A Coprocessor-Based Introspection Framework Via Intel Management Engine

Lei Zhou, Fengwei Zhang, Jidong Xiao, Kevin Leach, Westley Weimer, Xuhua Ding, and Guojun Wang

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 article, we propose an introspection framework called NIGHTHAWK that transparently checks system integrity and monitor the runtime state of target system. 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 introspect 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. Additionally, NIGHTHAWK can monitor the runtime state of host system against the suspicious applications running in target machine.

Index Terms—Intel ME, system management mode, introspection, integrity, transparency

✦

1 INTRODUCTION

SECURITY vulnerabilities [1] 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 gains an advantage over the attacker by executing in a more privileged context. More specifically, to detect low-level malware, virtualization-based defensive approaches [3], [4] and hardware-assisted defensive approaches [5], [6], [7] 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 [8]. Second, and more importantly, mainstream hypervisors generally 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 [6], Vigilare [5], and LO-PHI [9], typically use a coprocessor (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 [10], HyperCheck [11] 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 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.

- *Lei Zhou and Fengwei Zhang are with the Department of Computer Science and Engineering, and Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen 518055, China. E-mail: {zhoul6, zhangfw}@sustech.edu.cn.*
- *Jidong Xiao is with the Boise State University, Boise, ID 83725 USA. E-mail: jidongxiao@boisestate.edu.*
- *Kevin Leach and Westley Weimer are with the University of Michigan, Ann Arbor, MI 48109 USA. E-mail: {kjleach, weimerw}@umich.edu.*
- *Xuhua Ding is with the Singapore Management University, Singapore 188065. E-mail: xhding@smu.edu.sg.*
- *Guojun Wang is with the Guangzhou University, Guangzhou 510405, China. E-mail: csgjwang@gzhu.edu.cn.*

To address these limitations in 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 use the IME to construct a system introspection framework capable of efficiently (1) checking the integrity of kernel and hypervisor structures and system firmware, and (2) monitoring the system state by quickly analyzing critical target memory data. To the best of our knowledge, this is the first paper to consider using the IME for system introspection. Our proposed framework offers the following advantages in comparison to previous work:

- *No extra hardware required.* The IME has been integrated into most current commercially-available Intel chipsets, which ensures that our proposed framework can be deployed without 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 is 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 entailing critical code integrity checking and host state monitoring. This paper is an extended version of our published conference work.[3] The existing work focuses on host integrity checking and experimental results show that NIGHTHAWK can detect real-world rootkits, including kernel-level rootkits and SMM rootkits. Following that work, we design a new system state monitoring mechanism. We analyze the target system to extract the representative features in a running OS, including runtime processes, physical memory usage, and information in *procfs*. We evaluated the added system monitoring modules, the additional results show that we can effectively introspect the host's state

information without interrupting its execution environment. Our main contributions are:

- We present NIGHTHAWK, a novel introspection framework that transparently checks the integrity of the host system and monitors the host state at runtime. We leverage the IME, 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. In addition, NIGHTHAWK can monitor target system state with forensics analyzing critical data in target OS. Furthermore, NIGHTHAWK is robust against page table manipulation attacks and transient attacks.
- NIGHTHAWK causes low latency to introspect the critical data structures. Our results show that NIGHTHAWK takes 0.502 seconds 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.

2 BACKGROUND

Intel Management Engine. The Intel Management Engine is a subsystem which includes a separate microprocessor, its own memory, and an isolated operating system [15]. 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 (IE) [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 Intel IE.

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

---

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], [13].

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

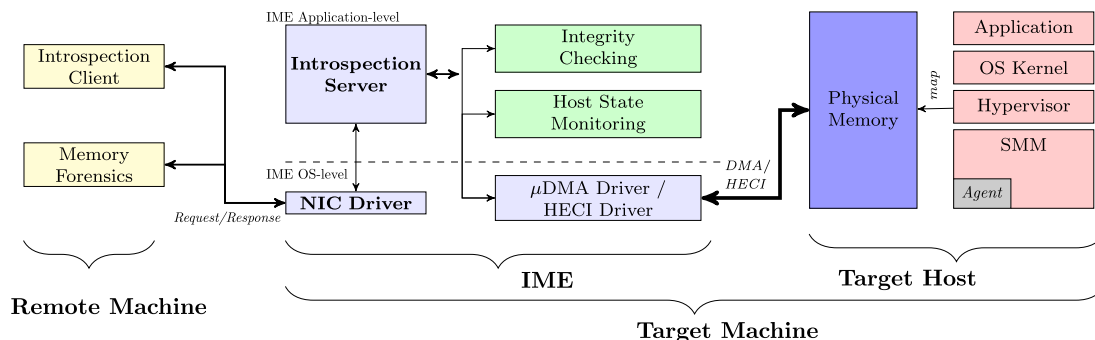3. NIGHTHAWK [14] has been published in ESORICS 2019.

Fig. 1. High level overview 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, Integrity Checking and Host State Monitoring Modules. When the user invokes an introspection command, the server dispatches the corresponding service 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.

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 [17] 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 [18] are out of scope because such attacks can be mitigated with official patch [19]. We assume the hardware can be trusted to function normally (e.g., hardware trojans are out of scope).

4 SYSTEM OVERVIEW

Fig. 1 illustrates the architecture of Nighthawk, where the Remote Machine and IME run in an trusted environment, and the Target Host runs in an untrusted environment. With the IME, Nighthawk transparently accesses the physical memory from the Target Host, and aims to reach two goals: First, we aim to transparently monitor the integrity of target system's memory (i.e., code and data) belonging to the application, kernel, any hypervisor present, and SMRAM. When an integrity violation is detected, our IME code asserts that an attack has occurred. Second, we aim to reconstruct the state of the runtime Target system. This can be effectively used to analyze the state of processes, resource usage (e.g., memory, cache, imports, interrupts) of runtime applications. With the Remote Sever, Nighthawk can remotely introspect and implement more powerful memory forensics module for application-level checking. We describe further details of each component below.

Target Machine. The Target Machine 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 two types of modules: integrity checking and state monitoring. The former focuses on verifying the objects of the kernel, hypervisor, and SMM; each object corresponds to a particular class of attack that may occur against the Target Machine. The latter is designed to monitor the behavior of the runtime system, from which abnormal state can be detected by analyzing the physical memory data.

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 checking and monitoring module is able to locate relevant data structures in the IME's external memory space and perform further introspection.

Remote Machine. The Remote Machine 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
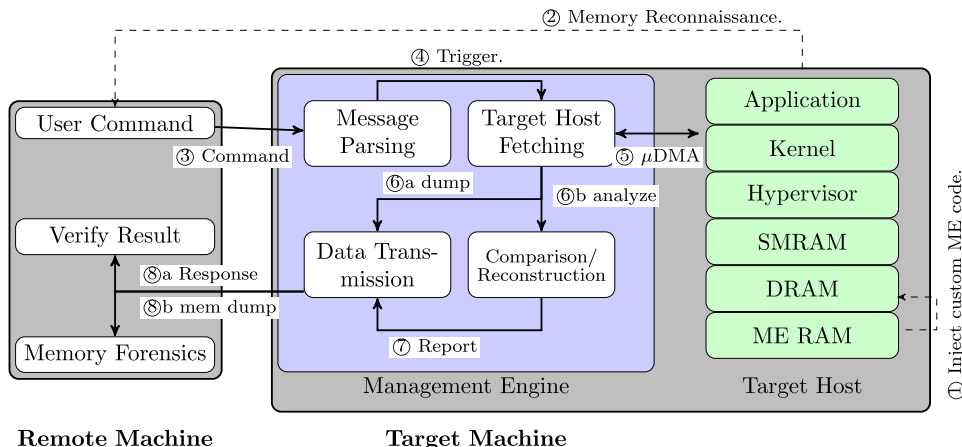
Fig. 2. High-level overview of the implementation. Following the numbered arrows, we (1) inject custom code into the IME on the Target Machine, and (2) acquire physical addresses of critical data structures. Next, the user (3) issues commands to the Introspection Server, which (4) triggers the corresponding 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) analyzed locally for integrity checking or state monitoring. If applicable, (7) the memory Comparison/Reconstruction handler produced the report of data analysis, and finally (8) transmitted back to the Remote Machine.

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 [20], which is a RESTful HTTPS protocol for remote management tasks. We reverse-engineered the protocol to augment it with custom commands used by our introspection code.

Summary. To summarize, we seek to check the integrity of target host using the IME, and further introspect the target runtime system. We use custom IME code to implement integrity checking for the Target Machine's kernel, hypervisor, and SMM code and data. Based on trusted code and data, we further implement host state monitoring for the Target Host's runtime system state. A user can interact with an Introspection Client to perform various introspection 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

We implemented a prototype of NIGHTHAWK based on the Intel x86 architecture. Fig. 2 describes implementation details pertinent to our prototype. 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.

5.1 Preparing the Target Machine

The Intel Management Engine is a system developed by Intel whose functions require significant engineering effort to expand. With several previous IME related research works [12], [13], we adopt the memory-remapping approach taken by Tereshkin and Wojtczuk [13]: essentially, the external IME RAM is made accessible by the Target Host by configuring several system registers that influence memory mapping. The workflow is shown in left parts of the 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 source code, we first reverse engineer the IME code—our prototype uses ARCompact [21]). Next, we can trigger remote commands to run related threads in the Target Machine's IME.

We can then acquire low-level runtime information (e.g., memory dumps) of running programs, which we can analyze (e.g., by seaching for branch instructions) to find addresses of suitable functions and positions for introspection. Finally, we insert introspection code while maintaining the original functionality (e.g., with trampolines).

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 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 [17], shown in the right side of Fig. 3.

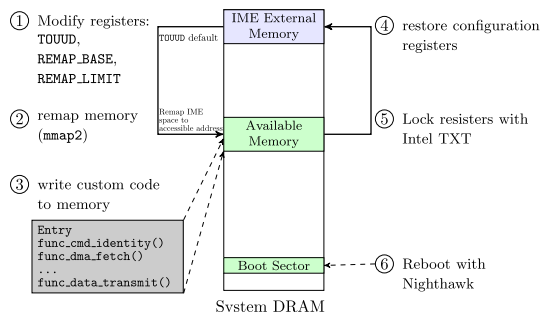1) We pre-install Trusted Boot [22] (TBoot), a booting module based on Intel TXT Technology to perform a

Fig. 3. Custom IME code injection and reusing blocked. 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). Last, we lock the registers at booting stage (step 5), and reboot the system with Nighthawk (step 6).

measured and verified launch of an OS kernel/ VMM. We can configure 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 the Remote Server with an IME-based remote reboot command.

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

5.2 Modules Designed in IME

In this subsection, we describe the design and implementation details of modules that we use as part of our prototype built on the Intel Management Engine.

IME is a tiny system in x86 chipsets with an independent CPU, memory, and other computing resources [23]. Function modules in Nighthawk are the extension of the original IME system. Through reverse engineering the runtime IME memory data (as discussed in Section 5.1), we locate each IME kernel function (e.g., *memcpy*, *memset*), network communication drivers (e.g., TCP/UDP packet processing functions) and other specific system modules. In the IME system, each device can be initialized directly by configuring auxiliary registers. Similarly, we can locate some key auxiliary registers in the IME core through some reverse engineering work (e.g., The physical address $0x5010$ to $0x5013$ for DMA channel registers, $0x6011]$ to $0x6016$ for timer registers). We configure the related auxiliary register to implement driver-like functions, which are used as a foundation for Nighthawk's capabilities.

Since the IME processor cannot address the host memory directly, two extended transfer engines — Direct Memory Access (DMA) and Host Embedded Controller Interface (HECI) — are used for data transmission between the IME memory and the Target Host memory. The DMA engine moves bulky data blocks between the IME memory and the Target Host memory. In our prototype environment, the IME adopts a tightly-coupled DMA engine called μDMA to achieve low latency and cycle-efficient DMA transfers. By configuring DMA-related registers (e.g., setting bits 2–4 of register [5010] with 010) to fetch physical memory data from the Target Host to the IME memory space, we

construct an effective DMA channel between the two. Furthermore, to securely access the Target Host's registers, we configure a HECI-like channel in SMM. Similar to the HECI driver, we configure related registers like H_CRS in SMM to implement byte-level data transferring.

In addition to the data transfer engine, modules for data encryption/decryption and analysis are designed as normal IME applications like AMT [15]. Such modules are inserted by hooking existing IME functions, like *memcpy* or *net_package_send*. The insertion operations change one instruction in the original function, and are resumed after the added module is executed to keep correct IME state. Since the IME system has limited resources (e.g., less than 3MB memory available for data processing), we add small functions and leverage existing functions to implement host introspection. For a complex introspection task, we divide them into pieces and migrate most of the work to the remote machine.

5.3 Target Host Reconnaissance

We describe challenges associated with: (1) verifying the integrity of its memory dumps about kernel, hypervisor, and SMM code and data; and (2) reconstructing the key data structures in host system. In addition, we consider the solutions we chose, and how these solutions mitigate certain attacks.

Static Kernel Integrity Checking. In a normal OS and hypervisor, the kernel code and data are in static memory segments, initialized during system boot. Typically, kernel code and several key data structures such as the system call table and the interrupt descriptor table do not change during runtime. However, attackers might modify these structures, violating the kernel's integrity. In general, its physical address can be found in the system symbol table (`System. map`) with a fixed offset change.[4] `System.map` is a map from kernel symbols to virtual addresses. To monitor kernel integrity, we similarly obtain that symbol's address from the system symbol table. 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 registers 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 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 Target 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

---

4. While this offset can be system-dependent, in most Linux setups, kernel virtual addresses are `0xc0000000` bytes from the corresponding physical address.
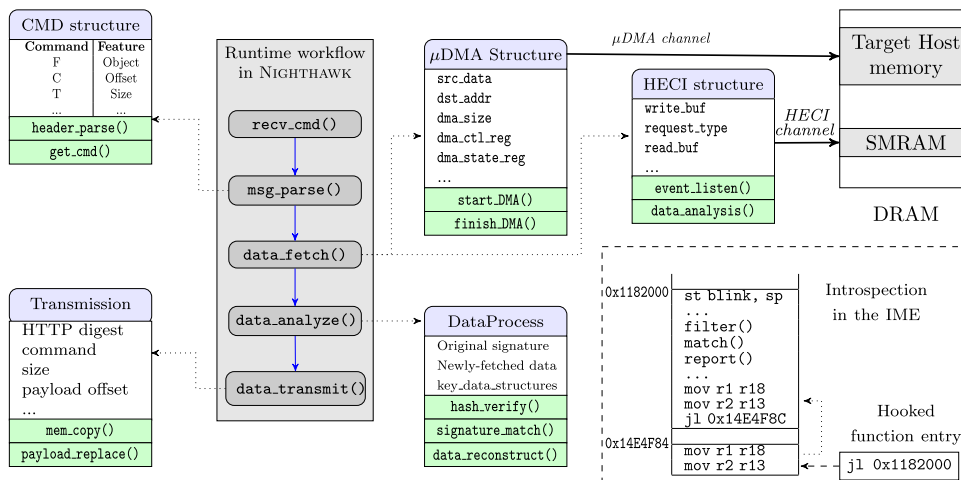
Fig. 4. The introspection workflow in the IME. We reverse-engineered the locations of several network-related functions in the existing IME code on our Target Machine. We added code to include custom commands to support our main goal of checking the integrity of the Target Host.

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.

Kernel Data Reconstruction. To introspect the state of the target system, NIGHTHAWK searches for critical system runtime information, similar to Virtual Machine Introspection [24]. Those information data is stored in instances of specific kernel objects, containing the state of processes, information of CPU interrupts, and statistical data from *procfs*, among other data. Since NIGHTHAWK must extract useful data from raw binary data, it generally relies on kernel data reconstruction to bridge the semantic gap. Kernel data reconstruction proceeds in two phases. First, we identify the symbols of each data structure and its storing address on physical memory. Second, we traverse all items in data structures to acquire valid data.

In the first phase, NIGHTHAWK relies on knowledge of the Target System's kernel version, compilation parameters, etc. With the same kernel source code and compiling conditions, NIGHTHAWK builds an analysis environment for the Target Host, then analyzes the source code to get the root address of data structure instances (and related pointers). For example, in Linux, each process has an associated *task_struct* object, and of whose instances are organized in a doubly- and circularly-linked list. The root element of this linked list is represented by the instance *init_task*, which we can find in the kernel symbol table. In some cases, the instances may be organized with different data structures (e.g., *proc_dir_entry* allocated with a Red-Black tree), which we support as needed.

In the second phase, with the root address and layout of data structure instances, NIGHTHAWK traverses each item linearly or following the data structure's pointers. Generally, the traversal starts at the symbol address, but for some special cases, the symbol address indirectly relates to the data structure. For example, the data structure *pglist_data* does not have the root address from the symbol table, but its subfield *node_mem_map* corresponds to the kernel symbol *mem_map*. The redirection is executed at the function *alloc_node_mem_map*, which is a fixed offset from the first address of the *pglist_data* structures. Thus, NIGHTHAWK

needs to analyze the path of data redirection before traversal. After searching each data structure instance, NIGHTHAWK fetches the effective data stored in fields of structures. NIGHTHAWK monitors the system data to introspect the state of the target system.

Mitigating Attacks. NIGHTHAWK is co-processor based approach that suffers from the address translation redirection attack (ATRA) [25] and transient attacks [5], [11]. 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 the kernel initialization stage. Second, we obtain the CR3 register value using 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 introspection overhead. Compared to the SMM-based monitoring approaches like Hyper-Check [11] and HyperSentry [10], the introspection interval of NIGHTHAWKbecomes much harder to be gleaned by attackers. Moreover, the code in the IME can run *continuously* without halting the Target Host, and thus attackers cannot predict when a memory page will be checked.

5.4 Measuring Integrity Via Custom IME

Next, we discuss the introspection workflow in NIGHT-HAWK. 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 a 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 first compares the hash value of the fetched memory with the original version established during boot in the IME system. During a reconstruction command, NIGHTHAWKanalyzes the binary data and extracts the valid messages from the corresponding data structure.

4) After analysis, `data_transmit()` transfers the results to the Remote Machine to continue analysis.

Next, we discuss key aspects of the introspection workflow.

μDMA Based Memory Fetching. NIGHTHAWK uses the μDMA engine to access the Target Host's physical memory from the IME. Our prototype's chipset [26] 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 a special case like ATRA attacks, 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 [27] 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.

5.5 Monitoring Host State Via Custom IME

After we verify the integrity of kernel on both OS and Hypervisor, we can further monitor the critical information on the Target System, including process information, physical memory usage, CPU interrupts, and *Procfs* information, which directly reflects the runtime state of the Target System [28], [29], [30], [31]. This serves as an effective basis for runtime monitoring of the Target Host.

Process monitoring is a key function for host introspection. In the user-space level, the processes running in the Target System are the main interface for users' services. However, malicious software or rootkits [2], [32] residing in the Target System will occupy computing resources. Additionally, they hide themselves by manipulating the kernel objects. To detect such malicious processes, we develop a cross-view comparison approach. Since the kernel object of each process is represented by a *task_struct*, and root data located in *init_task*, we traverse the doubly-linked list of tasks, then reconstruct a *pid* table of new processes. In addition, because *Procfs* can manage kernel modules, we can reconstruct another *pid* table with the data structure *proc_dir_entry* and root instance *proc_root*, similar to the *ps* command. Comparing these two tables, we can track the trace of process hiding rootkits or other malicious software. In addition, through continually monitoring processes, we can detect the behaviors of runtime process for study.

Physical memory page usage can explain how busy the CPU is as well as memory overhead. When processes run in a system, we calculate the memory overhead of the specific application based on runtime state. On one hand, we can estimate the memory requirement of process through source code analysis or emulation [24]. On the other hand, by continually monitoring physical page usages, we obtain an effective memory usage view, which can be used to detect the malicious behavior of memory operations like "malicious memory occupied or overflow" [33]. To monitor the physical memory page usage, we first address the global value *mem_map* from the IME side–the first address of an array of *pglist_data* structures. Through analyzing the value in the *flag* field in data structure instance, we read out the current state of the page. After collecting entire page items, and comparison with a previous state, we can deduce the memory usage of the current process.

Monitoring the record of CPU interrupts will show the number of interrupts per IRQ (Interrupt ReQuest). This can also reflect the behaviors of the processes. For instance, we detect the network interrupts with accessing the root address of *irq_desc_tree* and data structure *irq_data*, which can be used to find networking statistics. For a comparison, we can configure the performance monitoring register and detect interrupts from the SMM side. With the cross view of interrupts on network device, we can introspect such network service in the target system.

Procfs is a virtual file system designed in Linux kernel, which provides a crucial interface for users to access system core information (e.g., users set kernel variables or retrieve kernel information at the runtime system). Unlike typical durable filesystems, *Procfs* is an in-memory interface to various system statistics and runtime information. In addition, *Procfs* is a common target for various viruses and rootkits (e.g., Dynamic Kernel Object Manipulation (DKOM) [34] attacks modify the data transferring function executing once accessing the `/proc/iomem` data). Fortunately, reconstructing *iomem* data from *resources* data structure will directly get the raw data which cannot be hided by attacks. Similarly, monitoring on other `/proc` nodes like `/ioports`, *slabinfo* can also provide effective introspection for the Target OS.

TABLE 1
Communication Commands in NIGHTHAWK, Each Consisting of an Operation and Corresponding 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.
R	Reconstruct the special data in host system.	PT	The running processes in target system

Any command can be combined with any object.

5.6 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, and sizes 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 1. 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. Data reconstruction is also processed in the IME system because memory fetching follow the definite data structure and physical address layout. For the large memory dump, the memory forensic analyses are offloaded to the Remote Machine.

TABLE 2
The Effectiveness of NIGHTHAWK Introspection

Type	Attacked Object	Attacks [2], [32], [36]	Detected
OS kernel	system call table	benign	×
	kernel _text	*pusezk*	✓
	kernel _data	*Diamorphine*	✓
	IDT_table	*kbeast*	✓
	page directory entry	*amark*	✓
		adore-ng	✓
	page table entry	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	✓

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 [35] 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

6.1.1 Effectiveness of Lower Layer Introspection

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 2, which fall into two categories:

- 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* [32]. *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 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 (e.g., the SMM Reload program [36]) 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 Section 5.3 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.

6.1.2 Effectiveness of Host State Monitoring

We measure effectiveness of host state monitoring. Through physical memory analysis, we can reconstruct the process *task_struct*, physical memory page management, and *Procfs* structure, which effectively introspect the runtime state of each application running in the host. The additional experiment focuses on implementing kernel object monitoring applications, including the following three tasks: process listing, physical memory page usage, and system related functions.

First, we create a process status view by traversing all tasks running in the Target Host using the *init_task* symbol.[5] We fetch each *task_struct* from memory through one DMA access. For comparison, we then build a new process view by printing the */proc* file system nodes within the Target Host. The process state in those two views should be consistent if the system running in a normal state. In this experiment, we preset the processes running in the Target Host, and execute *Diamorphine* [2], a rootkit designed to hide itself from kernel-based anti-virus detection tools. We search the processes with our approach. The cross-view result shows that we can identify the rootkit from our constructed *task_struct*, but that cannot be found in */proc* nodes. Other attacks may hide itself from *task_struct* but exploit in */proc* node, or hide from both approaches. However, those
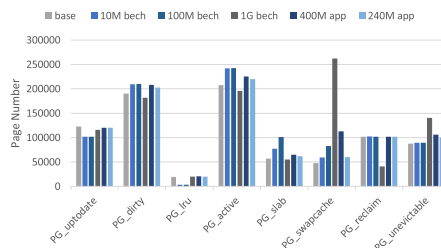

Fig. 5. Physical memory page usages monitoring under different runtime stages.

rootkits still leave the trace in physical memory and can be found by NIGHTHAWK (e.g., we can check the *run_list* of the scheduler for detection [37]).

Second, we develop a physical memory page monitoring module to check the state of physical memory pages (i.e., page update, active, dirty, etc.). Then, we can traverse all physical pages through this single node structure. In Linux, there is a initial data *contig_page_data* used by *Page* array. Each item in *Page* array represents a $4K$ size physical page. By accessing these items, we get the state of the corresponding pages. With this page traversal, we get the dynamic physical memory utilization from host. Different processes running in the Target System will cause different changes to page usage. For example, we use the *memtester* memory benchmark tool [38] to simulate memory utilization patterns. Fig. 5 shows the physical memory page state under different situations. The "base" means a basic system without any other software, "10M–1G bech" represents running the 10M to 1G memory benchmark test, the left instance "400M, 240M app" represents the manual designed process which required such size of memory space. The result shows that the changing page number for each page states follow the size of memory requirement. Note that, for an extreme case like the 1G benchmark memory test, the number of dirty page or active page is less than prediction. This is because the benchmark test is repeated on the same page. In a real situation, we also need to consider the specific memory operating in the workflow of process.

Third, we analyzed the binary code from physical memory dump to reconstruct the *Procfs* information. We selected part of functions in *Procfs* node, including `/iomem`, `/ioports`, `/slabinfo`, and `/interrupts`, which can reveal the memory allocations, ports, memory switching, and CPU interruption events on the Target Host. As mentioned at Section 5, most of *Procfs* nodes have their own data structure. By traversing these structures, we can fetch a complete view of */proc* information. We experimented above functions by reconstruct the key data. Fig. 6 lists parts of the result of memory forensics by NIGHTHAWK. With this function, we can monitor system state.

6.1.3 Effectiveness of Other Special Attack Introspection

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
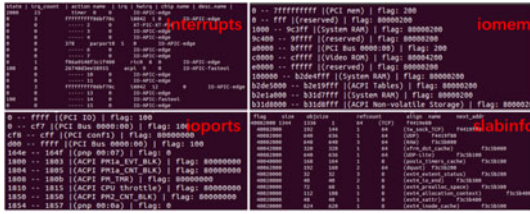
---

5. On our system, the address $0xc1938a00$ contains the symbol *init_task*; each *task_struct* object is 0xD8C bytes.

Fig. 6. Result about *proc* nodes reconstruction from Nighthawk side.

Fig. 7. Memory throughput degradation due to introspection.

Directory and CR3 changed under Kernel Page Table Isolation (KPTI) mechanism and IDT based attack [25]. 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 [38], 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 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, more details in previous work [14].

6.2 Efficiency

The efficiency of Nighthawk is mainly determined by the time cost of three logical operations: (1) data fetching, (2) IME-inner 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.

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. When the size of DMA-transmitted memory is smaller than 64 KB, the time consumed is approximately 0.26s. 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. More details about μDMA performance are in previous work [14].

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 [39] to measure the performance degradation imposed on the target machine. We run each host system functions (processes list, physical page usage and
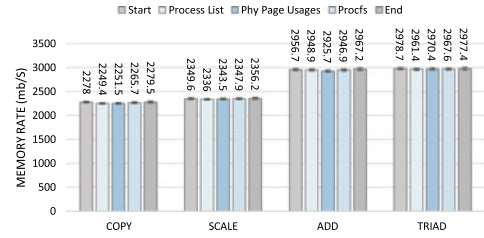
Procfs-similar) in IME to keep the host memory accessing. Fig. 7 shows there are minimum differences in memory bandwidth with and without Nighthawk host system introspection : most of the time, the performance degradation is less than 0.3 percent, and even in the worst case (i.e., in the *Add* function test), the degradation is only 0.98 percent.

6.2.2 IME-Inner Checking Overhead

The second operation we measure is integrity checking and key data reconstruction. First, for the lower-layer 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 [27]. Our test result shows that, to compute a hash value for a 4 KB memory page, the algorithm takes 7.3ms. 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 2ms each time. Second, for data reconstruction, the main overhead made at stage of iterative searching physical memory by multiple DMA accesses. The time spent on checking stages is to read the fixed offset in the memory dump, which takes less than 1ms 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 4.9s 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 3 summarizes the time spent in each operation and in total. For instance, the system call table or the SMRAM (unlocked[6]), the introspection takes less than 1.5 seconds to acquire the integrity status. For application level monitoring (e.g., searching the *iomem*

---

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

TABLE 3
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
iomem data	2.80	20.6±0.80	0.571±0.080	0.231±0.030	21.4±0.90

data), we incur more time overhead due to multiple physical memory traversals totaling about 20s. Fortunately, this performance can be substantially improved since the IME chip was updated in new x86 chipsets.

6.2.5 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 $n = 100, 200, \ldots, 10000$ 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. However, in latest system, Intel has switched ARC chip to using their own x86 Quark microcontroller. The new CPU speed can reach at 412 DMIPS (generally 412 million instructions each second), this should be much more higher than our testbed and it keeps as a further work.

7 RELATED WORK

Trusted Execution Environment. Trusted execution environments (TEEs) are intended to provide a safe haven for programs to execute sensitive tasks. Typically, software-based approaches leverage virtualization. Terra [40] 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 [41] 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. such as external hardware-based peripherals [5], ARM TrustZone [42], Intel SMM [43], Intel SGX [44], [45], and AMD memory encryption technology [46]. ZERO-KERNEl [47] designed secure GPU based mechanism to defense against the kernel privileged attacker with minor overhead. vTZ [48] is the representative work to use ARM TrustZone to virtualize TEE for multi-guest OS protection. HyperCheck [11] employs Intel SMM to build a TEE and monitor hypervisor integrity. Chevalier *et al.* [49] 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 [23], 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 [50], [51], [52]. For example, to analyze the code in the IME, Sklyarov [50] proposed an SPI-based approach to fetch the IME firmware from the storage flash chip. In other work, Sklyarov [51] 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], [13]. Tereshkin *et al.* [13] 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 [53], [54], [55] 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 [49].

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 [56]), 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 [57]. Note that the IME accessing reserved 16 MB memory at the top of DRAM does not go through the Intel VT-d remapping (i.e., I/OMMU implementation of Intel) [26], 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 (Section 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 CONCLUSION

In this paper, we presented NIGHTHAWK, a transparent introspection framework for verifying the memory integrity of a Target Machine and monitoring the state of runtime host system. 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: 1) NIGHTHAWK is able to detect real-world attacks against OS kernels, Xen- and KVM-based hypervisors, and System Management RAM. 2) NIGHTHAWK is able to monitor the state of runtime host system including executed processes, physical memory usage, critical information from *Proc* file system. The experimental results show NIGHTHAWK verifies the integrity of target host system with a low performance overhead, and effectively monitor the state of runtime host system in transparency.

ACKNOWLEDGMENTS

REFERENCES

[1] National Institute of Standards, NIST, "National vulnerability database," 2018. [Online]. Available: http://nvd.nist.gov
[2] Github, "RootKits list," 2018, [Online]. Available: https://github.com/d30sa1/RootKits-List-Download
[3] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction," in *Proc. 14th ACM Conf. Comput. Commun. Secur.*, 2007, pp. 128–138.
[4] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "VMM-based hidden process detection and identification using Lycosid," in *Proc. ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ.*, 2008, pp. 91–100.
[5] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang, "Vigilare: Toward snoop-based kernel integrity monitor," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2012, pp. 28–37.
[6] N. L. Petroni Jr., T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot–A coprocessor-based kernel runtime integrity monitor," in Proc. *USENIX Secur. Symp.*, 2004, pp. 179–194.
[7] F. Zhang, K. Leach, A. Stavrou, H. Wang, and K. Sun, "Using hardware features for increased debugging transparency," in *Proc. IEEE Symp. Secur. Privacy*, 2015, pp. 55–69.
[8] M. Malka, N. Amit, M. Ben-Yehuda, and D. Tsafrir, "rIOMMU: Efficient IOMMU for I/O devices that employ ring buffers," in *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 355–368, 2015.
[9] C. Spensky, H. Hu, and K. Leach, "LO-PHI: Low-observable physical host instrumentation for malware analysis," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2016, pp. 1–15.
[10] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky, "HyperSentry: Enabling stealthy in-context measurement of hypervisor integrity," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2010, pp. 38–49.
[11] F. Zhang, J. Wang, K. Sun, and A. Stavrou, "Hypercheck: A hardware-assisted integrity monitor," *IEEE Trans. Dependable Secure Comput.*, vol. 11, no. 4, pp. 332–344, Jul./Aug. 2014.
[12] M. Ermolov and M. Goryachy, "How to hack a turned-off computer, or running unsigned code in intel management engine," in *Proc. Black Hat Eur.*, 2017, pp. 1–16.
[13] A. Tereshkin and R. Wojtczuk, "Introducing ring-3 rootkits," in *Proc. Black Hat USA*, 2009, pp. 1–85.
[14] L. Zhou, J. Xiao, K. Leach, W. Weimer, F. Zhang, and G. Wang, "Nighthawk: Transparent system introspection from ring-3," in *Eur. Symp. Res. Comput. Secur.*, 2019, pp. 217–238.
[15] H. I. Gael, "Intel AMT and the intel ME," 2009. [Online]. Available: https://.intel.com/en-us/blogs/2011/12/14/intelr-amt-and-the-intelr-me
[16] Intel, "Innovation Engine," 2015. [Online]. Available: https://en.wikichip.org/wiki/intel/innovation_engine
[17] Intel Corporation, "Intel trusted execution technology (intel txt): Software development guide," 2017. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/guides/intel-txt-software-development-guide.pdf
[18] R. Wojtczuk and J. Rutkowska, "Attacking SMM Memory via Intel CPU Cache Poisoning," invisiblethingslab.com. https://blog.invisiblethings.org/2009/03/19/attacking-smm-memory-via-intel-cpu.html
[19] J. Yao, "SMM protection in EDK II," 2017. [Online]. Available: https://uefi.org/sites/default/files/resources/Jiewen%20Yao%20-%20SMM%20Protection%20in%20%20EDKII_Intel.pdf
[20] Open Source Project, "Meshcommander," 2019, [Online]. Available: http://www.meshcommander.com
[21] Synopsys, "embARC," 2019. [Online]. Available: https://embarc.org/embarc_osp/doc/build/html/arc/arc.html
[22] The Fedora Project, "TBoot," 2018. [Online]. Available: https://sourceforge.net/projects/tboot, 2018.
[23] X. Ruan, *Platform Embedded Security Technology Revealed: Safeguarding the Future of Computing with Intel Embedded Security and Management Engine.* New York, NY,USA: Apress, 2014.
[24] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion, "SoK: Introspections on trust and the semantic GAP," in *Proc. IEEE Symp. Secur. Privacy*, 2014, pp. 605–620.
[25] D. Jang, H. Lee, M. Kim, D. Kim, and B. B. Kang, "ATRA: Address translation redirection attack against hardware-based external monitors," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 167–178.
[26] Intel Corporation, "Intel 3 series express chipset family," 2007. [Online]. Available: https://www.intel.com/Assets/PDF/datasheet/316966.pdf
[27] A. Partow, "General purpose hash function algorithms," 2018. [Online]. Available: http://www.partow.net/programming/hashfunctions
[28] Y. Cheng, X. Fu, X. Du, B. Luo, and M. Guizani, "A lightweight live memory forensic approach based on hardware virtualization," *Inf. Sci.*, vol. 379, pp. 23–41, 2017.
[29] L. Cui, Z. Hao, L. Li, and X. Yun, "SnapFiner: A page-aware snapshot system for virtual machines," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 11, pp. 2613–2626, Nov. 2018.
[30] X. Zhang, X. Wang, X. Bai, Y. Zhang, and X. Wang, "OS-level side channels without procfs: Exploring cross-app information leakage on iOS," in *Proc. Symp. Netw. Distrib. Syst. Secur.*, 2018, pp. 1–15.
[31] F. Pagani and D. Balzarotti, "Back to the whiteboard: A principled approach for the assessment and design of memory forensic techniques," in *Proc. Conf. USENIX Secur. Symp.*, 2019, pp. 1751–1768.
[32] "Adore-ng," 2018. [Online]. Available: https://github.com/trimpsyw/adore-ng/
[33] G. Pék, Z. Lázár, Z. Várnagy, M. Félegyházi, and L. Buttyán, "Membrane: A posteriori detection of malicious code loading by memory paging analysis," in *Eur. Symp. Res. Comput. Secur.*, 2016, pp. 199–216.
[34] J. Stuettgen, "On the viability of memory forensics in compromised environments," Technische Fakultä Univ. Erlangen-Nuremberg, Erlangen, Germany, 2015.
[35] G. Combs, "Wireshark," 2019, [Online]. Available: https://www.wireshark.org
[36] L. Duflot, O. Levillain, B. Morin, and O. Grumelard, "Getting into the SMRAM: SMM Reloaded," in *Proc. CanSecWest*, 2009, pp. 1–47.
[37] F. Zhang, K. Leach, K. Sun, and A. Stavrou, "SPECTRE: A dependable introspection framework via system management mode," in *Proc. Ann. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2013, pp. 1–12.

[38] Github, "ToorKit," 2015. [Online]. Available: https://github.com/deb0ch/toorkit

[39] J. D. McCalpin, "Stream," 2018. [Online]. Available: http://www.cs.virginia.edu/stream/ref.html

[40] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," in *Proc. ACM SIGOPS Operating Syst. Rev.*, 2003, pp. 193–206.

[41] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," in *Proc. ACM Symp. Operating Syst. Princ.*, 2007, pp. 335–350.

[42] ARM Ltd., "ARM Security Technology - Building a Secure System using TrustZone Technology," 2009. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf

[43] Intel, "64 and IA-32 Architectures Software Developer's Manual," 2018. [Online]. Available: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[44] M. Hoekstra, R. Lal, P. Pappachan, C. Rozas, V. Phegade, and J. del Cuvillo, "Using innovative instructions to create trustworthy software solutions," in *Proc. Workshop Hardware Architectural Support Secur. Privacy*, 2013, Art. no. 11.

[45] H. Wang *et al.*, "Towards memory safe enclave programming with rust-SGX," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. ACM*, 2019, pp. 2333–2350.

[46] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption, White Paper," Apr. 2016. [Online]. Available: http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf

[47] O. Kwon, Y. Kim, J. Huh, and H. Yoon, "ZeroKernel: Secure context-isolated execution on commodity GPUs," *IEEE Trans. Dependable Secure Comput.*, to be published, doi: 10.1109/TDSC.2019.2946250.

[48] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vTZ: Virtualizing ARM trustzone," in *Proc. Conf. USENIX Secur. Symp.*, 2017, pp. 541–556.

[49] R. Chevalier, M. Villatel, D. Plaquin, and G. Hiet, "Co-processor-based behavior monitoring: Application to the detection of attacks against the system management mode," in *Proc. 33rd Annu. Compu. Secur. Appl. Conf.*, 2017, pp. 399–411.

[50] D. Sklyarov, "Intel ME: Flash file system explained," *in Proc. Black Hat Europe*, 2017, pp. 1–32.

[51] O. Sklyarov, "Intel ME: The way of the static analysis," *in Proc. TROOPERS17*, 2017, pp. 1–51.

[52] P. Stewin and I. Bystrov, "Understanding DMA malware," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, 2012, pp. 21–41.

[53] N. Corna, "ME cleaner: Tool for partial deblobbing of Intel ME/TXE firmware images," 2017. [Online]. Available: https://github.com/corna/me_cleaner

[54] M. Ermolov and M. Goryachy, "Disabling Intel ME 11 via undocumented mode," 2017. [Online]. Available: http://blog.ptsecurity.com/2017/08/disabling-intel-me.html

[55] Persmule, "Neutralize ME firmware on SandyBridge and Ivy-Bridge platforms," 2016. [Online]. Available: https://hardenedlinux.github.io/firmware/2016/11/17/neutralize_ME_firmware_on_sandybridge_and_ivybridge.html

[56] D. Abramson *et al.* "Intel Virtualization Technology for Directed I/O," *Intel Technol. J.*, vol. 10, no. 3, pp. 179–192, 2006.

[57] F. Zhang, H. Wang, L. Kevin, and A. Stavrou, "A framework to secure peripherals at runtime," in *Proc. Eur. Symp. Res. Comput. Secur.*, 2014, pp. 219–238.

Lei Zhou received the PhD degree in computer science from Central South University. He is currently a postdoctoral fellow with the Department of Computer Science and Engineering, Southern University of Science and Technology. His primary research interests include x86 systems security, including trustworthy execution, hardware-assisted security, and memory forensics.

Fengwei Zhang received the PhD degree in computer science from George Mason University. He is currently an associate professor with the Department of Computer Science and Engineering, Southern University of Science and Technology. His primary research interests include systems security, with a focus on trustworthy execution, hardware-assisted security, debugging transparency, transportation security, and plausible deniability encryption.

Jidong Xiao received the PhD degree in computer science from the College of William and Mary. He is currently an assistant professor with the Department of Computer Science, Boise State University. His research interests include cyber security, with a particular emphasis on operating system security and virtualization/cloud security. He has approximately six years industry experience, including various roles at Intel, Symantec, Nokia, and Juniper.

Kevin Leach received the PhD degree in computer engineering from the University of Virginia. He is currently a senior research fellow with the Computer Science and Engineering Division, University of Michigan—Ann Arbor. His research interests include systems security, specifically the debugging transparency problem, though occasionally work on conversational artificial intelligence, program analysis, medical informatics, and big data applications.

Westley Weimer received the PhD in computer science from the University of California, Berkeley. He is currently a professor with the Computer Science and Engineering Division, University of Michigan—Ann Arbor. His main research interests include consciousness, time, and advancing software quality by using both static and dynamic programming language approaches, and automatic or minimally guided techniques that can scale and be applied easily to large, existing programs.

Xuhua Ding received the PhD degree in computer science from the University of Southern California. He is currently an associate professor with the School of Information Systems, Singapore Management University. His research interests include network and system security, applied cryptography, trustworthy systems for data protection, to design the trustworthy systems in commodity x86, and ARM platforms to counter kernel space attacks.

Guojun Wang received the PhD degree in computer science from Central South University. He is currently a professor with the School of Computer Science and Cyber Engineering, Guangzhou University, China. His research interests include artificial intelligence, big data, cloud computing, Internet of Things, blockchain, trustworthy/dependable computing, network security, privacy preserving, recommendation systems, smart cities, and medical information systems.