Position Paper: Consider Hardware-enhanced Defenses for Rootkit Attacks

Guangyuan Hu
Princeton University
Princeton, NJ, USA
gh9@princeton.edu

Tianwei Zhang
Nanyang Technological University
Singapore
tianwei.zhang@ntu.edu.sg

Ruby B. Lee
Princeton University
Princeton, NJ, USA
rblee@princeton.edu

ABSTRACT

Rootkits are malware that attempt to compromise the system's functionalities while hiding their existence. Various rootkits have been proposed as well as different software defenses, but only very few hardware defenses. We position hardware-enhanced rootkit defenses as an interesting research opportunity for computer architects, especially as many new hardware defenses for speculative execution attacks are being actively considered. We first describe different techniques used by rootkits and their prime targets in the operating system. We then try to shed insights on what the main challenges are in providing a rootkit defense, and how these may be overcome. We show how a hypervisor-based defense can be implemented, and provide a full prototype implementation in an open-source cloud computing platform, OpenStack. We evaluate the performance overhead of different defense mechanisms. Finally, we point to some research opportunities for enhancing resilience to rootkit-like attacks in the hardware architecture.

KEYWORDS

Rootkit, Kernel Integrity, Hardware-enhanced Security

ACM Reference Format:
Guangyuan Hu, Tianwei Zhang, and Ruby B. Lee. 2020. Position Paper: Consider Hardware-enhanced Defenses for Rootkit Attacks. In *Hardware and Architectural Support for Security and Privacy (HASP '20), October 17, 2020, Virtual, Greece.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3458903.3458909

1 INTRODUCTION

Rootkits are serious attacks on operating systems, undermining system integrity for the entire machine. Software defenses have been proposed and implemented but so far, rootkit defenses have not received much attention from the computer architecture community. Our position is that it is an excellent time to take a look at building in proactive hardware architectural or micro-architectural defenses for rootkit-like attacks, especially as the computer architecture community is re-thinking computer architecture definitions (sometimes referred to as ISA 2.0) for speculative execution attacks like Spectre [30], Meltdown [32] and their many variants. We suggest that not just performance optimization features like speculative execution and hardware predictors can cause security vulnerabilities, fundamental features like jump tables, interrupt handlers and process execution lists can also cause security vulnerabilities, as in these rootkit-type attacks. Hence, the purpose of this paper is to describe rootkit attacks, understand how they work and their prime targets, and show insights on what is needed in defense against rootkits. This can lead to improved hardware architecture with better built-in resilience to attacks, including rootkit-like attacks.

A kernel rootkit compromises the security of OS software and data, while at the same time hiding the attacker's malicious objects like malware processes, files and network connections.

To perform a rootkit attack, an attacker needs to first conduct privilege escalation, which has been widely reported [38]. For example, the Common Vulnerabilities and Exposures (CVE) database [3] keeps reporting OS bugs that enable an attacker to conduct privilege escalation attacks to control the target system. After the attacker attains superuser status (with OS root Privilege Level, PL=0), it can modify some OS components or prevent OS software from functioning correctly. A rootkit typically tries to achieve two goals: (1) insert and execute arbitrary malicious code in the system's code path; (2) conceal the existence of malicious activities.

Cloud computing brings new challenges, as well as new opportunities, for rootkit detection and mitigation. In cloud computing, rather than executing on dedicated bare-metal physical machines, cloud customers rent computing and storage resources from public cloud providers in the form of Virtual Machines (VMs). Since the VMs run a standard operating system, the VMs have the same OS vulnerabilities as the ones in traditional computer systems, and the adversary can intrude into the system, and carry out privilege escalation and rootkit attacks in the same ways. In addition, past work [13, 48] showed that VM images in public clouds may contain new malware and software vulnerabilities implemented by a malicious (or compromised) image publisher.

However, cloud computing also gives new opportunities to defeat rootkit attacks inside a Virtual Machine. The main idea is to place rootkit detection and mitigation mechanisms in a lower protection ring (more privileged) than the OS in a Virtual Machine, such as the Virtual Machine Monitor (VMM, also called the hypervisor) [1], the privilege level of which (PL=-1) is higher than that of the guest OS in a VM (PL=0). The virtualization layer, namely VMM, is an ideal location for anti-rootkit defense mechanisms since it cannot be attacked by less privileged attackers. Many tools have been developed to enable a VMM to look inside a VM and monitor its execution. These have been called Virtual Machine Introspection (VMI) techniques to detect vulnerabilities inside a VM (e.g.,

HASP '20, October 17, 2020, Virtual, Greece

ACM ISBN 978-1-4503-8898-6/20/10…$15.00
https://doi.org/10.1145/3458903.3458909

[1] We use hypervisor and VMM interchangeably in this paper.

[25]). These tools have the hypervisor privilege to monitor the internal execution, states and data of the guest VM. They can produce stronger defenses, as malware running in the guest OS in a VM cannot interfere with the hypervisor layer.

Public cloud providers may still be using the traditional methods where security tools are located inside the guest VM, such as Amazon Web Services Inspector [1] and Microsoft Azure Anti-malware [11]. Since these tools and defenses are in the guest OS with PL=0, if the attacker already has this root privilege, they are much more vulnerable than the tools and defenses in the hypervisor. We describe a hypervisor-based defense later in this paper, as well as a prototype implementation in the OpenStack open-source cloud computing software. However, since hypervisors have also been attacked [36] and can introduce significant performance overhead, we believe there is an opportunity for hardware support to improve both the security and the performance of rootkit prevention, detection and mitigation defenses.

The main contributions of this paper are:

- Description of common techniques and mechanisms of kernel rootkits.
- Insights on what is needed for a rootkit defense, and the challenges that must be overcome.
- Prototype implementation of a rootkit defense at the VMM level, leveraging a few VMI functions and integrated into the open-source cloud platform, OpenStack, for easy deployability.
- Proposing new research opportunities for embedding resilience to rootkit-type attacks in hardware architecture.

2 ROOTKIT MECHANISMS

We now describe some common techniques used by rootkits to execute malicious codes and hide malicious objects. We analyze a few real-world rootkits to demonstrate each type of technique. Without loss of generality, the introduced attack mechanisms use the Linux OS as the victim system.

2.1 Hijacking Jump Tables

Jump-tables are lists of entry points of code routines that serve as addresses of kernel functions. These are widely used by OS kernels. Typical examples of jump-tables are the System-call Table and the Interrupt Descriptor Table. A System-call table stores an array of function pointers, where each pointer corresponds to a system call handler that user-space processes can use to invoke kernel functions and services. An Interrupt Descriptor Table (IDT) is used to transfer the execution of a program to special software routines that handle external interrupts or to signal exceptions.

A rootkit hijacks the jump-table, modifies one or more function pointers in the table to redirect them to its own handlers, which conduct malicious actions, and then jump back to the desired system call or interrupt handler. This trick is widely used in many kernel rootkits. Below we demonstrate two examples:

Hijacking System-call Table. Xingyiquan [12] is a rootkit that can establish a network backdoor socket for remote attackers. The attacker can log into the victim system through a terminal and gain OS root privilege (PL = 0), which gives the intruder an illegal inheritance of privilege.

Listing 1: The Xingyiquan rootkit hooks open system call

```
1  struct file sysmap = open("System.map-version");
2  long *syscall_addr = read_syscall_table(sysmap);

4  old_syscall_open = syscall_addr[__NR_open];
5  syscall_addr[__NR_open] = new_syscall_open();

7  malicious_object_name = {"xingyi", "bind_shell",
       "reverse_shell"...};
8  int new_syscall_open(char *object_name) {
9      if strstr(object_name, malicious_object_name)
10         return NULL;
11     return old_syscall_open(object_name)
12 }
```

To achieve stealthiness, the rootkit hides the malicious socket process, executable files and the network socket by redirecting the system call functions to malicious ones, and not returning the information about these malicious processes, files or sockets. For instance, the ps command is a user-level utility to display the running processes. The command collects the status of processes by searching the **/proc** directory in the Linux OS. There are many folders residing in this directory with each of them containing information about a certain process. While traversing the directory, the command invokes the syscall **open** to read the status files in these folders. In this case, the system call executed is actually malicious, which returns fake results.

Listing 1 shows the malicious code of Xingyiquan altering the `open` system call. The rootkit first reads the system map file which contains the base address of the `sys_call_table` (lines 1 – 2). Then the attacker gets the address of the `open` system call from the system-call table, indexed by `__NR_open` and redirects the jump address (lines 4 – 5) to its own open function (line 8). Inside the attacker's `new_syscall_open`, the rootkit checks if the object name being looked up matches the name of malicious objects (line 9). If so, the function returns `NULL` as non-existent (line 10). Otherwise, the function calls the original system call to execute the correct code. When the malicious backdoor program is running, the rootkit sets 'xingyi', 'bind_shell', 'reverse_shell', etc., to be the keywords. The malicious system calls hence hide the program's files that contain such keywords in their file names. The legal user will not be able to find the attacker who has logged in through the covert socket.

Hijacking Interrupt Descriptor Table. HookIDT [9] is a rootkit that inserts arbitrary malicious code in the interrupt handler. When the victim system invokes the interrupt, the rootkit's code will be executed. Listing 2 shows the code of this rootkit. In the x86 architecture, the privileged software can get the base address of the interrupt descriptor table (IDT) table using the `SIDT` instruction (lines 1 – 4). The attacker may redirect any interrupt handler to its own function (lines 6 – 7). In its own function, it can invoke arbitrary code (line 10) and then return to the original interrupt handler (line 11). As a result, the system will execute the malicious code with the kernel privilege every time it handles the interrupt. In its own function, it can invoke arbitrary code and then return to the original interrupt handler.

2.2 Modifying Kernel Codes

Instead of changing the function pointers in the jump-tables, the rootkit can also change the kernel functions. For instance, a rootkit

Listing 2: The HookIDT rootkit inserts malicious code in the interrupt

```
1  unsigned char idtr[6];
2  unsigned long idt_addr;
3  __asm__ volatile ("sidt %0" : "=m" (idtr));
4  idt_addr = *((unsigned long *)&idtr[2]);

6  old_idt_handler = idt_addr[handler_id];
7  idt_addr[handler_id] = new_idt_handler;

9  void new_idt_handler(args){
10     malicious_function();
11     return old_idt_hanlder(args);
12 }
```

can hijack a system call or interrupt handler by changing the first few bytes of its code to the `jmp` instruction, which will jump to the rootkit's malicious codes. The rootkit can thus run malicious codes without modifying the jump-table. This technique can evade the rootkit detectors that only check jump-table integrity.

Modifying system calls . The Cesare stealth-syscall rootkit [18] modifies the syscall code to execute malicious functions. The method is achieved by replacing the first 7 bytes of a syscall's code with a `jump` operation to its fake syscall function, as shown in Listing 3. The Cesare rootkit may also read the address of the system call from the System.map file as shown in (lines 1 – 3). After making a copy of the first 7 bytes in the original syscall code (line 5 – 6), it replaces the bytes to jump with a `movl` instruction and a `jmp` instruction (line 8 – 13). In its fake syscall, the rootkit is able to execute a malicious function (line 16) and then restore the first 7 bytes of the syscall using the copy (line 17). The attacker then returns the correct result from this restored syscall (line 18) as if the system call is not compromised.

2.3 Direct Kernel Object Manipulation

A rootkit can also directly modify some other critical kernel objects, in addition to jump tables and kernel routines. By doing so, the rootkit can hide the existence of malicious processes or network connections. For instance, the `proc` filesystem (`procfs`) is a special filesystem in Linux that establishes communication between kernel space and user space. It presents information about the OS kernel and system to the user program. It also enables users to change kernel parameters at runtime. However, a rootkit can tamper with the `proc` filesystem, and cause the system to present wrong information to the users, thus hiding malicious processes from victim users.

Modifying Procfs system. The Adore rootkit [4] attacks Linux and modern BSD systems. It is designed to hide files and processes controlled by the attacker. The Adore rootkit changes the **lookup** function, which is one of the file operations in the `proc` filesystem. This function is an attribute of the file directory and is a part of the **open** syscall implementation. The **open** syscall invokes this function as the last step to decide whether the operation on a file has been successful. With an altered **lookup** function, opening the status file of a malicious process will return as not successful, which indicates the process doesn't exist so the user is unable to see the hidden process.

Listing 3: The Cesare stealth-syscall to change open syscall function to insert malicious code

```
1  struct file sysmap = open("System.map-version");
2  long *syscall_addr = read_syscall_table(sysmap);
3  syscall_open = syscall_addr[__NR_open];

5  char old_syscall_code[7];
6  memncpy(old_syscall_code, syscall_open, 7);

8  char pt[4];
9  memncpy(pt, (long)malicious_open, 4)
10 char new_syscall_code[7] =
11 {"\xbd",pt[0],pt[1],pt[2],pt[3], // movl %pt, %ebp
12 "\xff","\xe5"};                  // jmp %ebp
13 memncpy(syscall_open, new_syscall_code, 7);

15 int malicious_open(char *object_name) {
16     malicious_function();
17     memncpy(syscall_open, old_syscall_code, 7);
18     return syscall_open(object_name);
19 }
```

In Listing 4 we demonstrate how the Adore rootkit hides a process given its id. The rootkit gets the operation list of the `proc` file system which is an attribute of the file structure (lines 1 – 3). To hide the specified process, the lookup function in the list is redirected to a new one (lines 4 – 5). This new function checks if the process to be looked up is the malicious one. If so, it returns NULL (non-existent) (lines 8 – 9). Otherwise, it calls the correct function (lines 10 – 11). Helped by this stealthy kernel module, the adversary is able to hide its processes.

2.4 More Sophisticated Exploits

In addition to the basic rootkit attacks described above, there are more sophisticated rootkit attacks. Shadow Walker [43] cheats the integrity checking utilities of the system. Subvirt [29] installs itself as a Virtual Machine Monitor (VMM) or hypervisor, and Blue Pill [40] creates a hypervisor on-the-fly. Cloaker [21] exploited the alternate exception vector in an older ARM processor to direct the execution to a malicious payload.

2.5 Summary of Basic Attack Mechanisms

The success of rootkits to cheat the system functions comes from the attacker's ability to escalate his privilege level to the OS root level and write to OS space, which is a severe breach of integrity. Specifically, three types of entities are typically targeted by illegal modifications: the jump tables storing the entry points of kernel functions, the code of kernel functions and the data structures of critical kernel objects.

For stealthiness, the rootkits hooking function pointers change the pointer of a lookup or read function to the address of its wrapper which calls the original function. The wrapper filters out the result about the object to be hidden so that this object is not reported to the user. The modification of function code can also hijack the control flow by replacing the first instruction of the original function with a jump instruction to the attacker's wrapper. A data structure, e.g. a doubly-linked list, could be hacked by the attacker who makes a malicious node's predecessor and successor point to each other so that the node is hidden from a list traversal.

Listing 4: The Adore rootkit hiding a process

```
int malicious_pid;
struct file *procfs = flipopen_open("/proc");
inode_operations *proc_op = procfs->inode_op;
old_lookup = proc_op->lookup;
proc_op->lookup = new_lookup;

dentry *new_lookup(int pid){
    if (pid == malicious_pid)
        return NULL;
    else
        return old_lookup(pid);
}
```

3 THREAT MODEL

We consider the threat model where hostile applications or services may be running inside the customers' VMs, gaining the root privilege of the guest OS, and the capability of compromising the system integrity of the VM. For a software-based solution, we assume that such hostile applications cannot tamper with the host OS and hypervisor layer. We make this assumption because hostile VMs only get guest VM's root privilege (ring 0) while the hypervisor has VMM privilege (ring -1). So the hostile VMs cannot subvert the security functions provided by the hypervisor, which is trusted in this threat model. While we discuss a hypervisor-based rootkit defense solution in Sections 5-7, we point the architecture community to consider hardware-enhanced solutions in the future, as the hypervisor may also be compromised.

4 DEFENSES AGAINST ROOTKITS

We discuss three defense solutions against the kernel rootkits described in Section 2.

4.1 Preventing Malicious Modifications

We can protect the integrity of important jump tables and system data by making sure they remain unchanged. The most straightforward way is to restrict the attacker's ability to write to the critical kernel space. In a virtualized system, there are two levels of memory translations: from guest virtual address to guest physical address which is controlled by the guest OS, and from guest physical address to host physical address, which is controlled by the hypervisor. So we can restrict the rootkits' permissions in one of the memory translations, as described below:

OS-level prevention. We can use the OS feature "Write xor Execute (W⊕X)", in which every page in the system's address space may be either writable or executable, but not both. This can prevent rootkits from modifying critical codes and then executing them. *However, if the rootkits take control of the operating system, they can easily disable such a protection feature by changing the protection bits in the page table entries.*

Hypervisor-level prevention. In hardware-assisted virtualization, the hypervisor uses another page table (termed extended page table (EPT) in Intel technology and rapid virtualization indexing (RVI) in AMD) to translate the guest physical address to the host physical address. Similar to the page table, an EPT entry also has permission bits to indicate whether the entry is readable, writable or executable. These bits provide a second layer of protection outside the bits of page table entries. The hypervisor can disable the Writable (W) bit in the EPT entries of the protected memory regions of the guest VM, e.g., jump tables and critical code sections.

4.2 Detecting and Repairing Integrity Breach

For "defense in depth", we propose a second method which can detect integrity breaches and then repair them, in real time.

Protect jump tables. As suggested in Section 2, the jump tables to syscall or interrupt handler functions could be the first target of the attacker. As the table contains just pointers and is not big, it is reasonable to check the table entries one by one.

We can keep an intact copy of the jump tables from an OS system map or a clean VM. (The head of the syscall table is obtained by reading the system memory layout supplied with the Linux OS, while the IDT base address comes from the interrupt descriptor table register (`IDTR`).) With this copy containing unmodified values, we detect potential rootkits by comparing the current jump table with the original one. For real-time mitigation, the jump table could be restored from the original copy which is saved during the boot-up of the guest VM. This solution requires secure storage for the pristine copy of the jump tables.

Protect system codes and data. Besides the jump tables, we should also protect general system data and code. As the space containing codes and data is much larger than that of jump tables, a word-by-word verification is too expensive to implement. Instead, we can compute cryptographic hashes of the memory region for kernel routines and static data, and compare the hashes with pre-calculated ones to verify integrity. If an integrity breach is found by finding the two hashes different, the functions could be restored from the untampered copy.

4.3 Cross-view Validation of Critical Objects

The first two methods prevent or detect rootkits' actions to target memory regions. We can use a cross-view validation method to detect a rootkit trying to hide objects. Objects checked during the cross-view validation include active processes and network sockets.

This cross-view validation can detect system integrity breaches by comparing two lists of objects obtained from two different views. The first, untrusted, view is to get the list using common user-level utilities. We obtain a second, trusted, view by directly reading kernel space using, for example, a trusted external VM monitoring technique. If some objects only show up in the trusted view, they are considered hidden from the user's untrusted view.

Upon finding hidden objects, the protection mechanism regards them as malicious objects, as regular applications do not hide their existence deliberately. Then actions are taken to kill these objects to prevent further malicious behavior.

5 CHALLENGES AND PRIMITIVES

We now discuss what primitives are needed for a hypervisor-based defense of the VM kernel against rootkit attacks. A key challenge in monitoring a VM is the semantic gap in addressing – the hypervisor needs to access the VM's memory without the OS context. A second challenge is to capture certain dynamic events and trap to the defense handlers without incurring performance overhead. A third challenge is to provide real-time damage mitigation.

5.1 Semantic Gap to Access Guest VM

Accessing guest VM's memory. One basic functionality needed for a defense is to access the memory of guest VMs. The challenge is the semantic gap between the high-level data observed by the guest OS and the low-level data observed by the hypervisor. A process inside the guest OS accesses data via its virtual address, which will be translated to the guest physical address by the guest OS, and then the host physical address by the hypervisor. If the hypervisor attempts to access data of a process in a guest VM at a specified guest virtual address, it has to conduct the two levels of address translation without the context of the guest OS. While painful, this can be done as follows:

The hypervisor first obtains the base (guest physical) address of the process's page directory. If this process is a kernel process, the kernel page directory is stored in a fixed known guest physical address. If it is a user-space process, the hypervisor first gets the process structure list stored in a fixed known guest physical address, and iterates this list until it finds the given process. From this structure, the hypervisor can get the address of this process's page directory. The hypervisor then translates the guest physical address of the page directory into a host physical address, takes the guest virtual address of the data and translates it to the data's guest physical address using the page table. The final step is to translate the data's guest physical address into a host physical address, and access the data from the host physical page. If Address Space Layout Randomization techniques are used in the VM, this can get much more complicated.

Accessing guest VM registers. The hypervisor must also access a VM's (virtual) registers as it may need the value of registers like the page table base address. Fortunately, hypervisors already maintain a set of data structures (e.g., VMCS in Intel processors, VMCB in AMD processors) to save and restore register values for each VM's virtual CPU during a VM context switch; so the hypervisor can read or write any register values from its internal structure.

5.2 Dynamic Event Capturing

A defense should be able to capture the occurrence of some critical functions (e.g, syscalls, APIs). When one such function occurs inside a VM, we want a VM exit to be invoked and the CPU trapped into the hypervisor. We can solve this in two ways: (1) Insert a debugging instruction `INT 3 (0xCC)` at the address of the monitored function, or (2) Set the memory page containing this function as Non-Executable (NX) in the Extended Page Table (EPT) entry. When the VM executes this function, a software interrupt or an EPT violation happens, respectively, and traps the processor to the hypervisor to handle this dynamic event. The rootkit defense handler can then confirm the detection of a rootkit and/or perform mitigation. Then the dynamic event monitoring must be restarted.

5.3 Realtime Mitigation

Real-time mitigation of rootkits can prevent security breaches, and it can be done if we allow the defense mechanism to change the VM's memory or execution paths. This in itself is dangerous, unless the hypervisor is trusted, which is true in our threat model. We recommend real-time mitigation with the following two procedures.

Killing a process. When the defense detects that a malicious process has been launched and running in the guest VM, we can kill this active process immediately to prevent further damage. The idea is to insert the process killing function (e.g., `sys_kill` in Linux) in the VM's code path, and set the function parameter as the malicious process's id. Then the VM will jump to the process killing routine and return to the original code after killing the process.

Repairing compromised data. When the guest OS kernel is compromised by malware modifying critical data, the hypervisor can restore the saved copy of modified data. For instance, malware can change a kernel function pointer to its malicious handler. To repair this, the hypervisor can replace the pointer with its original value from an intact OS of the same version so that the malware's function will not be invoked.

6 PROTOTYPE IMPLEMENTATION

We implement a prototype rootkit defense solution integrated with the open-source OpenStack cloud computing framework. The defense contains 3 mechanisms: preventing modifications, detecting and repairing integrity breaches and cross-view validation. The implementation leverages the routines from the LibVMI library [8].

6.1 Preventing Malicious Modifications

We use the hypervisor to set the Non-Writable(NW) permission bit for the critical codes and data, preventing them from being modified by rootkits. First, the guest virtual addresses of the sensitive jump tables and code are obtained by looking up the kernel symbol translation table in the system map provided by the VM owner. The hypervisor translates these virtual addresses into guest physical addresses. The hypervisor then sets the memory page containing these critical objects as Non-Writable (NW) in the EPT entry and listens for write violations. When a rootkit tries to modify these critical objects, an EPT violation occurs and the processor is trapped into the hypervisor. A callback function is invoked in this case which requires the hypervisor to suspend the VM and stop the damage. The cloud server may inform customers of these violations and perform mitigation methods afterward.

6.2 Detecting and Repairing Integrity Breach

The hypervisor can read the contents of jump-tables and critical kernel codes, and compare them with "good" ones. For the System-call Table, its base virtual address is denoted by the symbol `sys_call_table` in the system.map file which lists the memory layout of a Linux OS. For IDT, its base virtual address is stored in the register `IDTR_BASE`. From the base addresses, we can get the guest virtual address of each jump-table entry. Then we use the address translation scheme described above to translate them to the host physical addresses. We read the data in each syscall table entry from the corresponding host physical address, and compare it with the one from an intact OS kernel of the same version. If one data does not match the corresponding "good" one, we can suspect that the rootkit has changed this handler to its own malicious function. We then restore the original function address in the jump-table to fix this integrity breach.

We also need to verify the integrity of kernel codes. In the system.map file, the lower bound address of the kernel code memory

region that stores the critical kernel functions is named `_stext` and the upper bound address is named `_etext`. We calculate the cryptographic hash value of the data inside this region [`_stext`, `_etext`] and compare it with the one from an intact OS kernel of the same version. Mismatched hash values indicate that certain kernel functions inside this memory region have been compromised. We can then restore the original correct codes within [`_stext`, `_etext`] from an intact kernel to maintain the integrity of the kernel codes.

6.3 Cross-view Validation of Critical Objects

Detecting Hidden Objects. We use cross-view validation to detect objects hidden by rootkits. The untrusted view is from the user commands, and a trusted view is obtained by directly monitoring the kernel space. We can detect hidden processes, and hidden network sockets, by checking if the two views match.

To detect if the VM has hidden processes, the hypervisor needs to get two views of process lists inside this VM. The trusted view shows all the processes while the untrusted view might be tampered with by the rootkits. The trusted view can be obtained from the OS scheduler's task list – if a process is not on this list, it will not be scheduled for execution. We first obtain the virtual address of the list head from the kernel symbol `init_task`. Then we iterate this list, and read each process's information from the `task_struct` kernel structure, e.g., `comm` (process name); `tasks` (pointer to the next process); `mm` (memory descriptor); `pid` (process id), etc. By doing so, we can get all processes running in the OS. To get the untrusted view of the process list, the hypervisor can issue a remote `ps aux` command to the VM via SSH, which is a common way to execute commands on a remote machine. Then the users' view of the process list will be transmitted to the hypervisor. By comparing the two lists, we can identify any hidden process's name and id.

To detect if the VM has hidden network sockets, the hypervisor also needs to get the trusted view and an untrusted view of active network sockets in this VM. For the trusted view, the Linux kernel uses hashmaps to store the network sockets. The TCP hashmap is denoted by the kernel symbol `tcp_hashinfo` and the UDP hashmap is denoted by the symbol `udp_table`. We can get the virtual addresses of these hashmaps from these symbols, and iterate the table to retrieve the trusted list of active sockets. For the untrusted view, the hypervisor can issue a remote `netstat` command to the VM via SSH, and retrieve the list of sockets from the user's perspective. Through comparing the two lists, we can find the hidden TCP or UDP sockets.

We can use the method described in Section 5.3 to kill the hidden processes. For hidden network sockets, we can also kill the processes that establish the sockets to shut down the network sockets.

7 EVALUATION

7.1 Security Evaluation

Table 1 shows whether the rootkits discussed in Section 2 can be detected by the defense mechanisms we describe for our prototype implementation. *Whether the malicious functions invoked by HookIDT and Stealth-Syscall rootkits can be captured depends on what malicious objects the attacker tries to hide so we don't make a definite conclusion about their visibility to cross-view validation.* A combination of protection techniques can defeat all these rootkits.

Table 1: Detection Results

	Description	Preventing Modification	Detecting Integrity Breach	Cross-view Validation
Xingyiquan	Jump table hijacking (system call)	✓	✓	✓
HookIDT	Jump table hijacking (interrupt hander)	✓	✓	Depends
Stealth-Syscall	Kernel code modification	✓	✓	Depends
Adore	Direct kernel object manipulation			✓

7.2 Performance Evaluation

To evaluate the performance overhead of our methods, we use two realistic cloud-based applications, the Hadoop distributed application and an E-commerce website.

Hadoop Distributed Application. We use Hadoop to establish a distributed file system (DFS) with a master node and two slave nodes. The first benchmark is DFSIO, which produces massive file accesses under the MapReduce framework. We test the platform's performance to write and read 10 1-GB files under different integrity protections. The results of write and read operations are shown in columns 2-3 in Table 2.

We compare four protection mechanisms, namely the unprotected baseline, the protection by setting the no-write bit, the periodic integrity check and the periodic cross-view validation. Specifically, the last two methods can choose different periods for VM checking. We test the performance when the period is 1 second, 0.1 second or 0.01 second. The results of DFSIO show that the VMM-based protections add to the execution time. For both the reading and writing tasks of DFSIO, the no-write protection hardly introduces any overhead while the integrity check introduces overheads of 0, 8% and 18% for writing and 0, 10% and 13% for reading when using 1s, 100ms and 10ms respectively as the VM introspection period. The periodic cross-view validation increases the execution time of writing by 1%, 3% and 1% and reading by 2%, 1%, 6% when doing introspection in the three frequencies.

Column 4 in Table 2 shows another Hadoop task running the MapReduce Benchmark (MRBench). It generates many small jobs to test the platform's ability to handle lightweight threads which is complementary to the big jobs we run in the DFSIO task. We loop the small jobs 6 times and each run of a job involves 20 map and 20 reduce operations. We find the write protection and the introspections done every 1s introduce almost zero overhead. A period of 100ms brings 4% overhead for integrity check but it doesn't increase the execution time for cross-view validation. A fine-grained monitoring repeated every 10ms increases the overhead to 18% for integrity check and 11% for cross-view validation. The MRBench gives a similar conclusion as the DSFIO task: doing integrity check and cross-view validation at a period of 1s and 100 ms poses an overhead smaller than 10% while a large overhead occurs if the period is reduced to 10ms.

E-commerce Website. We also use Magento [10] to build an E-commerce website. The platform incorporates a Pound load balancer, some Apache web servers, a Mysql database and a Memcached server. The performance of the platform is measured when requests are sent to the load balancer which is the interface to the outside network. We use httperf tool [5] to establish connections and send requests. Both the connection time and reply time are recorded for requests sent in a minute at a rate of 30 and 40 requests per second.

Table 2: Performance for our 3 rootkit defense mechanisms for two cloud applications: a Hadoop distributed app and an E-commerce app. The numbers in parenthesis are the percentage overhead compared to the Baseline system in the first row which has no defense mechanisms implemented.

	Hadoop MapReduce			E-Commerce Website	
	DFSIO write time(s)	DFSIO read time(s)	MRBench Execution time(s)	Httperf Median Connection Time (ms) (Rate = 30 reqs/s)	Httperf Median Connection Time (ms) (Rate = 40 reqs/s)
Baseline	144.81(0.0%)	505.13(0.0%)	69.66(0.0%)	36.5(0.0%)	51.5(0.0%)
1)Nowrite	146.95(1.48%)	511.83(1.32%)	69.57(-0.12%)	34.5(-5.48%)	54.5(5.83%)
2a)Check 1000ms	144.85(0.03%)	504.91(-0.04%)	69.74(0.12%)	35.5(-2.74%)	46.5(-9.71%)
2b)Check 100ms	156.01(7.73%)	557.46(10.36%)	72.66(4.30%)	38.5(5.48%)	43.5(-15.53%)
2c)Check 10ms	170.77(17.92%)	572.26(13.29%)	82.21(18.02%)	38.5(5.48%)	54.5(5.83%)
3a)Cross 1000ms	146.56(1.21%)	514.16(1.79%)	69.43(-0.32%)	35.5(-2.74%)	48.5(-5.83%)
3b)Cross 100ms	148.66(2.65%)	511.01(1.16%)	69.70(0.05%)	38.5(5.48%)	56.5(9.71%)
3c)Cross 10ms	146.43(1.12%)	538.54(6.61%)	77.09(10.67%)	75.5(106.85%)	541.5(951.46%)

The median connection times to the server under different protection mechanisms are shown in columns 5-6 in Table 2. We observe that the connection time oscillates around the baseline when the platform uses write protection or introspects VMs at a period of 1s and 100ms. However, doing cross-view validation at a period of 10ms leads to very high overhead at about 107% and 951% for the rate of 30 and 40 (reqs/sec), respectively. The instability of results and slow connection times when using the cross-view method is likely due to the fact that the network task is less computation and memory intensive, so the network status and the pauses of VMs when doing validation can contribute more to the performance variance.

7.3 Observations and Discussion

While the protection mechanisms can effectively detect the target rootkits, the latency from the launch of attack to detection is also a key factor for security. The write prevention defense has zero detection latency in that any malicious modification to the kernel will be immediately detected and reported. However, this protection works at the page level and may lack enough flexibility. The periodic introspection method, implemented by our prototype, tries to enforce data integrity by using periodic monitoring of the VM, with an assumption that a higher monitoring rate will bring more trustworthiness. However, the problem remaining unsolved is the ***transient attack*** where the attacker makes illegitimate modification to some sensitive data and restores them after making use of them or before an integrity measurement starts, thus evading the detection. This requires the periodic detection to be done in a fine-grained manner. Our results show that a 10-ms period already introduces non-negligible performance overhead, which motivates hardware-assisted protection.

8 HARDWARE OPPORTUNITIES

The hypervisor-based rootkit defense solution we proposed above is easily deployable and is more secure than the current OS-based defenses [1][11]. Since the hypervisor is much smaller than a full-function OS, it is harder to attack, with a smaller attack surface. However, like all complex software, hypervisors have also been attacked [36]. Hence, we suggest that hardware support can be used to improve the security and the performance of rootkit defenses. Many other hardware solutions are possible in addition to what we introduce below.

Bridging the semantic gap. The semantic gap for addressing content inside a VM in the guest OS context, from outside the VM in the hardware context, becomes even more difficult for a hardware-based solution than for a VMM-based solution. Some architectural support that bridges this semantic gap is desired.

Maintaining an unforgeable view of critical objects. As covered in this paper, the user's view of certain objects could be cheated, e.g., when the view of the process list is obtained from the `proc` file system. The view is considered untrusted as the list is not produced by observing what is running (the active task list) in the hardware. Hardware approaches can provide an unforgeable view of active objects so that attackers cannot hide their malware and connections. This unforgeable view should be available for the many different types of critical objects in operating systems and hypervisors, not just the processes, network connections, and files, that we have discussed.

Faster dynamic monitoring. As suggested in Section 7.3, hardware support can be added to enable faster monitoring of security-critical data structures, jump tables, code regions, etc. Reducing the overhead means the monitoring can operate at a higher frequency and provide improved security. This may need to include automatic re-enabling of dynamic monitoring after an event has been handled.

Unused or Duplicate Resources. We caution against duplicate resources like an alternate interrupt vector table, which makes it easy for an attacker to insert and execute malicious code, by merely invoking this alternate jump table. This was in some older Arm processors [2] and rarely used.

Better hardware access control. We recommend considering hardware access control approaches over periodic monitoring approaches. Hardware access control mechanisms can prevent an unauthorized access immediately, rather than leave a window of opportunity for the attacker, as in the periodic monitoring approach. As we discussed in Section 7.3, transient attacks exist.

We also recommend more flexible hardware access control than just the simple No-Write bit or the W⊕X bit. This can go beyond the simple access permissions specified in page table entries and support more diversified rules implemented in hardware only. Since

these policies are not implemented as page-table bits or fields, malware cannot modify them. As a simple example, a write-once policy can allow the OS to initialize the jump tables in a clean state during boot-up and prevent later modification. Hardware can support this policy so that even if a rootkit is installed later with kernel privilege, it cannot disable the policy to hijack the jump tables.

Future attacks and defenses. An interesting area for future research is to consider future rootkit attacks using more sophisticated techniques rather than the jump table style of dispatching that we discussed in Section 2. For instance, ROP (return-oriented programming) [26] and JOP (jump-oriented programming) [19] have been leveraged to invoke malicious execution by jumping to execute short sequences of existing legitimate code, rather than having to insert malware code. Proactive defenses against these more advanced attacks can also be considered as future research opportunities.

9 PAST WORK

First, we reference the few hardware papers that address some aspect of rootkit detection.

Vashist and Lee [46] proposed SHARK to address the important process-hiding aspect of rootkits. They provide an excellent introduction to rootkit attacks and defenses, before proposing a hardware-OS solution to defeat the stealthiness of malware. SHARK proposed a hardware-assigned process identifier (HPID) that the OS must use; the solution is tightly integrated with the concept of counter-mode encryption that is used to encrypt and decrypt parts of the page tables, based on the HPID. Changing the PID would mean not being able to decrypt page tables correctly, leading to a crash. While a very early pioneering effort, it may be time to look at other hardware solutions, given today's commodity operating systems that are not likely to accept a hardware-generated process identifier. Also, tying an entire solution to an encryption scheme like counter-mode may not be ideal, and there are many unaddressed complications. SHARK also did not address the hiding of network connections and other critical kernel objects, nor the non-stealthy aspects of rootkits, as addressed in this paper.

Some studies showed that hardware performance counters (HPCs) can be leveraged to detect rootkit attacks. Numchecker [47] counted different instructions during the execution of a test program inside the trusted and untrusted OS's, respectively, and detected rootkit attacks using the deviation from trusted observations. Singh et al. [42] discovered 16 most useful HPCs as features to build machine learning based classifiers that can detect some real rootkits. Both methods can only detect the control flow hijacking rootkits but not the rootkits that perform direct kernel object manipulation.

Copilot [37] used a coprocessor to monitor the critical memory region of a Linux kernel. However, attacks were found to evade or cheat this hardware-based detection mechanism [41]. The issue of the semantic gap also added difficulty in implementing Copilot.

Commercial microprocessors have implemented secure enclaves, e.g., the Intel SGX extensions [33], to protect code and data at the application level, but not at the OS level. There have been proposals [14, 45] to make better use of this type of hardware support. Such past work typically considered the OS as a potential attacker, rather than a victim, as in kernel rootkit attacks.

We now also reference some of the many software defenses, as they can inform the design of new hardware defenses.

To address the stealth aspect of rootkits, many software mechanisms have been proposed to create a trusted list of kernel objects for cross-view validation. RootkitRevealer [20] detected hidden files in the Windows OS. Blacklight detected both hidden files and processes [16]. Klister [6] detected hidden processes by reading the kernel scheduler. These defenses execute at the same privilege level as the rootkits so a race condition exists between the rootkit trying to perform something malicious and the defense detecting the rootkit.

Much past work on Virtual Machine Introspection (VMI) exists. This was first proposed by Garfinkel and Rosenblum [25] and may give good ideas for hardware defenses. Payne et al. [34] designed XenAccess, a monitoring library for VMI on Xen. Quynh and Takefuji [39] designed XenKIMONO to detect kernel-level rootkits in Xen-based servers. Jones et al. [28] introduced Lycosid to detect hidden malicious processes. Payne et al. [35] designed Lares, to realize the event notification technique by placing hooks inside the introspected VM. Dinaburg et al. [22] designed Ether, which conducts the malware analysis using VMI. Lengyel et al. [31] built DRAKVUF, a dynamic malware analysis system for Windows OS which achieved fidelity and stealth using VMI, and scalability using VM cloning. Now some tools (e.g., LibVMI [8], Libbdvmi [7], HVI [15]) are well developed to achieve the VMI techniques needed.

To overcome the semantic gap between the guest OS and the hypervisor, Jiang et al. [27] cast the guest VM's view of the OS into the hypervisor to systematically reconstruct internal semantic views of a VM from the outside in a non-intrusive manner. Srinivasan et al. [44] designed a process out-grafting method, which migrated a suspect process from inside the monitored VM to a secure VM which runs the security monitoring tool. This can achieve isolation and removes the semantic gap. Dolan-Gavitt et al. [23] designed Virtuoso to automatically convert in-guest programs into out-of-guest programs that reproduced the same behaviors. Fu et al. [24] designed VM-Space Traveler, which automatically identified the critical data of the monitored VM and redirected the data from the monitored VM to a secure VM for monitoring. Carbone [17] inserted function calls into the introspected VM from the hypervisor to obtain OS information, thus bridging the semantic gap.

10 CONCLUSIONS

This paper introduces the techniques leveraged by rootkits to compromise kernel integrity and the defenses against these. We identify the key system targets of rootkit attacks. We show the main challenges a rootkit defense external to the Virtual Machine (VM) must overcome, and show how we do this for a hypervisor-based (i.e., VMM-based) defense against Linux rootkits, using Virtual Machine Introspection (VMI) techniques. Our evaluation indicates that although VMI services incur little performance overhead when monitoring is done at a low frequency, the cost increases visibly when we pursue finer-grained protection. This suggests hardware support to reduce overhead and also improve security.

We prototype a complete hypervisor-based defense for rootkits to illustrate what needs to be done, and how some of the challenges can be overcome in practice in a real OpenStack implementation.

This enables computer architects to understand innovative techniques that have been used, and evaluate what is best achieved in software versus hardware. Since rootkits attack system-level software, a hypervisor-based defense is good for attacks on a virtual machine's operating system, but if the hypervisor is also attacked, then a hardware-based defense seems appropriate. We suggest that more secure and higher performance rootkit defenses could be achieved with hardware support, and suggest this as a promising new opportunity for hardware security research.

ACKNOWLEDGMENTS

This work is supported in part by NSF SaTC #1814190, SRC Hardware Security Task 2844.002 and a Qualcomm Faculty Award for Prof. Lee. We also thank the reviewers for their comments.

REFERENCES

[1] [n.d.]. Amazon Inspector. https://aws.amazon.com/inspector/.
[2] [n.d.]. ARM926EJ-S Technical Reference Manual: Control Register c1. https://developer.arm.com/documentation/ddi0198/e/programmer-s-model/register-descriptions/control-register-c1.
[3] [n.d.]. Common Vulnerabilities and Exposures. https://cve.mitre.org/.
[4] [n.d.]. Explorations with adore-ng. http://ab-rtfm.blogspot.com/2007/07/explorations-with-adore-ng.html.
[5] [n.d.]. The httperf HTTP load generator. https://github.com/httperf/httperf.
[6] [n.d.]. Klister - Windows Kernel Level Rootkit Detector. https://securiteam.com/tools/5gp0315ffw/.
[7] [n.d.]. Libbdvmi. https://github.com/razvan-cojocaru/libbdvmi.
[8] [n.d.]. *LibVMI*.
[9] [n.d.]. Linux Hook IDT. https://github.com/majdi/deadlands/tree/master/srcs/linux/module/HOOK/IDT.
[10] [n.d.]. Magento Commerce. http://www.magento.com/.
[11] [n.d.]. Microsoft Antimalware for Azure Cloud Services and Virtual Machines. https://docs.microsoft.com/en-us/azure/security/azure-security-antimalware.
[12] [n.d.]. xingyiquan - simple linux kernel rootkit for kernel 3.x and kernel 2.6.x. https://sw0rdm4n.wordpress.com/2014/11/03/xingyiquan-simple-linux-kernel-rootkit-for-kernel-3-x-and-kernel-2-6-x/.
[13] Marco Balduzzi, Jonas Zaddach, Davide Balzarotti, Engin Kirda, and Sergio Loureiro. 2012. A Security Analysis of Amazon's Elastic Compute Cloud Service. In *ACM Symposium on Applied Computing*.
[14] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 1–26.
[15] Bitdefender. [n.d.]. Hypervisor Introspection. http://www.bitdefender.com/business/hypervisor-introspection.html.
[16] Jamie Butler and Peter Silberman. 2006. Raide: Rootkit analysis identification elimination. *Black Hat USA* 47 (2006).
[17] Martim Carbone, Matthew Conover, Bruce Montague, and Wenke Lee. 2012. Secure and Robust Monitoring of Virtual Machines Through Guest-assisted Introspection. In *Intl. Conf. on Research in Attacks, Intrusions, and Defenses*.
[18] Silvio Cesare. [n.d.]. Syscall Redirection Without Modifying the Syscall Table. http://www.ouah.org/stealth-syscall.txt.
[19] Ping Chen, Xiao Xing, Bing Mao, and Li Xie. 2010. Return-Oriented Rootkit without Returns (on the x86). In *Information and Communications Security*, Miguel Soriano, Sihan Qing, and Javier López (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 340–354.
[20] Bryce Cogswell and Mark Russinovich. 2006. Rootkitrevealer v1. 71. *Rootkit detection tool by Microsoft* (2006).
[21] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. 2008. Cloaker: Hardware Supported Rootkit Concealment. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*. 296–310.
[22] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. 2008. Ether: Malware Analysis via Hardware Virtualization Extensions. In *ACM Conf. on Computer and Communications Security*.
[23] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. 2011. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *IEEE Symp. on Security and Privacy*.
[24] Yangchun Fu and Zhiqiang Lin. 2012. Space Traveling Across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. In *IEEE Symp. on Security and Privacy*.
[25] Tal Garfinkel and Mendel Rosenblum. 2003. A Virtual Machine Introspection Based Architecture for Intrusion Detection.. In *Network and Distribution Security Symposium*.
[26] Ralf Hund, Thorsten Holz, and Felix C Freiling. 2009. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *USENIX security symposium*. 383–398.
[27] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. 2007. Stealthy Malware Detection Through Vmm-based "Out-of-the-box" Semantic View Reconstruction. In *ACM Conf. on Computer and Communications Security*.
[28] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2008. VMM-based Hidden Process Detection and Identification Using Lycosid. In *ACM International Conference on Virtual Execution Environments*.
[29] Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen Wang, and Jay Lorch. 2006. SubVirt: Implementing malware with virtual machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy* (proceedings of the 2006 ieee symposium on security and privacy ed.). Institute of Electrical and Electronics Engineers, Inc., 314–327. https://www.microsoft.com/en-us/research/publication/subvirt-implementing-malware-with-virtual-machines/
[30] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2019. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–19.
[31] Tamas K. Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, and Aggelos Kiayias. 2014. Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System. In *Annual Computer Security Applications Conference*.
[32] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 973–990. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp
[33] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the ACM International Workshop on Hardware and Architectural Support for Security and Privacy*.
[34] B. D. Payne, M. Carbone, and W. Lee. 2007. Secure and Flexible Monitoring of Virtual Machines. In *Annual Computer Security Applications Conference*.
[35] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. 2008. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *IEEE Symp. on Security and Privacy*.
[36] Diego Perez-Botero, Jakub Szefer, and Ruby B. Lee. 2013. Characterizing Hypervisor Vulnerabilities in Cloud Computing Servers. In *International Workshop on Security in Cloud Computing*.
[37] Nick L Petroni Jr, Timothy Fraser, Jesus Molina, and William A Arbaugh. 2004. Copilot-a Coprocessor-based Kernel Runtime Integrity Monitor.. In *USENIX security symposium*. San Diego, USA, 179–194.
[38] Niels Provos, Markus Friedl, and Peter Honeyman. 2003. Preventing Privilege Escalation.. In *USENIX Security Symposium*.
[39] Nguyen Anh Quynh and Yoshiyasu Takefuji. 2007. Towards a Tamper-resistant Kernel Rootkit Detector. In *ACM Symposium on Applied Computing*.
[40] Joanna Rutkowska. 2006. Introducing blue pill. *The official blog of the invisiblethings. org* 22 (2006), 23.
[41] Joanna Rutkowska. 2007. Beyond the CPU: Defeating hardware based RAM acquisition. *Proceedings of BlackHat DC* 2007 (2007).
[42] Baljit Singh, Dmitry Evtyushkin, Jesse Elwell, Ryan Riley, and Iliano Cervesato. 2017. On the Detection of Kernel-Level Rootkits Using Hardware Performance Counters. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (Abu Dhabi, United Arab Emirates) *(ASIA CCS '17)*. Association for Computing Machinery, New York, NY, USA, 483–493. https://doi.org/10.1145/3052973.3052999
[43] Sherri Sparks and Jamie Butler. 2005. Shadow walker: Raising the bar for rootkit detection. *Black Hat Japan* 11, 63 (2005), 504–533.
[44] Deepa Srinivasan, Zhi Wang, Xuxian Jiang, and Dongyan Xu. 2011. Process Outgrafting: An Efficient "out-of-VM" Approach for Fine-grained Process Execution Monitoring. In *ACM Conf. on Computer and Communications Security*.
[45] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-sgx: A practical library {OS} for unmodified applications on {SGX}. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*. 645–658.
[46] V. R. Vasisht and H. S. Lee. 2008. SHARK: Architectural support for autonomic protection against stealth by rootkit exploits. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*. 106–116.
[47] X. Wang and R. Karri. 2013. NumChecker: Detecting kernel control-flow modifying rootkits by using Hardware Performance Counters. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–7.
[48] Su Zhang, Xinwen Zhang, and Xinming Ou. 2014. After We Knew It: Empirical Study and Modeling of Cost-effectiveness of Exploiting Prevalent Known Vulnerabilities Across IaaS Cloud. In *ACM Symposium on Information, Computer and Communications Security*.