, at which point we can verify results within the IME. This approach enables transparent and rapid evaluation of SMM code and data even when the target machine is compromised.

Mitigating Attacks. Nighthawk is co-processor based approach that suffers from the address translation redirection attack (ATRA) [19] and transient attacks [27,48]. However, Nighthawk is able to detect these attacks. For ATRA attacks, first, we store a clean copy of kernel page table by accessing the symbol `swapper_pg_dir` at kernel initialization stage. Second, we obtain the CR3 register value by leveraging SMM (SMRAM is protected by SMM integrity checking). Thus, the binding between the virtual and physical memory addresses can be verified in the IME subsystem. For transient attacks, Nighthawk works in an independently environment with little introspecting trace. Compared to the SMM-based monitoring approaches like HyperCheck [48] and HyperSentry [5], introspection interval of Nighthawk becomes more harder to be gleaned by attackers. Moreover, the code in the IME can run *continuously* without halting the target OS, and thus attackers cannot predict when a memory page will be checked.

5.3 Measuring Integrity via Custom IME

Next, we discuss the introspection workflow in Nighthawk. As the IME is intended for remote administration, it contains basic networking code. We reverse-engineered our IME firmware to find these networking functions that could be reused by our injected IME code. The injected code is composed with list of introspected object structures and checking functions. Essentially, we modified the IME code to perform introspection activities in response to requests sent from the Introspection Client on the remote machine. The workflow consists of four steps, shown in Fig. 4.

1. When the target machine receives a network command, it is received by the remote machine in the `recv_cmd()` function. Then, `msg_parse()` determines which integrity checking operation it needs to perform.
2. Next, we fetch the specified target data. We use a μDMA channel between the Target Host and the IME to fetch the specified data from memory. If the target data is from locked SMRAM, `data_fetch()` creates the HECI channel between the IME and SMM.
3. After fetching, Nighthawk compares the hash value of the fetched memory with the original version established during boot in the IME system.
4. After comparison, `data_transmit()` transfer the results to the remote machine for continue analyzing.

Next, we discuss key aspects of the introspection workflow.

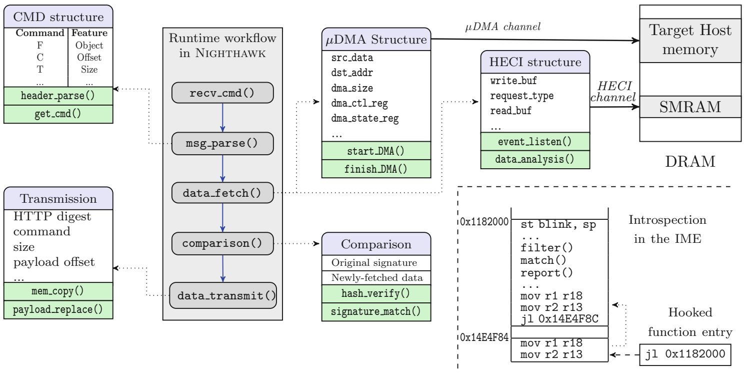

Fig. 4. The introspection workflow in the IME. We reverse-engineered the locations of several network-related functions in the existing IME code on our Target Machine. We added code to include custom commands to support our main goal of checking the integrity of the Target Host.

μDMA based Memory Fetching. NIGHTHAWK uses the μDMA engine to access the Target Host's physical memory from the IME. Our prototype's chipset [17] supports configuration of four μDMA channels (i.e., we can have four memory requests in-flight simultaneously). We use a number of auxiliary registers to control the size, direction, and other properties of the μDMA request. First, we write certain structures (e.g., the source and destination addresses) to auxiliary registers so as to engage the μDMA engine to automatically retrieve portions of the Target Host's physical memory. Then, the μDMA engine automatically stores the requested memory content in an IME-designated location. Once the function has acquired the specified amount of data, the μDMA request stops. Note that, in some special case like ATRA attack defense, we get the CR3 value first by leveraging SMM, and then fetch the corresponding memory page.

Checking Runtime SMRAM. For unlocked SMRAM, we directly access the memory through μDMA and check the integrity in the IME. For locked SMRAM, NIGHTHAWK introspects the SMRAM through the cooperative HECI channel. In the IME, we add a static SMRAM configuration during the initialization stage, which includes the SMM code and the original value for each SMM register (e.g., SMBASE – `0xa0000`). In SMM, we add two main functions: First, we use the SDBM hash algorithm [29] to calculate the integrity of SMRAM code, and we check the values of SMM-related registers at runtime. This helps us defend against attacks that attempt to change the SMM configuration or otherwise alter SMRAM. Second, we establish a communication channel between the IME and SMM by configuring a number of HECI Host registers: `H_CBRW` (Host Circular Buffer Read Window), `H_IG` (Host Interrupt Generate), and `H_CSR` (Host Control Status). In particular, writing to `H_IG` generates an interrupt to the IME. This HECI-based communication channel can pass data from SMM to the IME to check SMM code and data.

5.4 Remote Machine

We discuss how the Introspection Client interacts with the Target Host. There are two main functions implemented on the remote machine: information collection and transparent introspection of Target Host. The remote machine initiates a request over the network to begin introspecting the Target Host. Once the Target Host is initialized, a communication channel is established to collect memory address information, including symbol names, addresses, sizes, etc. from the target machine. The collected information is transmitted to the remote machine for later use. After this initiation, the introspection session can begin. The remote machine interacts with the target machine in three scenarios:

- First, system administrators set the IME username and password for secure login. The remote machine supplies credentials for user authentication to create a secure channel with target machine.
- Second, remote machine sends the introspection command following the developed small custom protocol, shown in Table 6 in Appendix. Moreover, the communication is encrypted via a session key established at runtime.
- Third, the remote machine receives responses to commands from the target machine. There are two types of response: integrity verification and forensic analyses. Integrity verification is processed in the IME system, thus the response would be a Boolean result indicating whether the integrity was violated. Memory forensic analyses are offloaded to the remote machine, so the response contains a large memory dump.

6 Evaluation

Our experimental environment consists of two physical machines: the target machine, with a 3.0 GHz Intel E8400 CPU, ICH9D0 I/O Controller Hub, and 2 GB RAM. An Intel e1000e Gigabit network card is integrated in the Intel DQ35JO motherboard. The BIOS version is JOQ3510J.86A.0933. For kernel integrity testing, the target machine runs Ubuntu with Linux kernel versions 2.6.x to 4.x. For hypervisor integrity testing, both Xen 4.4 and KVM 2.0 are used. The remote machine runs Microsoft Windows 10 with WireShark [7] installed for network packet monitoring. In this section, we evaluate Nighthawk from two aspects: *effectiveness* (i.e., does our system detect the presence of real-world threats?) and *efficiency* (i.e., does our system incur a low overhead?).

6.1 Effectiveness

We measure effectiveness by introspecting the Linux kernel, hypervisor, and SMM, as well as detecting ATRA and transient attacks.

Kernel Integrity Verification. We consider 5 real-world kernel rootkits, shown in Table 1, which fall into two categories:

Table 1. The effectiveness of NIGHTHAWK introspection.

Type	Attacked object	Attacks [1,2,9]	Detected
OS kernel	system call table	benign	✗
	kernel _text	*pusezk*	✓
	kernel _data	*Diamorphine*	✓
	IDT_table	*kbeast*	✓
	page directory entry	*amark*	✓
	page table entry	*adore-ng*	✓
		manual modification	✓
Hypervisor	kvm.ko	benign	✗
	kvm_intel.ko	*pusezk*	✓
	Xen kernel _text	*Diamorphine*	✓
	_stext _etext	*kbeast*	✓
	hypercall_page	*amark*	✓
	IDT_table	*adore-ng*	✓
	page directory entry	manual modification	✓
	page table entry		
SMM	SMRAM	benign	✗
		SMM reloaded	✓
		manual modification	✓

– System call table modification. Rootkits with kernel-level privilege can write to this table by manipulating the control register CR0. 4 of our 5 rootkits belong to this category: **Pusezk**, **Diamorphine**, **amark**, and **Kbeast** [2].
– Function pointer modification. For this category, we choose **adore-ng** [1]. **adore-ng** hooks the virtual file system interface to subvert normal detection. For example, to hide a malicious process, it redirects the iterate pointer in a kernel data structure `proc_root_operations` so that the malicious process will not be displayed in the `/proc` file system.

In addition to these real-world kernel rootkits, we also manually and randomly modify kernel memory pages in the kernel text and data segments.

Hypervisor Integrity Verification. In addition to installing our 5 rootkits in a Xen system, we also emulate hypervisor attacks in two ways. First, we modify the IDT, hypercall, and exception tables in a Xen system to represent a compromised Xen hypervisor. Second, we manually modify bytes in system memory of a KVM guest. In particular, we identify base addresses of KVM modules (`kvm.ko` and `kvm-intel.ko`), then randomly modify 5 bytes in these regions. These two approaches allow us to simulate an attacker that compromises the integrity of a Xen or KVM hypervisor.

SMM Integrity Verification. To demonstrate SMM integrity verification, we employ existing SMM attacks (i.e., the SMM Reload program [9]) to maliciously modify the SMI handler. We statically identify the `RSM` instruction that ends the SMI handler, and insert malicious instructions (e.g., `mov` $x, %addr) to simulate an attack that can modify arbitrary memory addresses. To detect these attacks, we verify memory pages in SMRAM (see Sect. 5.2 for details on acquiring this memory). We then compare their runtime states with their clean states, and we consider any discrepancy as an integrity violation. We can thus detect the existing and simulated SMM attacks described above.

ATRA Detection. We keep a clean copy of the kernel page table at system initialization stage through searching the *swapper_pg_dir* symbol. We use the CR3 value (acquired relying on SMM) to search for the corresponding physical page directory entry and page table entry via physical memory. In addition, we test the experiment when Page Global Directory and CR3 changed under Kernel Page Table Isolation (KPTI) mechanism and IDT based attack [19]. Finally, we compare the search data to determine if a change has been made. Our comparison results show that Nighthawk can detect the trace of ATRA.

Transient Attack Detection. To detect transient attacks, we continuously scan kernel pages in the IME system. We install a rootkit based on toorkit [15], the rootkit is able to timing change the pointer address of the system call table which leads to attacker-controller system calls. The rootkit emulates a transient attack by quickly invoking `insmod` and `rmmod` in the Linux OS. We also modify the code to parameterize the attack time (i.e., the time elapsed between `insmod` and `rmmod`). We sweep the attack time from 3 ms to 700 ms, and run each configuration 20 times. Our results in Table 2 show that Nighthawk can detect transient attacks if the attacking time is more than 700 ms. However, if the attacking time is less than 400 ms, the detection rate decreases linearly because Nighthawk requires a certain amount of execution time. That said, our approach can detect many real transient system attacks [24,43], which remain in memory for seconds at a time. While attacks such as bus snooping [23] are fast enough to evade detection, they require physical access to the machine and are thus out of scope.

Table 1 shows our experimental results for kernel-, hypervisor-, and SMM-level attacks. The results indicate that the rootkits as well as our manual modification are detected by Nighthawk. This demonstrates that Nighthawk is effective in monitoring the integrity of the OS kernel, the hypervisor, and SMM code. In addition, our experimental results also show that Nighthawk detects ATRA and transient attacks.

6.2 Efficiency

The efficiency of Nighthawk is mainly determined by the time cost of three logical operations: (1) data fetching, (2) integrity checking, and (3) data transmission. We measure the time consumed by each operation. For data fetching, we also measure its memory overhead, so that we can ascertain that Nighthawk does not have noticeable impact on the target system.

Table 2. Transient attack detection.

Execution time (ms)	Attacks detected rate
< 8	<2.5%
12	7.5%
63	8.3%
123	22.5%
218	33.3%
437	68.3%
515	81.4%
643	92.1%
>700	100%

Table 3. Time consumed by DMA.

Object	Size (KB)	Time (s)
(General data)	1	0.258 ± 0.010
	4	0.261 ± 0.010
	64	0.267 ± 0.010
	256	0.387 ± 0.120
	2,048	3.06 ± 0.350
	3,096	4.67 ± 0.430
System call table	4	0.261 ± 0.010
Linux kernel	6,466	9.75 ± 1.300
Hypervisor	336	1.31 ± 0.130
IDT	1	0.258 ± 0.010
Swapper_pg_dir	4	0.263 ± 0.010
SMRAM (unlocked)	128	0.383 ± 0.120
Random	10,240	15.4 ± 3.920

6.2.1 DMA Fetching Overhead

We first measure the DMA data fetching operation. Regardless of whether introspection is performed on the IME or on the remote machine, each Target Host memory segment must first be fetched into the IME space via μDMA.

Table 3 illustrates the time consumed by using DMA fetching. When the size of DMA-transmitted memory is smaller than 64 KB, the time consumed is approximately 0.26 s. This is due to the DMA channel using 16 lines to access the DRAM in parallel, allowing 2^{16} bytes of data each time. When the size is larger than 64 KB, the time consumed is linear to the amount of DMA operations. To improve the DMA effectiveness, we enable 4 μDMA channels to parallelly fetch at most 256 KB target physical memory one time. The bottom half of Table 3 shows the time consumed retrieving specific segments.

Figure 5 shows the wall clock time to perform different memory dump fetching. While `system_call_table`, PDE, and PTE pages are all 4 KB blocks of memory, the overhead is lower for `system_call_table` analysis because it requires only one fixed-address request. In contrast, the overhead is higher for page table analysis because acquiring page tables requires resolving additional indirection (i.e., fetching CR3 and separate requests to follow PDEs and PTEs) which needs multiple μDMA operations. In our tests, we found that it took 0.815 s to fetch the PTE entries and 1.28 s to verify the page table.

Since the DMA operations from the IME and the Target Host share the same RAM, concurrent RAM accesses are inevitable in our system. During DMA transfer, the CPU is idle and has no control of the memory buses. We use the STREAM benchmark [26] to measure the performance degradation imposed on the target machine. We use the `memcpy` function in an infinite loop to keep the DMA fetching operation running. Figure 6 shows there are minimum differences in memory bandwidth with and without NIGHTHAWK introspection: most of the

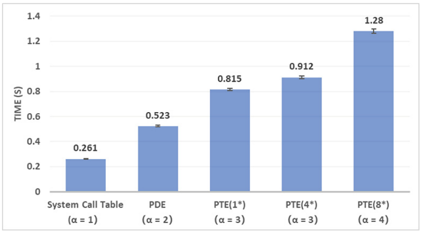

Fig. 5. Time consumed by fetching data. * represents the number of PTEs. α represents accessing times.

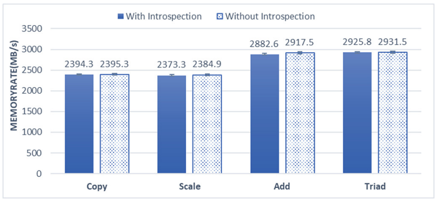

Fig. 6. Memory throughput degradation due to introspection.

time, the performance degradation is less than 0.2%, and even in the worst case (i.e., in the *Add* function test), the degradation is only 1.47%.

6.2.2 Integrity Checking Overhead

The second operation we measure is integrity checking. For each memory segment in question, we compute a hash value, and compare it with a pre-computed value supplied by the remote machine representing the clean state. Therefore, the time cost depends on the hash algorithm we choose. Recall for simplicity we chose to implement SDBM hashing [29]. Our test result shows that, to compute a hash value for a 4 KB memory page, the algorithm takes 7.3 ms. To verify the page table address, we simply compare each entry item in the table by value.

We only check the kernel page table, and at most 257 4 KB-size pages we need to compare—however, in practice about 10 pages suffice. Thus, compared to the fetching stage, the overhead for comparison is much lower—less than 2 ms each time.

6.2.3 Transmission Overhead

The third operation we measure is data transmission. In general, we send an introspection command from the remote machine and receive the verification result. We use one small message to pass the data ($< 1\,KB$), taking 228 ms on average. When considering a memory dump (i.e., $> 64\,KB$) to the remote machine, we divide the data into multiple packets and transmit them into multiple messages. We find that transmitting 64 KB data takes 4.9 s and that this duration grows linearly with the transmit size.

6.2.4 Efficiency Evaluation Summary

Overall, a typical introspection cycle contains the above three logical operations. Table 4 summarizes the time spent in each operation and in total. For instance, the system call table or the SMRAM (unlocked[4]), the introspection takes less than 1.5 s to acquire the integrity status.

[4] Even when SMRAM is locked, using our HECI-based communication channel, we incur roughly 17 ms to perform end-to-end integrity checking.

Table 4. The performance of the complete introspection about NIGHTHAWK.

Object	Size (KB)	Data fetching time (s)	Comparison time (s)	Data transmission time (s)	Total time (s)
System call table	4	0.26 ± 0.010	0.007 ± 0.001	0.224 ± 0.030	0.50 ± 0.030
kvm_intel.ko	336	1.31 ± 0.130	0.601 ± 0.010	0.231 ± 0.030	2.14 ± 0.150
PDE	4	0.52 ± 0.010	0.007 ± 0.001	0.230 ± 0.030	0.76 ± 0.040
SMRAM (unlocked)	128	0.39 ± 0.150	0.320 ± 0.005	0.228 ± 0.030	0.94 ± 0.200

7 Related Work

In this section, we survey the related work. Our research is mainly related to two categories of work: trusted execution environments, Intel ME.

Trusted Execution Environment. Trusted execution environments (TEEs) are intended to provide a safe haven for programs to execute sensitive tasks. We can use software- or hardware-based approaches to create TEEs.

Typically, software-based approaches leverage virtualization. Terra [14] runs applications with diverse security requirements in different virtual machines managed by a trusted Virtual Machine Monitor so that compromised applications do not interfere with others. Some hypervisor-based introspection approaches like SecVisor [34] can also provide a small TCB, but still incurs significant overhead, whereas NIGHTHAWK does not. In contrast, hardware-based approaches rely on different hardware features. KI-Mon [23] is a hardware-based DMA module and hash accelerator on the external SoC component used as an event-triggered kernel integrity monitor. GRIM [22] uses GPUs to check the kernel's integrity at high speed. TZ-RKP [4] is the representative work using ARM TrustZone to construct a TEE for OS kernel protection. HyperCheck [48] and HyperSentry [5] both employs Intel SMM to build a TEE and monitor hypervisor integrity. Chevalier *et al.* [6] proposed using a co-processor to monitor SMM code behavior, but it requires modifying the SMM code for instrumentation which is implemented with QEMU and simulation. In this paper, we build our TEE using the IME, and use it to monitor the host system.

Works on Intel ME. By design [33], the IME has full access to the system's memory, peripheral devices, and networks. Because of this high privilege, the IME has attracted attention from security researchers [35,36,38]. For example, to analyze the code in the IME, Sklyarov [35] proposed an SPI-based approach to fetch the IME firmware from the storage flash chip. In other work, Sklyarov [36] presented a static analysis approach in which he was able to distinguish the different functions in the IME via matching the signature of each code module. In addition, security vulnerabilities in the IME were also discovered [12,40]. Tereshkin *et al.* [40] proposed a memory remapping approach which enables the host CPU to access the IME memory. Ermolov *et al.* [12] revealed multiple buffer overflow vulnerabilities in the IME, which allows local users to perform a privilege-escalation attack and run arbitrary code. Due to the powerful but uncontrolled function in IME, some researchers [8,10,11,31] tried to disable the IME or confine its ability to interact with the host system, yet do not cause any disruption to the normal operation in the host system. In this paper, we demonstrate that defenders can leverage IME to introspect the host system.

8 Discussion

Security Issues: In our prototype, we implement Nighthawk via code injection into the IME. It is possible to be compromised by new attacks despite mitigating the interface for code injection. The security arms race will persist, however the IME has a reasonably small TCB. Nighthawk is able to defense the SMM attacks which intend to access the locked SMRAM by reconfiguring the SMM related registers. However, if the SMM code can be manipulated directly by attackers, SMM based functions like CR3 reading operation may not be trusted but we can defense it by integrating the work [6].

Other Kernel-Level Attacks: In our evaluation, if an attacker operates faster than the checking time required by Nighthawk, we may not be able to detect it. To reduce the risk of transient attacks, we can reduce the integrity checking time. There are several optimizations we could make with additional engineering effort. Other kernel-level attacks like Direct Kernel Object Manipulation (DKOM) can also be detected by Nighthawk by using similar approaches like Perkins *et al.* [30] with additional effort.

DMA Access: The introspection workflow in Nighthawk leverages μDMA to fetch host memory. If the μDMA channel from the IME is blocked (e.g., by I/OMMU [3]), it will prevent Nighthawk from reading the Target Host memory. Fortunately, I/OMMU can be configured to allow this access in the BIOS. Moreover, Nighthawk is able to check the I/OMMU configuration similar to IOCheck [47]. Note that the IME accessing reserved 16MB memory at the top of DRAM does not go through the Intel VT-d remapping (i.e., I/OMMU implementation of Intel) [17], thus, I/OMMU cannot block IME from accessing its inner memory.

Performance: The performance of Nighthawk heavily depends on the hardware design of the IME. In this paper, our testbed's IME suffered from low performance (Sect. 6.2) mainly due to a slow ME processor speed. However, this situation can be improved with a powerful chipset [12]. In addition, we reverse engineered our testbed's IME to inject code. This approach may not have resulted in the best performance (i.e., there may have been a higher-performance method of customizing IME code).

9 Conclusions

In this paper, we presented Nighthawk, a transparent introspection framework for verifying the memory integrity of a Target Machine. It leverages Intel ME, an existing co-processor running aside with the main CPU with ring -3 privilege, so that our approach has a minimal TCB, is capable to detect low-level system software attacks, and introduces minimal overhead. To demonstrate the effectiveness of our system, we implemented a prototype of NIGHTHAWK with two physical machines. The experimental results show that NIGHTHAWK is able to detect real-world attacks against OS kernels, Xen- and KVM-based hypervisors, and System Management RAM. The experimental results show NIGHTHAWK verifies the integrity of target host system with a low performance overhead.

Acknowledgments. Lei Zhou was supported by the China Scholarship Council at Wayne State University. This work is supported in part by the National Natural Science Foundation of China under Grant Number 61632009, the Guangdong Provincial Natural Science Foundation under Grant Number 2017A030308006.

A Appendix: Intel ME

An overview of system components and the IME is shown in Fig. 7.

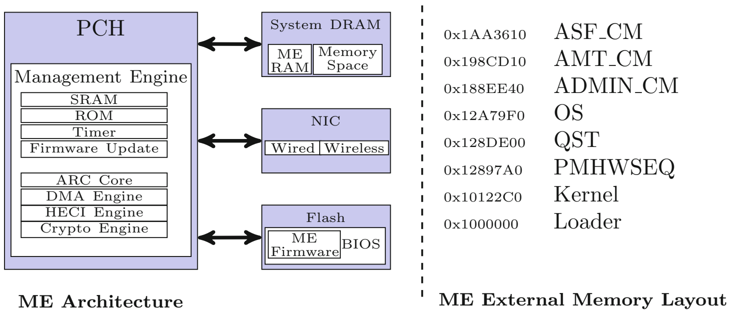

Fig. 7. Overview of the IME. We use its isolation features to provide transparent system introspection capabilities. The left shows the IME in relation to other parts of a host system. The right shows the IME's memory layout on our prototype. Adapted from Ruan [33]

B Appendix: Code added in Intel IME

Properties of our custom IME added code are shown in Table 5. All told, we wrote 400 lines of new C code and 270 lines of new assembly code, all of which fit in an IME firmware image less than 2 KB in size.

C Appendix: Remote Communication Protocol

Here we present the details about remote communication protocol between remote server and IME in target machine.

Table 5. Introspection code added in custom IME firmware

Code section	Language	Size (# lines)
DMA fetching	C	210
Integrity checking	C	70
Introspection Server	C	120
IME injection	ASM	270

Table 6. Communication commands in NIGHTHAWK, each consisting of an operation and corresponding object. Any Command can be combined with any Object.

Command	Description	Object	Description
F	Fetch the physical memory from Target Host to the IME	SCT	The information about System Call Table
C	Compare the Target Host memory in the IME system	LK	The information about Linux Kernel
T	Transmit the introspection results from the IME to Remote Machine	HYP	The information about Hypervisor
D	Dump the Target Host memory from the IME to Remote Machine	SMM	The information about SMRAM

D Appendix: Performance of the IME Core

We run experiments to investigate the computational capabilities of the IME. In particular, we develop a CPU speed testing benchmark, which we inject into the `memcpy` function in the IME. That is, this benchmark executes every time `memcpy` is invoked. The testing program is a nested-loop (inner loop: n, outer loop: m) function with 15 instructions in the inner loop such that $n \times m = 10^6$. We read the time stamp counter at the beginning and the end of the benchmark—denoted as T_1 and T_2, and thus approximate the average speed of the IME CPU using the formula $v \approx \frac{15 \times 10^6 \times (n \times m)}{(T_2 - T_1)}$. We sweep $n = 100, 200, ..., 10000$ and $m = 100, 200, 1000$; the experimental result shows that the IME CPU executes approximately 15 million instructions each second. Compared to the target system's main CPU (which can execute billions of instructions per second), the IME CPU has a significantly lower performance.

References

1. Adore-ng (2018). https://github.com/trimpsyw/adore-ng/
2. RootKits List (2018). https://github.com/d30sa1/RootKits-List-Download
3. Abramson, D., et al.: Intel virtualization technology for directed I/O. Intel Technol. J. **10**(3), 179–192 (2006)

4. Azab, A.M., et al.: Hypervision across worlds: real-time kernel protection from the arm trustzone secure world. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS) (2014)
5. Azab, A.M., Ning, P., Wang, Z., Jiang, X., Zhang, X., Skalsky, N.C.: HyperSentry: enabling stealthy in-context measurement of hypervisor integrity. In: Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS) (2010)
6. Chevalier, R., Villatel, M., Plaquin, D., Hiet, G.: Co-processor-based behavior monitoring: application to the detection of attacks against the system management mode. In: Proceedings of the 33rd Annual Computer Security Applications Conference (2017)
7. Combs, G.: Wireshark (2019). https://www.wireshark.org
8. Corna, N.: ME cleaner: tool for partial deblobbing of Intel ME/TXE firmware images (2017). https://github.com/corna/me_cleaner
9. Duflot, L., Levillain, O., Morin, B., Grumelard, O.: Getting into the SMRAM: SMM Reloaded. CanSecWest (2009)
10. Erica, P., Peter, E.: Intel's Management Engine is a security hazard, and users need a way to disable it (2017). https://www.eff.org/deeplinks/2017/05/intels-management-engine-security-hazard-and-users-need-way-disable-it
11. Ermolov, M., Goryachy, M.: Disabling Intel ME 11 via undocumented mode (2017). http://blog.ptsecurity.com/2017/08/disabling-intel-me.html
12. Ermolov, M., Goryachy, M.: How to Hack a Turned-Off Computer, or Running Unsigned Code in Intel Management Engine. Black Hat Europe (2017)
13. Gael, H.I.: Intel AMT and the Intel ME (2009). https://intel.com/en-us/blogs/2011/12/14/intelr-amt-and-the-intelr-me
14. Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., Boneh, D.: Terra: a virtual machine-based platform for trusted computing. In: ACM SIGOPS Operating Systems Review (2003)
15. Github: ToorKit (2015). https://github.com/deb0ch/toorkit
16. Intel: Innovation Engine (2015). https://en.wikichip.org/wiki/intel/innovation_engine
17. Intel Corporation: Intel 3 Series Express Chipset Family (2007). https://www.intel.com/Assets/PDF/datasheet/316966.pdf
18. Intel Corporation: Intel Trusted Execution Technology (Intel TXT): Software Development Guide (2017). https://www.intel.com/content/dam/www/public/us/en/documents/guides/intel-txt-software-development-guide.pdf
19. Jang, D., Lee, H., Kim, M., Kim, D., et al.: Atra: address translation redirection attack against hardware-based external monitors. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (2014)
20. Jiang, X., Wang, X., Xu, D.: Stealthy malware detection through VMM-based out-of-the-box semantic view reconstruction. In: Proceedings of the 14th ACM conference on Computer and Communications Security (CCS) (2007)
21. Jones, S.T., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: VMM-based hidden process detection and identification using Lycosid. In: Proceedings of the fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE) (2008)
22. Koromilas, L., Vasiliadis, G., Athanasopoulos, E., Ioannidis, S.: GRIM: leveraging GPUs for kernel integrity monitoring. In: Monrose, F., Dacier, M., Blanc, G., Garcia-Alfaro, J. (eds.) RAID 2016. LNCS, vol. 9854, pp. 3–23. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45719-2_1

23. Lee, H., et al.: KI-Mon: a hardware-assisted event-triggered monitoring platform for mutable kernel object. In: USENIX Security Symposium (2013)
24. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., et al.: Meltdown: reading kernel memory from user space. In: Proceedings of the 27th Conference on USENIX Security Symposium (2018)
25. Malka, M., Amit, N., Ben-Yehuda, M., Tsafrir, D.: rIOMMU: efficient IOMMU for I/O devices that employ ring buffers. In: ACM SIGPLAN Notices (2015)
26. McCalpin, J.D.: STREAM (2018). http://www.cs.virginia.edu/stream/ref.html
27. Moon, H., Lee, H., Lee, J., Kim, K., Paek, Y., Kang, B.B.: Vigilare: toward snoop-based kernel integrity monitor. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS) (2012)
28. National Institute of Standards, NIST: National Vulnerability Database (2018). http://nvd.nist.gov
29. Partow, A.: General Purpose Hash Function Algorithms (2018). http://www.partow.net/programming/hashfunctions
30. Perkins, J.H., et al.: Automatically patching errors in deployed software. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (2009)
31. Persmule: Neutralize ME firmware on SandyBridge and IvyBridge platforms (2016). https://hardenedlinux.github.io/firmware/2016/11/17/neutralize_ME_firmware_on_sandybridge_and_ivybridge.html
32. Petroni Jr, N.L., Fraser, T., Molina, J., Arbaugh, W.A.: Copilot-a Coprocessor-based Kernel Runtime Integrity Monitor. In: USENIX Security Symposium (2004)
33. Ruan, X.: Platform Embedded Security Technology Revealed: Safeguarding the Future of Computing with Intel Embedded Security and Management Engine. Apress (2014)
34. Seshadri, A., Luk, M., Qu, N., Perrig, A.: SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In: Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP) (2007)
35. Sklyarov, D.: Intel ME: flash file system explained. Black Hat Europe (2017)
36. Sklyarov, D.O.: ME: The Way of the Static Analysis. TROOPERS17 (2017)
37. Spensky, C., Hu, H., Leach, K.: LO-PHI: low-observable physical host instrumentation for malware analysis. In: NDSS (2016)
38. Stewin, P., Bystrov, I.: Understanding DMA malware. In: Flegel, U., Markatos, E., Robertson, W. (eds.) DIMVA 2012. LNCS, vol. 7591, pp. 21–41. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37300-8_2
39. Synopsys: embARC (2019). https://embarc.org/embarc_osp/doc/build/html/arc/arc.html
40. Tereshkin, A., Wojtczuk, R.: Introducing ring-3 rootkits. Black Hat USA (2009)
41. The Fedora Project: TBoot (2018). https://sourceforge.net/projects/tboot
42. UPnP Forum: MeshCommander (2018). http://www.meshcommander.com/
43. Wei, J., Payne, B.D., Giffin, J., Pu, C.: Soft-timer driven transient kernel control flow attacks and defense. In: 2008 Annual Computer Security Applications Conference (ACSAC) (2008)
44. Wojtczuk, R., Rutkowska, J.: Attacking SMM memory via Intel CPU cache poisoning. Invisible Things Lab (2009)
45. Yao, J.: SMM Protection in EDK II (2017). https://uefi.org/sites/default/files/resources/Jiewen
46. Zhang, F., Leach, K., Stavrou, A., Wang, H., Sun, K.: Using hardware features for increased debugging transparency. In: 2015 IEEE Symposium on Security and Privacy (SP) (2015)

47. Zhang, F., Wang, H., Leach, K., Stavrou, A.: A framework to secure peripherals at runtime. In: Kutyłowski, M., Vaidya, J. (eds.) ESORICS 2014. LNCS, vol. 8712, pp. 219–238. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11203-9_13
48. Zhang, F., Wang, J., Sun, K., Stavrou, A.: Hypercheck: A hardware-assistedintegrity monitor (2014)