Design and Implementation of SecPod, A Framework for Virtualization-Based Security Systems

Xiaoguang Wang, Yong Qi, *Member, IEEE*, Zhi Wang, Yue Chen, and Yajin Zhou

Abstract—The OS kernel is critical to the security of a computer system. Many systems have been proposed to improve its security. A fundamental weakness of those systems is that page tables, the data structures that control the memory protection, are not isolated from the vulnerable kernel, and thus subject to tampering. To address that, researchers have relied on virtualization for reliable kernel memory protection. Unfortunately, such memory protection requires to monitor every update to the guest's page tables. This fundamentally conflicts with the recent advances in the hardware virtualization support. In this paper, we present the design and implementation of SecPod, a practical and extensible framework for virtualization-based security systems that can provide both strong isolation and the compatibility with modern hardware. SecPod has two key techniques: *paging delegation* delegates and audits the kernel's paging operations to a secure space; *execution trapping* intercepts the (compromised) kernel's attempts to subvert SecPod by misusing privileged instructions. We have implemented a prototype of SecPod based on KVM. Our experiments show that SecPod is both effective and efficient.

Index Terms—Virtualization, operating system, kernel security, paging-based isolation

1 Introduction

With its privilege, an operating system (OS) kernel is critical to the security of the whole system. Unfortunately, modern kernels are too complicated to be secure-they often consist of tens of million lines of source code. Consequently, an increasingly large number of vulnerabilities are discovered in all major kernels each year [1]. These vulnerabilities are routinely being exploited to take over the system. To address that, researchers and practitioners have proposed many solutions. For example, modern kernels all have built-in exploit mitigation mechanisms such as address space layout randomization(ASLR) [2] and data execution prevention (DEP, or $W \oplus X$) [3]. They significantly raise the bar of functioning kernel exploits. However, these systems are built on top of a weak foundation that *page tables, the data structures that control the memory protection, are always writable in the kernel* (to facilitate frequent page table updates). Therefore, a powerful attacker could exploit a kernel vulnerability to manipulate critical kernel data structures, such as page tables, and circumvent the in-kernel memory protection. To that end, a stream of research has proposed to deploy memory and other protections "out-of-the-box" in a virtualized environment [4], [5], [6], [7], [8], [9], [10], [11]. For example, Patagonix extends the hypervisor to identify and protect the code running in the VM [6]. NICKLE achieves a similar goal through memory shadowing [7].

Virtualization-based security systems are often at odds with recent advances in the hardware virtualization support: many security tools need to intercept and respond to key events in the VM. Each intercepted event causes one or more expensive *world switches* between the virtual machine and the hypervisor. On the other hand, the hardware virtualization support, such as AMD-V and Intel VT, strives to reduce world switches. In particular, the nested paging allows guests to freely update their page tables without involving the hypervisor. However, the guest page table update is a key event that many security tools are interested in [5], [6], [7], [11]. This forces the hypervisor to run in the less-efficient shadow paging mode where updates to guest page tables are trapped and verified by the hypervisor. To reconcile this conflict, it calls a new approach that can accommodate the needs of virtualization-based security tools, but also take full advantage of the hardware virtualization support.

In this paper, we propose SecPod, an extensible framework for virtualization-based security systems. SecPod encapsulates a security tool in a trusted execution environment that coexists with and yet is strictly isolated from the vulnerable kernel. Specifically, it creates a dedicated address space (the secure space) in parallel to the existing kernel address space (the normal space). The secure space is rigorously protected from the normal space by the two key techniques of SecPod, *paging delegation* and *execution*

- X. Wang and Y. Qi are with the Department of Computer Science, Xi'an Jiaotong University, Xi'an 710049, China. E-mail: xiaoguang@stu.xjtu.edu.cn, qiy@mail.xjtu.edu.cn.
- Z. Wang and Y. Chen are with the Department of Computer Science, Florida State University, Tallahassee, FL 32306. E-mail: {zwang, ychen}@cs.fsu.edu.
- Y. Zhou is with Qihoo 360, Beijing 100015, China. E-mail: yajin@vm-kernel.org.

Manuscript received 5 Oct. 2015; revised 19 Jan. 2017; accepted 22 Feb. 2017. Date of publication 2 Mar. 2017; date of current version 16 Jan. 2019. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TDSC.2017.2675991

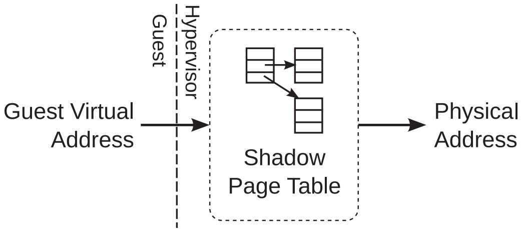


Fig. 1. Shadow paging (GPT not in effect).

trapping: in the former, the kernel delegates all its paging operations, including page tables and their updates, to the secure space. The kernel is deprived of the privilege to directly modify the effective page tables. The secure space enforces a non-bypassable memory isolation by sanitizing the guest page table updates. The latter foils the attacker's attempts to subvert the secure space by misusing privileged instructions. The hypervisor notifies the secure space any such attempts via signals. The secure space can accordingly respond to the event by, say, issuing an alert or terminating the VM. The synergy of these two techniques isolates a security tool from the (compromised) kernel.

We have implemented a prototype of SecPod based on the popular KVM hypervisor [12]. Our prototyping efforts show that SecPod can be integrated into an existing hypervisor with a minimal increase to its code base. Our experiments demonstrate the efficiency and effectiveness of SecPod. For example, SecPod introduces about 2 percent of overhead on average for the I/O-intensive SysBench FileIO benchmark, and about 5 percent overhead on average for the SysBench online database transaction benchmark.

The rest of this paper is organized as the following: in Section 2, we define the scope of the problem and the threat model. We then describe the design, implementation, and evaluation of SecPod in Sections 3, 4, and 5, respectively. Finally, we present the related work in Section 6 and conclude the paper in Section 7.

2 Problem Overview

In this section, we give a brief overview of the hardware virtualization support, particularly the memory virtualization support, and explain how they impact the design of security tools. Early hypervisors for x86 virtualize the guest memory with shadow paging, in which a guest page table (GPT) is superseded by its shadow page table (SPT) [13] (Fig. 1). Specifically, the hypervisor manages a SPT for each guest page table. Any changes to the GPT must be synchronized to its SPT to take effect. This provides an opportunity for security tools to examine and control every change to guest page tables [5], [6], [7], [8], [11], [14]. In shadow paging, GPTs translate guest virtual addresses to guest physical addresses, i.e., the virtual and physical addresses from the guest's perspective. Guest physical addresses must be further translated to the actual physical addresses used by the memory controller. Since SPTs are the only effective page tables, they map directly from guest virtual addresses to physical addresses (Fig. 1).

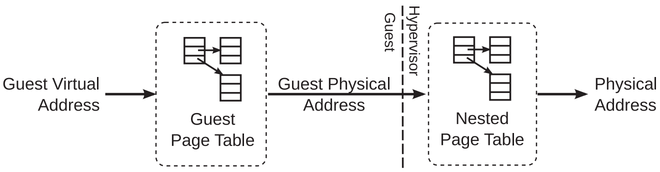


Fig. 2. Nested paging (GPT and NPT both in effect).

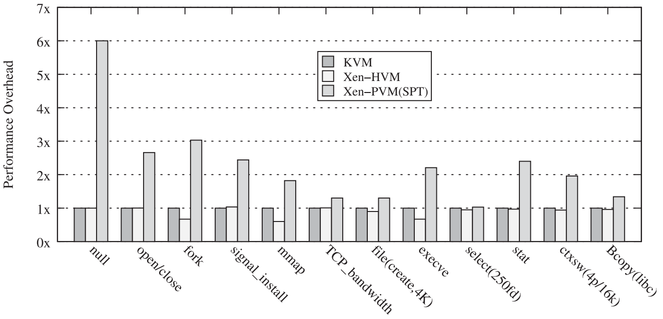


Fig. 3. LMBench performance comparison on KVM guest (with NPT), Xen-HVM (with NPT), Xen-PVM (with SPT). The performance of Xen-HVM and Xen-PVM are normalized to the KVM guest.

Recent x86 processors have the hardware virtualization support. Early extensions focus on trapping sensitive guest instructions, such as `SGDT`, `SIDT` and `MOV to CR3`, to allow the hypervisor to virtualize the related resources. Later revisions aim at improving the performance with the direct support for critical virtualization tasks. Particularly, nested paging is a hardware support for memory virtualization in which the processor translates guest memory accesses with two levels of page tables (Fig. 2): the GPT maps guest virtual addresses to guest physical addresses, and the nested page table further maps guest physical addresses to physical addresses (NPT is also called extended page table. For clarity, we use NPT). Fig. 3 shows the system call performance comparison of NPT and SPT backed VMs using the LMBench micro-benchmark. We used a KVM guest and a Xen HVM (Hardware-assisted VM) guest to measure the performance of NPT-backed VMs. Since KVM does not support memory para-virtualization, we chose a Xen PVM (Paravirtualized VM) guest to evaluate the performance of SPT-backed VMs. All of the VMs run on the same physical machine and have the same virtual hardware configuration (1 vcpu, 2 GB memory). The results of Xen HVM and Xen PVM are normalized to the KVM guest in Fig. 3. We can see that SPT-backed VMs have much higher performance overhead than NPT-backed VMs (the former has about *2-6x* worse performance for most of the test cases, such as *null syscall, fork, context switch*). This illustrates that recent virtualization hardware brings significant performance benefit over the traditional para-virtualization (shadow paging) based approach. Similar results are reported by others [15].

With the hardware virtualization support, the guest has full control over its GPTs, while the hypervisor manages NPTs and is not aware of changes to GPTs. Consequently, memory protection enforced in NPTs can be circumvented by remapping the (protected) guest virtual memory in GPTs. For example, data execution prevention (DEP) enforced in the NPT can be foiled by remapping the guest kernel code to the writable-and-executable physical memory. Because of this, many virtualization-based security systems cannot take full advantage of nested paging, which has tremendous advantages in performance over shadow paging [15].

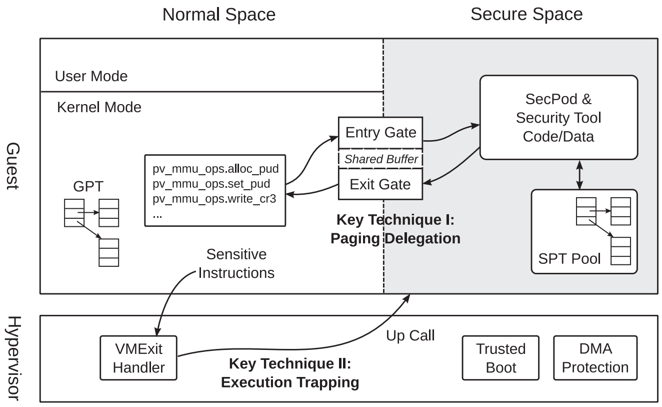


Fig. 4. The overview of SecPod.

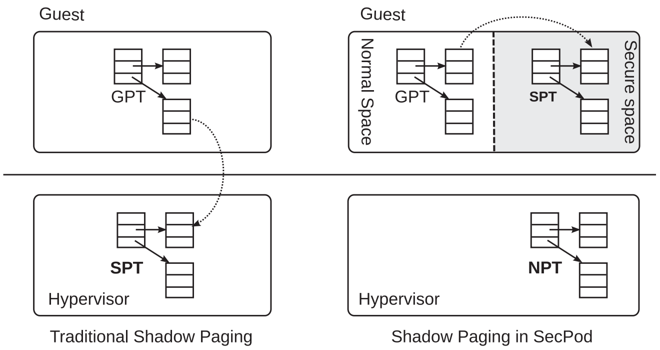


Fig. 5. Shadow page table in virtualization & SecPod.

Threat Model and Assumptions. In this paper, we assume a trusted booting protocol, such as tboot [16], is used to securely load the hypervisor, which in turn loads the guest OS and initializes SecPod. The guest kernel is benign but contains exploitable vulnerabilities. After boot, we assume the presence of a powerful attacker that can change arbitrary memory of the kernel by exploiting some vulnerabilities. Moreover, we consider the hypervisor to be trusted. This can be guaranteed by recent advances in the hypervisor integrity through formal verification and integrity protection and monitoring [17], [18], [19], [20].

3 System Design

3.1 System Overview

SecPod aims at providing a trusted execution environment for virtualization-based security tools. Fig. 4 gives an overview of SecPod with the two key techniques: *paging delegation* and *execution trapping*. In this architecture, security tools run in a dedicated secure space defined by the SecPod page table, while the kernel runs in the normal space defined by the kernel page table. An entry gate and an exit gate are responsible for switching these two spaces. This is essentially a page table based isolation [9], [20], [21]. To switch the space, the entry or exit gate only needs to load the respective next page table into `CR3`, the page table base register of x86. The entry gate is the only way to enter the secure space from the normal space as guaranteed by execution trapping. SecPod provides one-way visibility into the kernel-a security tool in SecPod can introspect and even modify the kernel memory, but not the other way around.

However, simple page table based isolation is not secure for three reasons: *first*, the kernel still has full control over its page table. This allows the (compromised) kernel to subvert SecPod by mapping and modifying the secure space memory. It is thus critical to validate the kernel's page table updates to enforce strict memory isolation. SecPod solves this challenge with the first technique, *paging delegation*, in which the kernel delegates all its paging operations to the secure space, including page tables, page table updates, and task switches (one step of a task switch is to load the page table of the next process to `CR3`). Accordingly, the kernel, including kernel exploits, cannot modify its page tables. All the updates must be delegated to and sanitized by the secure space. *Second*, the kernel is still privileged and free to execute privileged instructions. These instructions can be misused to compromise SecPod. For example, the kernel could use the `MOV to CR3` instruction to load a crafted page table to bypass the secure space. SecPod relies on the second technique, *execution trapping*, to eliminate this threat. Specifically, the hypervisor intercepts sensitive privileged instructions executed by the kernel, and forwards the captured events to the secure space as signals. The secure space can decide how to respond, for example, by issuing alerts, ignoring them, or terminating the violating kernel. It can also dispatch the events to the security tools. This whole process is similar to the signal handling in traditional OSes. *Third*, the attacker could attempt to subvert SecPod through DMA attacks [22]. DMA operations by hardware devices use physical addresses, and thus are not translated by page tables (page tables are used by the CPU to translate software memory accesses.) The hypervisor should have already employed IOMMU to thwart DMA attacks. The secure space should be excluded from the memory accessible to devices in IOMMU as well. In the rest of this section, we describe these two key techniques in detail.

3.2 Paging Delegation

SecPod delegates the kernel's paging operations to the secure space in order to enforce memory isolation. Specifically, the secure space maintains the shadow page tables (SPTs) for the kernel. SPTs stay synchronized with the kernel's page tables. Any updates to the kernel page tables must be merged to SPTs to take effect because SPTs are the only page tables used by the CPU. The kernel may keep its own page tables to facilitate implementation, but they are never loaded to the CPU for address translation. This is technically similar to shadow paging in the traditional virtualization systems. Fig. 5 compares these two shadow paging designs. In virtualization, SPTs are managed by the hypervisor, which is responsible for synchronizing any GPT updates to SPTs. SPTs are the only page tables in use for the guest. Accordingly, SPTs translate guest virtual addresses directly to physical addresses (Fig. 1); In SecPod, SPTs are instead managed by the *in-VM* secure space. It is further backed by the nested page tables (NPTs). Both SPTs and NPTs are used by the CPU to translate guest addresses. SPTs thus map guest virtual addresses to *guest* physical addresses. In most cases, a SPT in SecPod is a simple replica of the kernel's page table (unless a memory safety violation is detected and rejected). Shadow paging in SecPod is thus straightforward to implement. This is in stark contrast

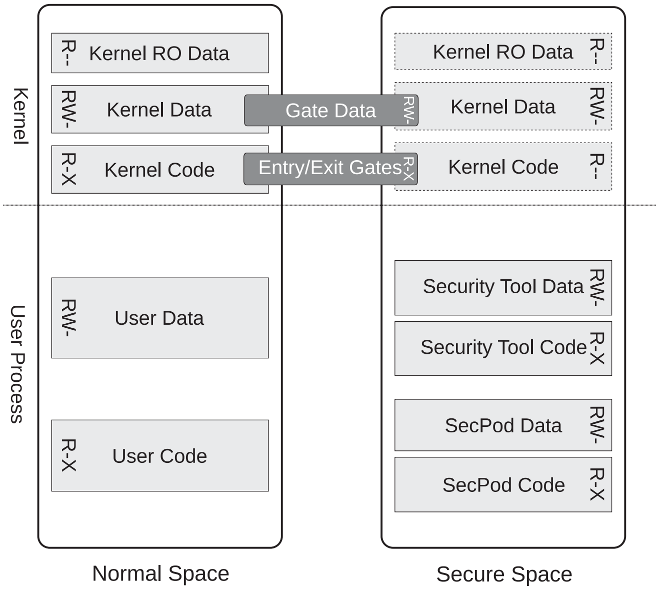


Fig. 6. SecPod address space layout.

against shadow paging in virtualization, which is one of the most complicated modules in a hypervisor due to its support of many paging modes of x86 and the intricate out-of-sync shadowing. Shadow paging in SecPod is also more efficient than the traditional shadow paging-updating SPTs in SecPod take a fast context switch, instead of a much slower world switch in virtualization. In short, SecPod keeps both the simplicity and efficiency of the nested paging. Even though shadow paging has long been used in virtualization, it is, to the best of our knowledge, the first time to be proposed in this architecture.

The kernel delegates its page tables and all paging-related operations to the secure space, such as page table allocation, page table updates, task switches (to write to CR3), and TLB flushing. The secure space exposes, through the entry gate, a service for each of these operations. To delegate these operations, we could replace every paging operation in the kernel with a call to the respective service in the secure space. Fortunately, for kernels that can run in a para-virtualized (PV) VM [13], these hooks have already been embedded into the kernel. For example, the Linux kernel has a pvops framework that can figure out at run-time whether it is running in a virtualized system and accordingly switch to the optimized low-level operations. The pvops framework consists of several groups of low-level operations, such as pv_time_ops, pv_cpu_ops, pv_mmu_ops, and pv_lock_ops (defined in file arch/x86/include/asm/paravirt_types.h). We can repurpose pv_mmu_ops to implement paging delegation (Section 4.1). For a kernel without the PV interface, we can potentially patch the kernel to implement a similar interface.

3.2.1 SecPod Address Space Layout

Fig. 6 shows the layout of the normal and secure spaces. The normal space, as usual, consists of the kernel and the user space. The kernel is mapped at the same location in the secure space as in the normal space. Accordingly, a security tool in SecPod can access the kernel as if it is running inside it since key kernel data structures remain at their supposed locations. This helps mitigate the semantic gap problem [23]. The kernel memory is set to non-executable in the secure space to prevent security tools from executing the (untrusted) kernel code. In the secure space, the secure code and its data are placed in the lower address space because the kernel usually sits at the top (e.g., the Linux kernel often occupies the top 1 GB of the address space). The secure code provides security tools with a compact library of useful functions such as malloc, free, and string functions. The secure data includes a repository of shadow page tables and several hash-based data structures for fast index of that repository (Section 3.2.3). The entry gate is the only entrance to the secure space from the normal space, while the exit gate returns to the normal space. Both gates should be mapped at the same location in the normal and secure spaces because the page table is reloaded during each context switch, and the page-table-reloading code is architecturally required to remain unchanged before and after a context switch [24]. There is also a shared page to pass data between two spaces.

The memory for the secure space is allocated from the kernel when the secure space is created. It is subsequently removed from the kernel so that the kernel will not use it for other purposes. We enforce $W \oplus X$ in the secure space; i.e., the secure space can be either writable or executable, but not both simultaneously [3]. This thwarts code injection attacks against the secure space in case the security tool contains exploitable vulnerabilities. Other attack mitigation mechanisms can also be employed to provide stronger protection of the secure space [2], [25].

3.2.2 Secure and Efficient Context Switch

SecPod implements the page-table based isolation. To switch the spaces, we need to load the page table of the next space into CR3. The secure space only has one page table, the SecPod page table, but the normal space has many shadow page tables, one for each user process. We need to ensure the security and atomicity of context switches. To this end, the entry gate saves the kernel state to the stack (generic registers and interrupt enable/disable status), clears the interrupt (twice), and then enters the secure space by loading its page table and stack to the processor. This process has been described in detail by earlier papers [9], [14]. The exit gate performs the opposite operations in the reverse order to return to the normal space.

To prevent the kernel from subverting the secure space by loading a crafted page table, we request the hypervisor to intercept and check every write to CR3 by the guest (Section 3.3). However, trapping every CR3 write could cause substantial performance overhead due to frequent context switches. To reduce the overhead, we leverage a hardware feature called CR3 target-list [24]. Loading CR3 with one of the four page tables in the CR3 target-list will not be trapped by the hypervisor. This feature has been employed by earlier work for similar purposes [9], [14]. The major difference lies in how memory is virtualized. The previous systems use shadow paging to virtualize the guest memory. Guest task switches are thus handled by and in the hypervisor. This provides a convenient opportunity to update the CR3 target-list (CR3 target-list can only be updated by the hypervisor). On the downside, this prevents

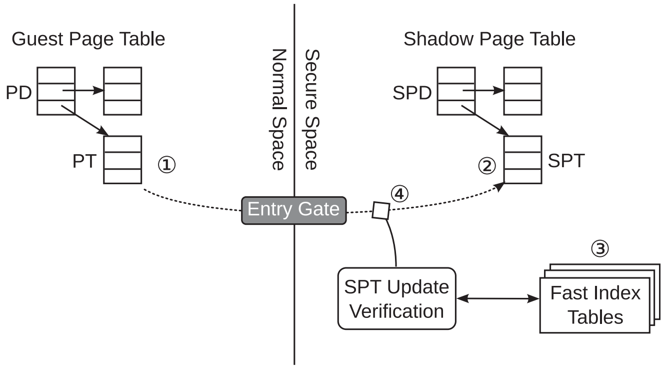


Fig. 7. Kernel page table update verification.

these systems from taking advantage of nested paging. SecPod is designed to avoid this problem.

The hypervisor in SecPod uses nested paging, and the guest delegates its paging operations to the secure space, including task switches. Ideally, task switches in the guest should not involve the hypervisor, just like in the normal nested paging. However, there are many shadow page tables for the guest yet the `CR3` target list can only hold four page table roots. The entry gate will never cause any VM exits because the SecPod page table is locked in the list. But the exit gate will if the SPT for the normal space is not in the list. Neither the kernel nor the secure space can update the `CR3` target-list because they both run in the guest mode. To address that, we allocate a Fixed Top-Level Page Table (FTLPT) in the secure space and copy the top-level page table of the next SPT to it during the task switch. As such, SecPod appears (to the hardware) to be using only two page tables, FTLPT and the SecPod page table. Both of them can be registered in the `CR3` target-list. Therefore, legitimate context switches between the normal and secure spaces will not be trapped by the hypervisor. Our prototype uses the PAE (Physical Address Extension) mode of x86 [24], in which the top-level page table consists of four entries and can thus be copied quickly. Most modern Linux distributions by default use the PAE mode in their kernels because the NX (non-executable) bit is only available in this mode. We would like to emphasize that FTLPT is a part of the SPT pool in the secure space and thus is not accessible by the kernel. Note that we cannot use PCID (Process Context Identifier, also known as ASID) to tag the TLB-the TLB needs to be flushed during context switches because FTLPT translates addresses for many processes. Moreover, PCID is set in the `CR3` register, but the `CR3` target-list can only be changed by the hypervisor.

Context Switch Optimization. `CR3` target-list allows SecPod to utilize in-VM context switches (instead of world switches) to handle page table updates. Nevertheless, context switch is still a relatively expensive operation because it causes TLB to be flushed and the instruction cache polluted (by the secure code). To reduce this overhead, we can employ the lazy page table update: updates to page tables will not take effect unless the TLB is freshened with the new translations. Therefore, we can temporarily delay the page table updates until the TLB is flushed by the kernel, either explicitly using special instructions (e.g., `invlpg`) or implicitly through task switches. This is technically similar to the out-of-sync shadow paging in hypervisors like Xen [13], in which shadow page tables are not synchronized with guest page tables all the time. They are resynchronized when necessary. It is also similar to the asynchronous, exception-less system call handling [26].

3.2.3 Page Table Update and Validation

The kernel delegates paging to the secure space to prevent unauthorized modifications to its page tables. It leverages the para-virtualized MMU interface (`pv_mmu_ops`) to forward low-level paging operations to the secure space. Fig. 7 illustrates how a new level-3 (L3) page table is created and filled. When the kernel needs to allocate a new L3 page table, it sends the request to the secure space (①in Fig. 7), which responds by allocating a blank L3 page table from the SPT pool and linking it to the parent shadow page table (②). The mapping between the GPT and the SPT is then recorded in a hash table for fast indexing (③). When new page table entries are added to the GPT later, it is synchronized to the associated SPT only if no violation of memory protection is found (④). The verifier uses several hash tables for fast fact checking. The secure space has full control over the kernel's memory protection. Any updates to shadow page tables must be vetted by the secure space. By default, the secure space enforces the normal/secure space isolation and $W \oplus X$ for the kernel:

Normal/Secure Space Isolation. This policy prevents the (untrusted) kernel from manipulating the secure space memory. Specifically, the kernel is prohibited from mapping any of the secure space memory, except the entry and exit gates at their fixed location. For each request to change a shadow page table, SecPod checks whether the physical page belongs to the secure space and whether the virtual address overlaps with the two gates (one code page and one data page). The update is denied if either test returns true. By doing so, the kernel cannot map the secure space memory or change the gates.

Kernel $W \oplus X$. Kernel code integrity ($W \oplus X$) is essential to many security tools [6], [8], [11]. Previous virtualization-based systems leverage shadow paging in the hypervisor to protect kernel integrity. SecPod provides the same level of protection in the VM. We use a template-based approach to enforce $W \oplus X$. Specifically, modern kernels have already deployed $W \oplus X$ (without protecting the page table) [3]. The initial kernel page table could serve as a template for the kernel memory protection. For each update to the kernel mapping, SecPod only needs to compare the new memory protection against the template. Note that SecPod does not intend to externally address weaknesses in the kernel's original $W \oplus X$ implementation (it is better to root-cause and fix them in the kernel). Enforcing $W \oplus X$ in the secure space makes it much harder to bypass. Moreover, key kernel data structures like the system call table are also write-protected for both their virtual addresses and the physical contents.

Some kernel modules might periodically update the access attributes of memory pages. For example, the Just-In-Time (JIT) compilation [27] and the dynamic binary translation (DBT) technique [28] write the generated code to a page and later make it executable. Currently, SecPod does not protect kernel modules with dynamically generated code since JIT/DBT might mark code cache pages writable or

TABLE 1
Trapped Sensitive Instructions

Instruction	Semantics
`LGDT`	Load global descriptor table
`LLDT`	load local descriptor table
`LIDT`	load interrupt descriptor table
`LMSW`	load machine status word
`MOV to CR0`	write to `CR0`
`MOV to CR4`	write to `CR4`
`MOV to CR8`	write to `CR8`
`MOV to CR3`	load a new page table
`WRMSR`	write machine-specific registers

executable from time to time. However, it is feasible to port those tools to SecPod for a more secure JIT/DBT. Specifically, code cache pages could be marked executable in normal space, while these pages are set as writable in the secure space. The JIT/DBT engines are also loaded into the secure space. Therefore, the code cache in normal kernel space is always executable without having to be set writable. This can prevent the code cache from being maliciously modified when it is being updated. Similar approach has been proposed to protect the JavaScript JIT engine [29].

3.3 Execution Trapping

In SecPod, the kernel still has the necessary privilege to execute critical system instructions. Without constraints, this privilege could be misused to subvert the secure space, for example, by loading a malicious page table or even disabling paging. Hence, it is necessary to control the instructions executed by the guest. Simply disallowing these instructions in the kernel's binary does not work because the x86 architecture has variable instruction lengths and "unintended" instructions can be created out of legitimate instructions [30]. Previous software fault isolation systems remove unintended instructions through compiler or binary transformations [20], [31]. In SecPod, we instead configure the virtualization hardware to trap these instructions, no matter whether they are benign or "unintended". Table 1 gives a (partial) list of sensitive instructions trapped by SecPod. Each of them controls some important aspects of the processor. For example, `LIDT` loads the interrupt descriptor table, which determines how interrupts are handled; `MOV to CR0` writes to `CR0`, which consists of switches for many CPU operation modes (e.g., paging enable, protected mode, write-protect bits) [24]. Intercepting these instructions will not cause large performance overhead because most of them are not executed frequently after the kernel has initialized. A notable exception is the `MOV to CR3` instruction that is used by the entry and exit gates for context switches. However, our design guarantees that legitimate context switches will not be trapped by the hardware (Section 3.2.2). Note that, SecPod not only protects these registers, but also the associated data structures, such as the global descriptor table and the interrupt descriptor table (Section 3.2.3).

After the hypervisor intercepts a sensitive instruction executed by the guest, it notifies the secure space of the event. This is similar to the signal delivery in traditional OSes [32]. In fact, they both implement an up-call, except that a signal is delivered from the kernel to a user process while an event in SecPod is delivered from the hypervisor to the secure space. When an instruction is intercepted, the hypervisor saves the current virtual CPU state to the virtual machine control block (VMCB) [24], and copies the saved registers to the data page of the entry gate (to provide the context of the violating instruction). The hypervisor then updates the saved instruction pointer in VMCB to the entry gate and returns to the guest. The CPU restores the guest state from the VMCB and continues its execution to the entry gate. The secure space recognizes that this is an up-call from the hypervisor and handles the violation accordingly.

4 Implementation

We have implemented a prototype of SecPod based on the popular KVM hypervisor [12]. Both the host and the guest run Linux. We added about 100 lines of source code to the hypervisor to set the `CR3` target-list and trap the execution of sensitive instructions. Another 800 lines of source code were added to the guest kernel for paging delegation. The secure space has about 2,300 lines of source code. Our prototype reserves 12 MB of physical memory for holding the SecPod code and data (including SecPod code/data, SPTs, and 64 KB for secure tools), and this trunk of memory is mapped in the secure space at virtual address `0x200000`. While SecPod only occupies 8 KB of the virtual address space in the normal space (i.e., two shared pages for the entry/exit gates and a shared buffer). In the rest of this section, we describe this prototype in detail.

4.1 Paging Delegation

In SecPod, the guest kernel delegates its paging operations to the secure space. This gives the latter full control over the guest's memory mapping and protection. In our prototype, we leverage the Linux kernel's `pvops` interface to forward paging requests to the secure space. The `pvops` interface originates from the Xen project's efforts to create a generic para-virtualized kernel that can adapt to different hypervisors as well as the native, non-virtualized platforms. `Pvops` groups the key para-virtualization operations into several structures, such as `pv_time_ops`, `pv_cpu_ops`, `pv_mmu_ops`, `pv_lock_ops`, and `pv_irq_ops`, and substitutes native operations in the kernel with the corresponding PV operations. For example, the native x86 system uses a single `MOV to CR3` instruction to load the page table. `Pvops` replaces it with an indirect call to the `pv_mmu_ops`→`write_cr3` function. Each virtualization system, as well as the native platform, provides its own implementation of these functions. Particularly, functions for the native platform are simple wrappers of the original native instructions or functions. `Pv_mmu_ops` has all the necessary functions for SecPod to delegate paging to the secure space. For example, it has functions for `write_cr3`, `set_pte`, `set_pmd`, `flush_tlb_kernel`, etc. We only need to implement the required functions of `pv_mmu_ops` with the respective services provided by the secure space. In essence, this creates a MMU-only para-virtualized platform as all the other PV operations remain the same as the native platform.

Pvops replaces the native low-level hardware operations with indirect calls through the pv_xxx_ops structures. This introduces some minor but measurable performance overhead to native systems as some of these functions are frequently used by the kernel. Kernel developers have to reclaim the lost performance for native systems. Observing that these functions remain unchanged after initialization, they patch the kernel code to specialize each indirect pvops call with a direct call to the corresponding native function, and even inline simple operations like write_cr3. Therefore, we need to replace the function pointers in pv_mmu_ops before the specialization. Changes to the pv_mmu_ops structure after the specialization will not take effect. To this end, we modify the kernel source code to set up the pv_mmu_ops structure early in the boot process. Because the secure space has not been initialized yet, we use a temporary page table as an in-kernel "shadow page table" and commit the page table updates to it. The temporary page table has to be statically allocated because the kernel memory allocator has not be initialized either. After the secure space is ready to run, we copy the temporary page table to a shadow page table in the secure space.

Our guest kernel is essentially a native kernel with the para-virtualized MMU. We intercepts the MMU operations during the early boot stage. However, any page tables created before that have to be explicitly copied to the secure space. Swapper_pg_dir is such a case. It is statically allocated in the kernel and serves as a master page table for the kernel address space [33]. Each process in Linux has its own user space memory mapping but shares an identical kernel part copied from swapper_pg_dir. No other processes except the idle task use swapper_pg_dir for address translation. If swapper_pg_dir is being loaded to CR3 for the first time, we simply create a new shadow page table for it.

SecPod provides the entry and exit gates for the normal space to call services of the secure space (e.g., to update a page table). Because these gates are the only shared code between the two spaces, context switches have to go through them. The secure space enforces a strict normal/secure space isolation to protect these gates. The implementation details of these gates resemble that of SIM [9]. Specifically, the entry gate first saves the current CPU state to the stack and disables the interrupt with the CLI instruction. It then loads the SecPod page table into CR3 to enter the secure space. The entry gate has to execute CLI again in the secure space in case the (untrusted) kernel has skipped the first CLI instruction [9]. Without a second CLI instruction if the first is skipped, interrupts happened in the secure space halt the (virtual) processor because the interrupt handlers are not executable in the secure space, leading to a denial-of-service attack. Finally, the entry gate loads the secure stack to the stack pointer (the ESP register) and calls the service handler. The exit gate performs the opposite operations in the reverse order to return to the normal space. We also fill the unused space around the entry and exit gates with nop instructions to avoid accidental instructions out of otherwise random bytes [30].

There is a subtle issue in the implementation of the entry and exit gates regarding TLB (translation lookaside buffer) [34]. TLB is a fast cache of the virtual to physical address translation. To access the memory, the CPU first searches the TLB for a matching virtual address. If a match is found in the TLB (a TLB hit), the resulting physical address is sent to the memory unit to access the data. If the mapping is not cached by the TLB (a TLB miss), the CPU walks the page table to translate the address and saves the result in a TLB entry for future references. Therefore, the TLB ultimately determines accessibility of the memory. Simply reloading a new page table cannot guarantee that the TLB contains fresh address translations because global pages will *not* be flushed out of the TLB during context switches (non-global pages are flushed each time a page table is loaded. For example, one way to flush all the TLB entries for the user-space is to simply reload the current page table.) The Linux kernel sets its kernel pages to global because all the processes share the same kernel memory mapping. It is thus unnecessary to flush the kernel mapping from the TLB during task switches. Note that global pages are accessible regardless of the PCID settings. Therefore using PCID cannot solve this problem.

Global pages could potentially cause serious vulnerabilities in SecPod. For example, an attacker could synthesize,[1] in an executable global page, a function that loads the SecPod page table and manipulates the secure space memory. This function remains executable after entering the secure space because its mapping remains in the TLB after the context switch. On the other hand, if the secure space memory is set to global, it remains accessible after returning to the normal space. To address this pitfall, we clear the global bits in both shadow page tables and the SecPod page table, except for the entry and exit gates. By doing so, the TLB will always contain fresh address mappings after context switches, avoiding the aforementioned pitfalls. The entry and exit gates can be set to global because their memory is protected by the secure space and they do not contain enough useful gadgets for return-oriented programming [30].

4.1.1 Page Table Update Batching

As previously mentioned, TLB allows us to batch page table updates because these updates only take effect when loaded into TLB (Section 3.2.2). Thus we could use TLB reloading operations as checkpoints. Guest page table updates inbetween checkpoints can be temporarily cached without being committed to shadow page tables in the secure space. Specifically, we allocate a shared page of memory for the kernel and the secure space. If a paging operation can be delayed, we save its information in that page and directly return to the kernel. The kernel acts as if the change has been committed to the corresponding SPT. This should not cause any problems as the kernel is not supposed to actually use the translation until the TLB is freshened. If a paging operation cannot be delayed, SecPod immediately enters the secure space and commits all the pending operations to SPTs in chronological order.

SecPod delegates the kernel's paging operations to the secure space using the pv_mmu_ops interface. We found that most of the pv_mmu_ops operations can be deferred. For example, alloc_pte(/_pmd/_pud) allocates a new shadow page table. The new SPT cannot be used for address

[1] This can be achieved with the return-oriented programming since SecPod prevents code injection to the kernel.

translation until it is mapped into the address space (by another pv_mmu_ops operation). Moreover, operations like pte_/pmd_update and pte_/pmd_clear have to be followed by a flush_tlb operation to become effective. Flush_tlb is one of the checkpoint operations. The other two types of checkpoint operations are write_cr3 and set_pte: write_cr3 switches to a new page table by reloading the cr3 register. This operation implicitly flushes the TLB. As such, write_cr3 is considered a checkpoint operation; Set_pte (including its variants) interestingly is another checkpoint operation. It is used to set a pte entry in the last level page table.[2] If the entry is originally marked as non-present (i.e., an invalid mapping), there is no need to flush the TLB for the change to take effect. This is because the TLB of the x86 architecture does not cache invalid mappings. A re-read of the address automatically loads the new translation (from the page tables) into the TLB. Therefore, set_pte should be considered as a checkpoint operation as well.

Page table update batching only slightly increases the complexity of SecPod, but it can significantly reduce the performance overhead as demonstrated by our experiments (Section 5).

4.2 Security Tools Case Study

SecPod is an extensible framework for virtualization-based security tools. A security tool running in SecPod is strictly isolated from the vulnerable kernel, but still has flexible visibility into the kernel. First, the kernel memory is mapped identically in the secure and normal spaces (but with different protection). Key kernel symbols and data structures thus can be accessed at their original locations. Second, any changes to the kernel's memory mapping can be intercepted and adjusted, if necessary, because the kernel delegates its paging to the secure space. SecPod also has a simple loader and linker to dynamically load security tools, similar to the kernel module support.

To demonstrate the flexibility of the SecPod framework, we have built two security tools with SecPod. The first tool detects and prevents unauthorized kernel code from execution (e.g., kernel rootkits) [6], [7]. This tool is relatively simple to implement in SecPod, assuming the cryptographic hashes of benign kernel code are known. Specifically, it registers a call back function for kernel page table updates. If a new executable page is created in the kernel, it verifies whether the hash of the page belongs to the hashes of benign code pages. If so, the page is marked executable in the shadow page table. Otherwise, it has detected an attempt to execute unauthorized kernel code and raises an exception. There are a number of challenges in implementing this system. For example, when a kernel module is loaded, the kernel needs to resolve the called kernel functions (e.g., printk) and patches the module with the correct offsets to these functions. This effectively changes the page's hash, leading to a false positive if the hash is calculated on the modified code. We solve this problem by reversing the changes made by the kernel module loader and computing the hash based on the clean code page. After that, we restore the changes and verify that each patched function is an exposed kernel function. Many such challenges have been addressed by previous work [6], [7]. Moreover, we employ a new feature called supervisor mode execution protection (SMEP) in recent Intel processors to prevent the kernel from executing user code. The x86 architecture allows the kernel to execute user code *with the kernel privilege*. SMEP is designed to specifically address this attack. Software based defense is also available [35].

This tool provides a similar security guarantee as Patagonix [6] and NICKLE [7]. Both systems are based on the then-current virtualization technologies, the Xen hypervisor with shadow paging and hypervisors using dynamic binary translation, respectively. In contrast, the implementation based on SecPod can take advantage of nested paging. Note that detecting unauthorized code solely in the NPT is vulnerable unless all the code in the guest is authorized. Otherwise, an attacker can manipulate the GPT, which he has full control over, to map kernel code pages to the unauthorized user code.

The second tool detects kernel malware using data invariants. Many kernel data structures are intricately interconnected. Kernel malware often inadvertently change their connections. For example, all the runnable processes should be contained in the "all-processes" list. Some kernel rootkits hide a process by removing it from the "all-processes" list. The process can still be scheduled for execution as it remains in the runnable process list. Similarly, network connections enumerated from the internal kernel data structures should match those observed from the user space (e.g., with the netstat command). As such, we could detect kernel malware by checking kernel data invariants [36]. Our second tool focuses on the invariants that are often targeted by kernel rootkits. For example, system call table is frequently hooked by rootkits to hide malicious processes and kernel modules. Our tool verifies that system call table entries point to legitimate core kernel functions, not ones in loadable kernel modules. Moreover, the Linux kernel routinely uses structures consisting of function pointers (similar to vtables in C++) to enable a generic design. For example, structure file_lock_operations contains two function pointers related to file locks. There are about 200 such structures. They are often hooked by rootkits to hide malicious entities such as TCP connections. Our tool randomly validates the integrity of those structures related to files, processes, and network connections. For example, it checks that all the TCP connections point to the original, unmodified seq_operations structure (instead of a malicious doppelganger created by rootkits). This tool is not complete in case of the supported data invariants, but demonstrates that SecPod is a practical framework to support this type of security tools: the tool is protected from the potentially-compromised kernel, but still has direct access to key kernel data structures (Section 3.2.1).

The tools we have implemented in secure space have almost the same performance impact on the whole system as the tools implemented inside the kernel. But the security tools in SecPod are much more secure as SecPod provides better memory isolation. Moreover, we expect SecPod enabled security tools are more efficient than tools implemented in a SPT based hypervisor. First, a SPT-backed VM

[2] X86 uses multiple levels of page tables. Setting the last level page table entry makes it ready for address translation.

is less efficient than a NPT-backed VM (as we have shown in Section 2). Second, even without performance consideration, it is hard to implement a security tool in the hypervisor. This is because security tools in hypervisor have to overcome the semantic gap [23]. Meanwhile, the design of SecPod imposes some constraints on the security tools. For example, they cannot directly introspect user-space processes since the user-space memory is not mapped in the secure space; they cannot perform DMA operations to access the guest VM's devices such as the virtual disk; and they cannot execute guest kernel functions (e.g., to kill a malicious process) since the guest kernel is deemed untrusted. Nevertheless, these constraints do not impair the effectiveness of our main focus-to introspect the guest operating system kernel.

5 Evaluation

In this section, we evaluate the security and performance of our SecPod prototype. All the experiments were conducted on a physical machine with a 2.5 GHz Intel Core i5 CPU and 8 GB of memory. The host system runs Ubuntu 12.04 LTS with a kernel version of 3.11.0. The guest is configured with 2 GB of memory, and runs Ubuntu 12.04 LTS Server with a kernel version of 3.10.32.

5.1 Security Analysis

We first evaluate the security guarantee of SecPod by analyzing how SecPod can prevent various attacks. We organize these attacks from four perspectives: memory isolation violation, instruction misuse, malicious address translation redirection and malicious devices, with a focus on the first three. Malicious devices can subvert the secure space (and the hypervisor) via DMA attacks. This can be prevented using IOMMU. Due to the space limit, we did not report the security evaluation for the DMA attack. Curious readers could refer to [37] for more details.

Memory Isolation Violation. A key requirement of SecPod is to strictly isolate the security tool from the vulnerable kernel. This isolation is enabled by the synergy of SecPod's two key techniques: paging delegation and execution trapping. The first category of attacks attempts to maliciously modify the secure space memory. Because the secure space memory is not mapped in the normal space (except the entry and exit gates), the attacker cannot directly change it. Instead, the attacker has to map the secure space memory into the normal space directly or by tricking the secure space to do so. Both attacks are prevented in SecPod. *First*, the kernel delegates its paging operations to the secure space. Its own page tables are never put in effect as prevented by execution trapping. Shadow page tables in the secure space are not directly accessible by the compromised kernel either. *Second*, the kernel might request SecPod to map the secure space memory to the normal space. This is foiled by SecPod's page table update validation which enforces the normal/secure space isolation. Specifically, it disallows the normal space from mapping any physical pages of the secure space, and protects both the virtual address and the physical content of the entry and exit gates. Note that the attacker could write any values into the shared memory page. However, he cannot deceive the secure code to unmap the secure space or modify the kernel memory mapping because the secure code checks the memory mapping operations against malicious mapping modifications. While an attacker could write junk information to the shared memory and make it full, leading to a deny-of-service attack. The SecPod secure code could detect the frequent bad requests and notify the hypervisor of the anomalous events.

Instruction Misuse. The second category of attacks tries to subvert the secure space by misusing existing instructions. No new code can be injected to the kernel as SecPod enforces $W \oplus X$ for the kernel, but code reuse attacks like return-oriented programming (ROP) [30] may still succeed due to the lack of control flow integrity [25]. In addition, the kernel still has the required right to execute privileged instructions. For example, it could load a crafted page table that allows manipulating the secure space. We address this type of attacks by trapping and vetting the execution of critical instructions by the kernel, such as `MOV to CR3` (Table 1). SecPod ensures that loading a page table other than the two legitimate ones will be trapped and denied. It also protects the associated data structures for instructions like `LGDT`. Since the kernel cannot load arbitrary page tables, it might try to enter the secure space with interrupts enabled. This can be achieved through the entry gate, for example, by skipping the first `CLI` and triggering an interrupt right before the second `CLI`. The CPU would then execute the interrupt handler in the secure space. Our design can foil this attack because the interrupt handler is not executable as soon as the CPU switches to the secure space. Nevertheless, this might cause the virtual CPU to halt because of the non-executable interrupt handler. The attack can also be launched with the return-oriented programming. Normally, as soon as the CPU enters the secure space, the kernel code becomes non-executable and the ROP program cannot continue. However, there is a subtle case in which the ROP program switches to gadgets in the secure space upon entering it (the secure space has its own code and stack data, which satisfies the basic requirement of conducting a ROP attack). By doing so, the program can continue running across the context switch because the attacking stack is mapped in the secure space. This attack overall is hard to use because the secure space might not contain enough useful gadgets. It can also be mitigated by applying existing ROP defenses to the secure space, such as control flow integrity [25], code randomization [2], and systematic removal of gadgets [5].

Address Translation Redirection Attack (ATRA). Jang et al. proposed an attack that can bypass the hardware memory monitor by directly modifying the page table related kernel objects [38]. Specifically, they wrote a kernel rootkit to modify the page table entries and redirect the page table pointers to the compromised ones (i.e., the maliciously constructed page table objects). By doing so, the actual kernel memory used by the system is redirected to a place that is not monitored by the hardware. While with SecPod's paging delegation, all the effective kernel page tables are maintained in the secure space. Any malicious attempts to modify the page tables will be intercepted and vetted by the secure code. Specifically, the SecPod secure code will notice that there is no corresponding page table entry in the SPT pool if someone has launched a *memory-bound ATRA*. Moreover, the attacker cannot directly modify the CR3 register value

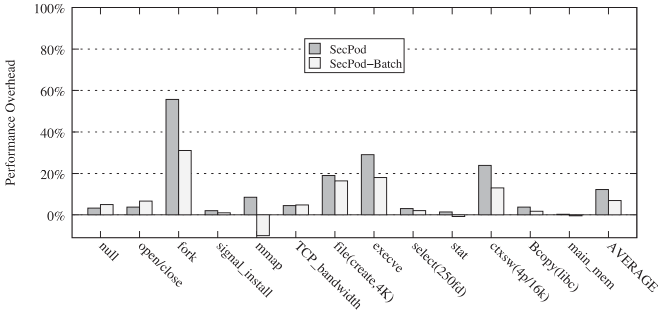


Fig. 8. LMBench overhead.

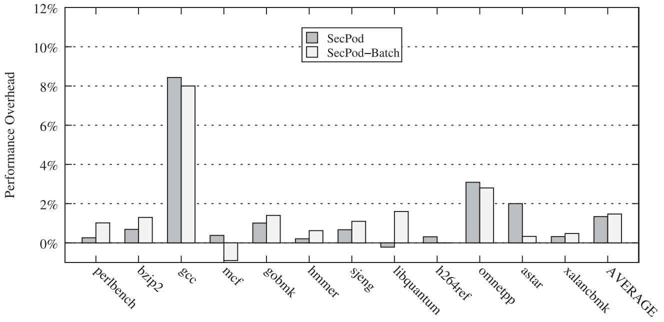


Fig. 9. SPEC INT2006 overhead.

(i.e., the *CR3-ATRA*) as SecPod's execution trapping traps every unintended CR3 loading (the intended, benign CR3 values are maintained in the CR3-target list as we described in Section 3.2.2). The authors also discussed the difficulty to prevent ATRA attack on hypervisor-based virtual machine introspection system (Appendix A.2, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TDSC.2017.2675991, in [38]). They suggest that "NPT allows guest kernel to modify its page tables ...and the removal of the write-protection in guest page tables would make the mitigation of ATRA more difficult". The design of SecPod securely re-enables those hypervisor-based introspection systems to run on the hardware with NPT.

Synthetic Attack. To further validate the security of SecPod, we create a synthetic kernel rootkit that hooks the system call table to intercept system calls like `sys_read` and `sys_mkdir`. Our experimental security tool can detect the loading of the malicious rootkit because its hash is not in the list of hashes of benign code pages. Even without this tool, SecPod can detect the rootkit's attempts to modify the (read-only) system call table-the rootkit calls a kernel function to make the syscall table writable. This request is forwarded to the secure space and subsequently denied because the secure space does not allow the syscall table to be changed.

5.2 Performance Evaluation

To evaluate the performance of SecPod, we experimented with micro-benchmarks, SPEC INT2006, and application benchmarks. The micro-benchmarks measure SecPod's impact to fine-grained operations, such as system calls; SPEC INT2006 quantifies the performance overhead introduced by SecPod for CPU- and memory-intensive workloads; and the (I/O-intensive) application benchmarks measure the overall system performance under SecPod. All the experiments were repeated three times and the average results are reported, except for ApacheBench which was repeated ten times. SPEC INT 2006 was measured in the reportable run mode. We noticed that the deviation is negligible for most of these experiments. Apachebench has a just slightly higher deviation than others that is reported as error bars in Fig. 10. We compared the performance of SecPod, with and with-out page table update batching (for brevity, we call page table update batching just batching), to that of an unmodified VM backed by the nested paging (the baseline). SecPod's VM is also backed by the nested paging. However, its paging operations are expected to be less efficient than the baseline because they are delegated to the secure space. Note that SecPod relies on a per-VM (guest OS) secure space for the paging delegation. Thus the performance degradation will not spread across the VMs, so that SecPod could be easily scaled. Even though we did not compare the performance of SecPod to that of the VMs backed by shadow paging, previous benchmarks demonstrate that Intel EPT provides substantial performance gains over shadow paging for most tested benchmarks. For example, Intel EPT can achieve an acceleration of up to 48 percent for MMU-intensive benchmarks [15].

5.2.1 Micro-Benchmarks

Fig. 8 shows the performance overhead of SecPod for LMBench. LMBench is a set of benchmarks to measure the system call performance. Our prototype incurs less than 5 percent overhead for most of the system calls LMBench tests, such as `open`, `close`, `signal_install`, and `stat`. These system calls do not contain operations that require services from the secure space. Consequently, the impact of SecPod over these system calls is minimal. The performance degrade is probably caused by normal task switches (of other processes) during the tests. On the other hand, system calls that involve page table operations suffer most. Particularly, `fork` has the highest overhead (56 percent) for the SecPod prototype without batching, followed by `execve`, `mmap`, `file creation`, and `context switch` (all at around 20 percent). Most of these system calls involve heavy page table operations. For example, the `fork` system call creates a child process that duplicates the parent process's address space (with copy-on-write) [32], and each task switch in SecPod requires an extra loading of the SecPod page table (Section 3.2.2). Batching can significantly reduce this overhead by committing updates in batches (Fig. 8). For example, the overhead of the `fork` system call was reduced from 56 to 31 percent. In addition, the `mmap` test case has slightly lower latency than even the baseline. We suspect that batching can reduce the times TLB is flushed. On average, SecPod and SecPod with batching introduce about 12 and 7 percent performance overhead for LMBench, respectively.

5.2.2 SPEC INT2006

We measured the performance overhead of SecPod with the SPEC INT2006 benchmark suite. SPEC INT2006 consists of several CPU-intensive benchmarks, stressing the system's processor and memory subsystems. The results are reported in Fig. 9. For the majority of these benchmarks, SecPod has

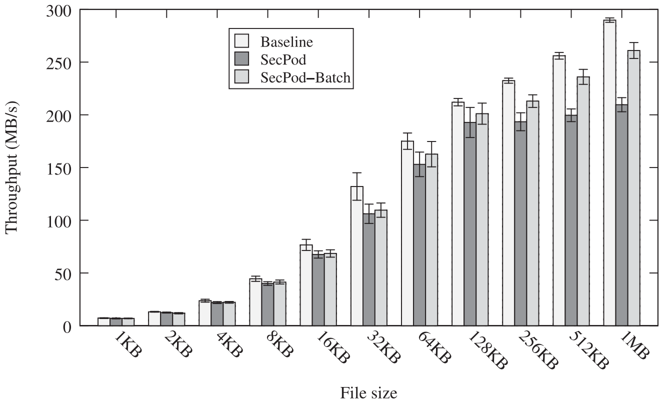


Fig. 10. Apache bench throughput.

the similar performance (less than 2 percent overhead) as the baseline Linux, with or without batching. This result is expected because those benchmarks are CPU-intensive without large memory footprints. The high overhead related to process creation (e.g., `fork`) is amortized by the relatively long execution time of those benchmarks. The `403.gcc` benchmark has the highest overhead for SecPod, at around 8 percent. This benchmark is based on gcc version 3.2. It has 9 input workloads (i.e., pre-processed C code). Most of these workloads have large and fluctuating memory footprints, ranging from 300 to 800 MB.[3] For example, input `s04.i` and `g23.i` both use more than 800 MB of memory at the peak, and about 100 and 250 MB of memory at the bottom, respectively. Naturally, this kind of workloads can lead to higher performance overhead for SecPod. On average, SecPod introduces about 1.3 percent performance overhead for the SPEC INT 2006 benchmark suite.

5.2.3 Application Benchmarks

To measure SecPod's impact on the overall system performance, we experimented with two server side benchmarks (ApacheBench and SysBench) and one desktop application (firefox Kraken benchmark). ApacheBench is a program to measure how fast the system can process web traffic. In this experiment, we run the Apache server (2.2.22) in the VM, and ApacheBench on the host Linux. Fig. 10 shows the throughput of the Apache server with regard to different file sizes (from 1 KB to 1 MB). Each file was generated by collecting random data from the `/dev/random` device. For file sizes up to 16 KB, the overhead of SecPod is less than 9 percent and increases to about 22 percent for 512 KB files and 27 percent for 1 MB files. When the file size increases, the kernel needs to allocate and update the page table more frequently to accommodate frequent file accesses, leading to a relatively high performance overhead. In comparison, SecPod with batching has much lower performance overhead than that without batching. For example, batching reduces the overhead from 22 percent to just 7.8 percent for file size 512 KB, and from 27 percent to just 10 percent for file size 1 MB. Overall, batching is an effective performance optimization to SecPod for memory-intensive applications.

SysBench is a suite of multi-threaded benchmarks to evaluate the performance of a database system under intensive workloads. We use SysBench to measure SecPod's impacts on the file I/O and the MySQL processing. Both experiments are repeated with many different numbers of threads. In the file I/O experiment, we measure the throughput using 128 files (1 GB in total) and a block size of 16 KB. The results are shown in Fig. 11. The largest overhead is 2.4 percent. We also measure the MySQL performance with SysBench's online transaction processing (OLTP) benchmark. Specifically, we build a MySQL database with 1,000,000 entries and query the database using various numbers of threads. The results are shown in Fig. 12. The performance loss is in the range of 2.3 to 8.5 percent with an average of 5 percent. Additionally, SysBench is an I/O-intensive benchmark. Batching thus could only slightly improve the performance of SecPod.

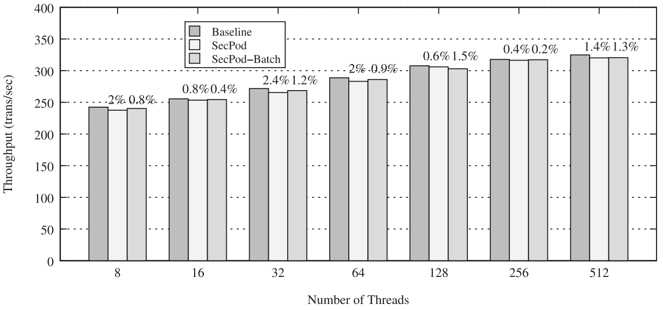


Fig. 11. Throughput of SysBench FileIO.

To measure SecPod's impact on desktop applications, we run the Kraken benchmark with the firefox browser. Kraken is a browser benchmark testing the Javascript performance, such as AI computing, audio, and image processing. The results are reported in Fig. 13 (normalized to the baseline NPT-backed Linux VM). For most of the test cases, SecPod introduces less than 2 percent performance overhead, except the audio processing test case which has a higher overhead (about 7 percent). This is likely because the audio processing test includes a loop that handles a big chunk of memory,[4] which may incur more frequent page table updates. For CPU-intensive tests like AI and stanford crypto, SecPod with batching does not have significant performance advantage over SecPod without batching. But we could still observe the relatively large performance benefit for memory-intensive tests like image processing and json.

6 Related Work

Virtualization-Based Security. The first category of the related work is a long stream of virtualization-based security systems with diverse focuses, such as malware analysis [39], virtual honeypot [40], kernel rootkit detection and prevention [5], [41]. In particular, virtualization has been applied often in the context of virtual machine introspection. Livewire pioneers the concept of "out-of-VM" introspection to understand the in-VM states and activities by parsing the raw VM resources [42]. Semantic gap is one of the main challenges for VMI systems because VMI aims at semantically inferring the in-VM activities and states from the raw VM data (e.g., memory, disk). A number of recent systems try to address this challenge from different perspectives [4],

[3] http://hpc.aut.uah.es/informes/TR-HPC-01-2009.pdf

[4] https://wiki.mozilla.org/Kraken_Info

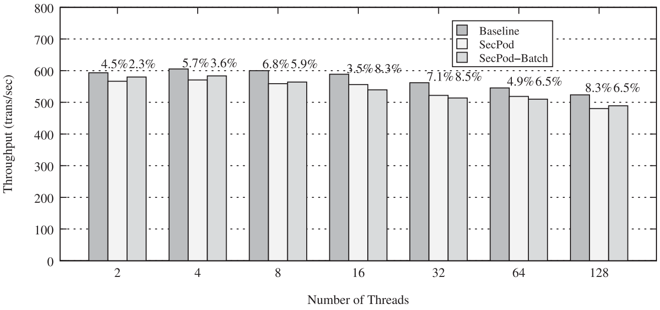


Fig. 12. Throughput of SysBench OLTP.

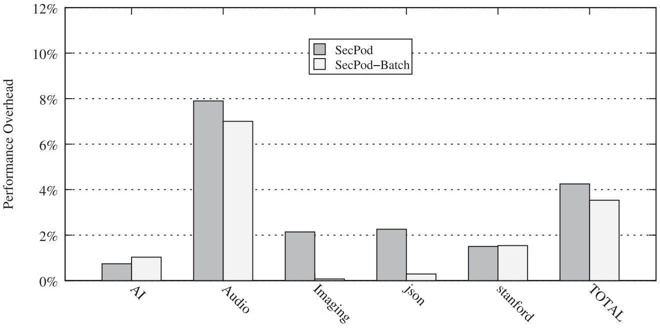


Fig. 13. Firefox kraken benchmark overhead.

[9], [43], [44]. For example, Virtuoso [43] can effectively automate the process of building introspection-based security tools. SIM is the most closely related system. It firstly leverages the CR3 target-list to effectively and efficiently turn out-of-VM monitoring in-VM. SIM is a monitoring framework while SecPod targets at supporting generic virtualization-based systems. Particularly, SecPod creates a trusted execution environment for the security tool by combining two key techniques, paging delegation and execution trapping. In addition, SecPod uses the CR3 target-list differently to support the nested paging (Section 3.2.2). VMI systems can be integrated with and benefit from SecPod's code integrity guarantee and fine-grained page table monitoring.

Virtualization is also a popular choice of platforms to enhance the kernel or application security [6], [7], [11], [14], [45]. For example, Overshadow is designed to protect the secrecy of the user data even if the kernel is completely compromised [45]. Patagonix protects the kernel code integrity through virtualization-based code identification [6]. HookSafe addresses the protection granularity problem through systematic hook redirection [11]. Most of these systems require a reliable kernel code integrity. Otherwise, an attacker could subvert their protection by injecting malicious code. SecPod is an ideal platform for these systems. Security tools in SecPod are strictly isolated from the vulnerable kernel, but still have the visibility of an in-kernel tool. As a proof-of-concept, one of our implemented security tools for SecPod tries to prevent the unauthorized code from executing in the kernel. This provides a security guarantee similar to Patagonix [6] and NICKLE [7] (Section 4.2).

Virtualization-based systems, including SecPod, assume that the hypervisor is trusted due to its smaller code base and attack surface. However, the bloated code base of modern hypervisors and recent attacks put this assumption into question. There have been a series of recent efforts in protecting the hypervisor integrity, via formal verification [17], [46], security enhancements [19], and size reduction and disaggregation [18], [47]. These systems can be naturally integrated with SecPod to provide a strong foundation of security.

Kernel/User Application Security. The second category of related work includes a large number of research efforts in the kernel and user application security. Address space layout randomization [48] and data execution prevention [3] are two popular exploit mitigation mechanisms in modern kernels. These kernel-level protection schemes suffer from the pitfall that the page table is not protected from exploits. SecPod reliably enforces DEP for the kernel. ASLR and DEP could be bypassed mainly by return-oriented-programming. Control flow integrity is an effective defense against most control flow attacks, including ROP, by mandating that run-time control flow must follow the program's control flow graph [25]. Most of the previous CFI systems target user applications. They rely on the kernel to provide the necessary memory protection of the code and read-only data. Recent efforts to adapt CFI to the kernel turn to virtualization for essential supports [49]. For example, KCoFI [49] leverages the Secure Virtual Architecture [50] to interpose the software and hardware interactions. All software, including the kernel, is compiled to the virtual instruction set of SVA. Kernel CFI can also be support by SecPod as it provides both strong isolation and reliable memory protection for security tools. There is also a series of prior efforts in implementing software fault isolation (SFI) [31], [51], [52]. SFI aims at confining untrusted code in a host application. For example, Native Client [31] uses two layers of sandboxes to safely run untrusted native plugins in a web browser. SFI technologies have been utilized to isolate untrusted device drivers in the kernel [14], [21], [51].

TZ-RKP [53], HyperSafe [19], and nested kernel [54] are three closely related systems. TZ-RKP leverages the ARM TrustZone to protect the kernel running in the normal world. Specifically, it instruments the kernel to prevent it from executing certain privileged instructions or updating page tables. These operations instead must be handled by the secure world. However, the instrumentation-based instruction access control of TZ-RKP is not directly applicable to the x86 architecture because x86 has variable instruction lengths and thus unintended privileged instructions can be created out of the existing ones [30]. This problem can be solved by adopting the techniques of NaCl [31]. HyperSafe write-protects the hypervisor page table and uses the x86 write-protect (WP) bit to allow benign page table updates. It further enforces the control flow integrity [25] to prevent that from being bypassed. Nested kernel similarly protects page tables for the OS kernel, but enforces the kernel code integrity and removes unintended privileged instructions from the kernel code (instead of enforcing CFI). SecPod also controls the guest page table updates though paging delegation, but its design revolves around the goal to provide security tools with an extensible framework that is not only compatible with the recent virtualization hardware, but also allows them to intercept key events in the guest kernel. For example, the separation of the normal and secure spaces isolates security tools from the untrusted kernel and simultaneously enables an easy access to the kernel data. Recently, Intel introduced a security enclave technique called Software Guard Extension (SGX). This trusted executing environment can prevent even the privileged system software (e.g., hypervisor, OS kernel, and BIOS) from compromising the enclave's code and data. We believe SGX could also be used to provide an isolated executing environment for critical system services as SecPod does.

Optimizing Boundary Crossing Calls. The third category of related work consists of systems that use batching to optimize boundary crossing calls (e.g., system calls). SecPod delegates page table updates to the secure space. Each page table update thus requires a relatively expensive round-trip between the normal space and the secure space (though it is still much faster than a world switch). We reduce this overhead by caching some page table updates and committing them in batches. Batching has been used for similar purposes in existing works such as the exception-less system call [26], [55]. For example, Cassyopia is a compiler-assisted system call optimization technique [55]. It proposes the idea of system call clustering to reduce the number of boundary crossing required [55]. FlexSC conducted a detailed study of the system call cost, and proposes the exception-less system call [26]. Specifically, the application stores system call requests in a "system call page", and the kernel uses a dedicated kernel thread to asynchronously process those requests. Similarly, sophisticated hypervisors like Xen [13] and KVM [12] support out-of-sync shadow paging in which shadow page tables are synchronized with guest page tables when necessary. For example, Xen shadow paging synchronization has implemented a *copy-on-write* alike mechanism [56]. The Xen hypervisor temporarily removes the write permission of the guest page tables while maintaining an out-of-sync page list, and resynchronizes the page list on page faults or context switches. Meanwhile, SecPod is built as a security framework. The primary purpose of the secure space is to maintain the security properties for the kernel memory. Thus we use a RPC-like (remote procedure call) mechanism. Moreover, SecPod does not need to have complete memory virtualization support; it simply maintains an identical shadow copy of the kernel page tables. Consequently, SecPod SPT batching can use a much simpler design; i.e., it caches all the page table update requests in a buffer and late submits it.

7 Summary

We have presented the design, implementation, and evaluation of SecPod, a practical and extensible framework for virtualization-based security systems. SecPod provides a trusted execution environment for security tools. They are not only strictly isolated from the vulnerable kernel, but also have full visibility into it. Particularly, any updates to the guest's page tables can be intercepted and regulated by these tools, allowing the fine-grained control over the guest kernel's memory protection. By using the in-VM shadow paging, SecPod is fully compatible with the recent advances in the hardware virtualization support, particularly the nested paging.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments. This work was partly supported by US National Science Foundation (NSF) under Grant No. 1453020, National Science Foundation of China under Grant No. 61672421, 61402358, and PhD Programs Foundation of Ministry of Education of China under Grant No. 20120201110010. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of these agencies. Part of the work was done when the first author was visiting Florida State University. Yong Qi and Zhi Wang are corresponding authors.

References

[1] CVE Database, "Common vulnerabilities and exposures database," [Online]. Avialable: http://www.cvedetails.com/

[2] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *Proc. IEEE Symp. Secur. Privacy*, 2014, pp. 276–291.

[3] Data execution prevention. [Online]. Available: http://en.wikipedia.org/wiki/Data_Execution_Prevention

[4] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection and monitoring through VMM-based "out-of-the-box" semantic view reconstruction," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 2, 2010, Art. no. 12.

[5] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram, "Defeating return-oriented rootkits with "Return-Less" kernels," in *Proc. 5th ACM Eur. Conf. Comput. Syst.*, Apr. 2010, pp. 195–208.

[6] L. Litty, H. A. Lagar-Cavilla, and D. Lie, "Hypervisor support for identifying covertly executing binaries," in *Proc. 17th USENIX Secur. Symp.*, Jul. 2008, pp. 243–258.

[7] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing," in *Proc. 11th Recent Advances Intrusion Detection*, Sep. 2008, pp. 1–20.

[8] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," in *Proc. 21st ACM SIGOPS Symp. Operating Syst. Principles*, Oct. 2007, pp. 335–350.

[9] M. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-VM monitoring using hardware virtualization," in *Proc. 16th ACM Conf. Comput. Commun. Secur.*, Nov. 2009, pp. 477–487.

[10] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu, "Process out-grafting: An efficient "out-of-VM" approach for fine-grained process execution monitoring," in *Proc. 18th ACM Conf. Comput. Commun. Secur.*, 2011, pp. 363–374.

[11] Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering kernel rootkits with lightweight hook protection," in *Proc. 16th ACM Conf. Comput. Commun. Secur.*, Nov. 2009, pp. 545–554.

[12] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: The Linux virtual machine monitor," in *Proc. Ottawa Linux Symp.*, Jun. 2007, pp. 225–230.

[13] P. Barham, et al., "Xen and the art of virtualization," in *Proc. 19th ACM Symp. Operating Syst. Principles*, Oct. 2003, pp. 164–177.

[14] A. Srivastava and J. Giffin, "Efficient monitoring of untrusted kernel-mode execution," in *Proc. 18th Annu. Netw. Distrib. Syst. Secur. Symp.*, Feb. 2011, pp. 115–133.

[15] VMware, "Performance evaluation of Intel EPT hardware assist," [Online]. Available: https://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf

[16] Trusted Boot project, "Trusted boot," [Online]. Available: http://tboot.sourceforge.net/

[17] G. Klein, et al., "seL4: Formal verification of an operating-system kernel," *Commun. ACM*, vol. 53, no. 6, pp. 107–115, 2010.

[18] J. Szefer, E. Keller, R. B. Lee, and J. Rexford, "Eliminating the hypervisor attack surface for a more secure cloud," in *Proc. 18th ACM Conf. Comput. Commun. Secur.*, Oct. 2011, pp. 401–412.

[19] Z. Wang and X. Jiang, "HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *Proc. 31st IEEE Symp. Secur. Privacy*, May 2010, pp. 380–395.

[20] Z. Wang, C. Wu, M. Grace, and X. Jiang, "Isolating commodity hosted hypervisors with HyperLock," in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, Apr. 2012, pp. 127–140.

[21] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," in *Proc. 19th ACM Symp. Operating Syst. Principles*, Oct. 2003, pp. 207–222.

[22] Wikipedia, "DMA attack," [Online]. Available: http://en.wikipedia.org/wiki/DMA_attack

[23] P. M. Chen and B. D. Noble, "When virtual is better than real," in *Proc. 8th Workshop Hot Topics Operating Syst.*, 2001, Art. no. 133.

[24] *Intel 64 and IA-32 Architectures Software Developerś Manual, Intel, Santa Clara, CA, USA*, Feb. 2014.

[25] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, 2009, Art. no. 4.

[26] L. Soares and M. Stumm, "FlexSC: Flexible system call scheduling with exception-less system calls," in *Proc. 7th Symp. Operating Syst. Des. Implementation*, 2010, pp. 33–46.

[27] A JIT for packet filters. [Online]. Available: https://lwn.net/Articles/437981/

[28] P. Kedia and S. Bansal, "Fast dynamic binary translation for the kernel," in *Proc. 24th ACM Symp. Operating Syst. Principles*, 2013, pp. 101–115.

[29] C. Song, C. Zhang, T. Wang, W. Lee, and D. Melski, "Exploiting and protecting dynamic code generation," in *Proc. Symp. Netw. Distrib. Syst. Secur.*, 2015, pp. 546–561.

[30] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proc. 14th ACM Conf. Comput. Commun. Secur.*, Oct. 2007, pp. 552–561.

[31] B. Yee, et al., "Native client: A sandbox for portable, untrusted x86 native code," in *Proc. 30th IEEE Symp. Secur. Privacy*, May 2009, pp. 79–93.

[32] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*. Hoboken, NJ, USA: Wiley, 2012.

[33] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*. Sebastopol, CA, USA: O'Reilly, 2005.

[34] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Mateo, CA, USA: Morgan Kaufmann, 2012.

[35] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kGuard: Lightweight kernel protection against return-to-user attacks," in *Proc. 21st USENIX Conf. Secur. Symp.*, 2012, pp. 39–39.

[36] A. Baliga, V. Ganapathy, and L. Iftode, "Detecting kernel-level rootkits using data structure invariants," *IEEE Trans. Depend. Secure Comput.*, vol. 8, no. 5, pp. 670–684, Sep./Oct. 2011.

[37] A. Markuze, A. Morrison, and D. Tsafrir, "True IOMMU protection from DMA attacks: When copy is faster than zero copy," in *Proc. 21st Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2016, pp. 249–262.

[38] D. Jang, H. Lee, M. Kim, D. 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.

[39] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," in *Proc. 15th ACM Conf. Comput. Commun. Secur.*, 2008, pp. 51–62.

[40] X. Jiang and X. Wang, ""Out-of-the-Box" monitoring of VM-based high-interaction honeypots," in *Proc. 10th Int. Conf. Recent Advances Intrusion Detection*, 2007, pp. 198–218.

[41] R. Riley, X. Jiang, and D. Xu, "Multi-aspect profiling of kernel rootkit behavior," in *Proc. 4th ACM SIGOPS Eur. Conf. Comput. Syst.*, Apr. 2009, pp. 47–60.

[42] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proc. 10th Netw. Distrib. Syst. Secur. Symp.*, Feb. 2003, pp. 191–206.

[43] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Proc. IEEE Symp. Secur. Privacy*, 2011, pp. 297–312.

[44] Y. Fu and Z. Lin, "Space traveling across VM: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection," in *Proc. IEEE Symp. Secur. Privacy*, 2012, pp. 586–600.

[45] X. Chen, et al., "Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems," in *Proc. 13th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2008, pp. 2–13.

[46] T. Murray, et al., "seL4: From general purpose to a proof of information flow enforcement," in *Proc. IEEE Symp. Secur. Privacy*, 2013, pp. 415–429.

[47] P. Colp, et al., "Breaking up is hard to do: Security and functionality in a commodity hypervisor," in *Proc. 23rd ACM Symp. Operating Syst. Principles*, Oct. 2011, pp. 189–202.

[48] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *Proc. 21st USENIX Conf. Secur. Symp.*, 2012, pp. 40–40.

[49] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete control-flow integrity for commodity operating system kernels," in *Proc. IEEE Symp. Secur. Privacy*, 2014, pp. 292–307.

[50] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve, "Secure virtual architecture: A safe execution environment for commodity operating systems," in *Proc. 21st ACM SIGOPS Symp. Operating Syst. Principles*, 2007, pp. 351–366.

[51] U. Erlingsson, S. Valley, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: Software guards for system address spaces," in *Proc. 7th USENIX Symp. Operating Syst. Des. Implementation*, Nov. 2006, pp. 75–88.

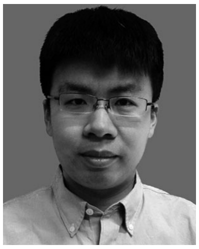
[52] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proc. 14th ACM Symp. Operating Syst. Principles*, Dec. 1993, pp. 203–216.

[53] A. M. Azab, et al., "Hypervision across worlds: Real-time kernel protection from the ARM TrustZone secure world," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 90–102.

[54] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve, "Nested kernel: An operating system architecture for intra-kernel privilege separation," in *Proc. 20th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2015, pp. 191–206.

[55] M. Rajagopalan, S. K. Debray, M. A. Hiltunen, and R. D. Schlichting, "Cassyopia: Compiler assisted system optimization," in *Proc. 9th Conf. Hot Topics Operating Syst.*, 2003, pp. 103–108.

[56] Xen 3.3 feature: Shadow 3. [Online]. Available: https://blog.xenproject.org/2008/08/27/xen-33-feature-shadow-3/

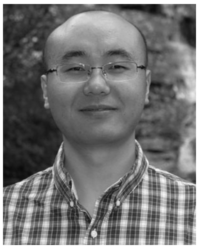


Xiaoguang Wang received the bachelor's degree in software engineering from Northwestern Polytechnic University, China, in 2010. He is currently working toward the PhD degree in the Department of Computer Science, Xi'an Jiaotong University. His research interests include system security, operating system, and virtualization.

Yong Qi received the PhD degree in computer software and theory from Xian Jiaotong University, in 2001. He is currently a professor in the School of Electronic and Information Engineering, Xian Jiaotong University and the director of the Institute of Computer Software and Theory. His research interests include operating system, distributed system, pervasive computing, and security in cloud computing. He is a member of the IEEE.

Zhi Wang received the BS degree in computer science from Xi'an Jiaotong University, in 1999 and the PhD degree in computer science from North Carolina State University, in 2012. He is currently an assistant professor of computer science with Florida State University. His research interests mainly include operating system security, cloud computing security, and mobile security.

Yue Chen received the bachelor's degree in information security from the Harbin Institute of Technology, in 2011 and the master's degree in computer science from Northeastern University, in 2013. He is currently working toward the PhD degree in computer science at Florida State University. His current research interest is system security.



Yajin Zhou received the bachelor's degree in computer science from Suzhou University, China, in 2003 and the PhD degree in computer science from North Carolina State University, in 2015. He is currently a senior researcher with Qihoo 360. His research interests include operating system security and smart phone security.