SecFortress: Securing Hypervisor using Cross-layer Isolation

***Abstract*—Virtualization is the corner stone of cloud computing, but the hypervisor, a crucial software component that enables virtualization, is known to suffer from various attacks. It is challenging to secure the hypervisor due to at least two reasons. On one hand, commercial hypervisors are usually integrated into a privileged Operating System (OS), which brings in a larger attack surface. On the other hand, multiple Virtual Machines (VM) share a single hypervisor, thus a malicious VM could leverage the hypervisor as a bridge to launch "cross-VM" attacks. In this work, we propose SecFortress, a dependable hypervisor design that decouples the virtualization platform into a mediator, an outerOS, and multiple HypBoxes through a cross-layer isolation approach. SecFortress extends the nested kernel approach to deprivilege the host OS from accessing the hypervisor's memory and creates an isolated hypervisor instance, HypBox, to confine the impacts from the untrusted VMs. We have implemented SecFortress based on KVM and evaluated its effectiveness and efficiency through case studies and performance evaluation. Experimental results show SecFortress can significantly improve the security of the hypervisor with negligible runtime overhead.**

I. Introduction

As the corner stone of cloud computing, virtualization plays a leading role in facilitating cloud computing services. A hypervisor, the most critical software component in virtualization, provides resource management and the support of running multiple VMs on a single physical server. However, modern hypervisors face severe security threats mainly from two aspects. Firstly, the hypervisor takes the untrusted VM states as input. For instance, VM escape vulnerabilities can be exploited by malicious VMs to execute arbitrary code inside the hypervisor, placing the entire virtualization platform at risk. Moreover, the monolithic design of controlling multiple VMs exacerbates the problem of cross-VM attacks. Secondly, the hypervisor usually involves a much more vulnerable but privileged OS as a virtualization assistant. For example, KVM [1] is integrated into Linux kernel and leverages its functionalities to improve the efficiency. Compared with the hypervisor however, Linux kernel is much more complicated and vulnerable, thus resulting in a larger attack surface.

Existing approaches try to address the above problems based on "hardening" or "isolation". The former attempts to strengthen the hypervisor with various techniques, e.g., protecting the control flow integrity [2], [3], minimizing the hypervisor [4]–[6] and decomposing the hypervisor [7], [8]. However, the hardening approach suffers from two limitations, unable to completely eliminate the vulnerabilities and unusable hypervisor due to the removal of essential features. The former leverages the isolated or encrypted memory to "deprive" the hypervisor of the privilege to manage VMs' resources, such as nested hypervisor [9], [10], TEE-based [11], [12] and VM-encrypted [13], [14] solutions. However, the isolation approach usually involves additional VM exits or memory context switches, resulting in significant performance overhead. Moreover, some works (e.g., [10]) need instrumentation inside the guest OS kernel, impossible for commodity OSes, while others (e.g., [14]) cannot defeat attacks from malicious VMs.

In this paper, we propose SecFortress, a cross-layer isolation mechanism based on the principle of least privilege, significantly reducing the trusted computing base (TCB) of virtualization platforms. In particular, SecFortress partitions the virtualization layer into a tiny privileged *mediator* (the only TCB in our system), a large untrusted *outerOS*, and multiple *HypBox* instances. The mediator is a tamper-proof isolated context used to control the system resources and the memory management unit (MMU) to interpose all memory mapping updates, ensuring the strong isolation between outerOS and HypBoxes. The mediator extends the nested kernel [15] by adding security services, including memory protection, instruction protection, and enforced control flow to protect the confidentiality and integrity of each component. Meanwhile, the mediator is built in the same privilege level as the outerOS and HypBoxes for performance consideration. Moreover, SecFortress establishes a separated hypervisor runtime instance called HypBox for each VM, and confines their operations through the mediator. A HypBox contains per-instance private data and duplicated hypervisor code. The mediator strictly controls the memory layout of each HypBox instance through monitoring all memory operations, enforcing cross-VM data confidentiality and control-flow integrity. Therefore, even if a malicious VM attacks its HypBox instance by exploiting its vulnerabilities, it cannot further affect other VMs.

We implemented the prototype of SecFortress based on KVM with moderate modification to the Linux kernel. The TCB of our system is around 2K LOC (Lines of Code). We analyzed 20 hypervisor vulnerabilities in detail, and evaluated SecFortress against the corresponding exploits. We find that most of these vulnerabilities cause memory problems (e.g., information leakage or memory corruption), and the root cause of these problems is the lack of a proper memory isolation among the VMs. Our evaluation results show that SecFortress can defeat the above exploits with negligible overhead.

Contributions. The contributions of this paper are summarized as below:

- We design SecFortress, a cross-layer isolation approach that enhances the hypervisor runtime security by isolating the host OS from the hypervisor and confining untrusted VMs into a per-VM hypervisor instance.
- We minimize the TCB of the entire virtualization layer by moving the host OS out of the TCB and limiting the hypervisor's capabilities.
- We developed a prototype of SecFortress on KVM and evaluated it comprehensively based on 20 real world hypervisor

vulnerabilities. Our evaluation results show that SecFortress can defeat the above vulnerabilities with negligible overhead.

II. Background

A. Hypervisors

The hypervisor is the most crucial part of the virtualization software stack. Hypervisors can be classified into bare-metal hypervisors and hosted hypervisors. Hosted hypervisors (the focus of this paper) are usually integrated into traditional operating systems, which are responsible for managing physical resources, but do not have virtualization capabilities. Instead, virtualization is supported by hosted hypervisors, which are loaded as kernel modules. Typically, hosted hypervisors access physical resources by calling the host operating system's services to achieve CPU, memory, and I/O virtualization. For example, KVM is a hosted hypervisor based on the Linux OS and requires hardware-assisted virtualization support. The management of physical resources remains the responsibility of Linux, so KVM calls some specific Linux functions to implement CPU and memory virtualization. QEMU [16] provides I/O virtualization, and manages the life cycle of KVM VMs. Each VM runs as an individual QEMU process in Linux.

B. Nested Kernel

The nested kernel approach [15] embeds a small isolated kernel within a monolithic kernel. By intercepting all updates to virtual address translation to protect physical memory, nested kernel can significantly reduce the TCB in a complicated system. In addition, nested kernel proposes a new MMU virtualization approach to isolate the TCB from untrusted components, different than traditional approaches, e.g., using different CR3 base addresses or CPU privilege levels. The former approach isolates different contexts by creating multiple page tables. Transition between isolated contexts is achieved by switching the CR3 base address. However, this approach needs to flush TLB frequently, resulting in significant performance overhead. The latter approach isolates contexts using CPU privilege mechanism, such as nested virtualization [17], with the TCB usually running at a higher privilege than untrusted components. This approach also involves significant performance overhead because of context switch between different privilege levels. Unlike the above two approaches, the nested kernel monitors all modifications to the virtual-to-physical mapping and removes all sensitive instructions from the untrusted components. All memory operations in the untrusted components are restricted by the isolated kernel. Therefore, the nested kernel achieves logic isolation between the isolated kernel and untrusted components in the same address space using a single privilege level.

III. Overview

A. Motivation

Modern hypervisors are integrated into a privileged OS, so they not only handle VM exits from multiple VMs but also interact with the privileged OS. As a result, modern hypervisors face two kinds of threats: from the host mode and from the guest mode. On one hand, softwares running in the host mode mainly include a hypervisor, a privileged OS, and many host applications. Typically, the privileged OS and the hypervisor are located in the same address space, thus the privileged OS can access the hypervisor's memory arbitrary. Furthermore, the Common Vulnerabilities and Exposures [18] shows that the privileged OS contains more vulnerabilities than the hypervisor, which makes the former a perfect "bridge" to attack the latter. Hence, an effective approach to reduce the attack surface is to isolate the privileged OS from the hypervisor and deprivilege it. For instance, based on the microkernel design and ARM VE [19], HypSec [12] removes the host OS and most hypervisor's functions from the TCB.

On the other hand, modern hypervisors use numerous uncertified VM status as input. Malicious VMs may exploit vulnerabilities to compromise the hypervisor, and then control the entire virtualization platform. In addition to the programming vulnerabilities, a single point of failure design is also a reason for such a threat. Multiple VMs often share one hypervisor, which implies that once a malicious VM takes control of the hypervisor, it will be able to harm all others. Moreover, a malicious VM can perform the DoS attack on other VMs or the hypervisor, causing the entire system to crash or freeze. HyperLock [20] and DeHype [21] create a KVM instance for each VM to limit its impact to per-VM instance.

To the best of our knowledge, existing works either consider the threats from the host mode (e.g., HypSec [12]), or from the guest mode (e.g., HyperLock [20]), but no one considers both. We believe it is necessary to design a reliable cross-layer isolation mechanism, which not only isolates the privileged OS from the hypervisor but also limits the privileges of each hypervisor instance. Such isolation mechanism should effectively address both of these threats discussed above, which will be demonstrated in the paper. The hosted hypervisors are becoming more and more popular in the cloud market [22] due to their advantages of easy maintenance and lightweight. It is important to enhance hosted hypervisors' security.

B. Threat Model

We assume that the attacker can take control of the host OS by exploiting its vulnerabilities, and then hijack the hypervisor's control flow to compromise the confidentiality and integrity of the VMs. Meanwhile, the attacker can deliberately deploy a malicious VM to attack the hypervisor or the host OS through privilege escalation, and then compromise other VMs. The attacker can also take full control of an application in a normal VM, and try to attack the hypervisor by exploiting its vulnerabilities. We assume the cloud provider is trusted and the virtualization platform is trusted during booting up (e.g., via secure boot [23]). However, the platform may be compromised during the run-time. A VM deliberately leaking its sensitive data, side-channel attacks, and physical attacks are beyond the scope of this paper. Although side-channel attacks are not considered in our adversary model, SecFortress does not conflict with the existing defenses [24].

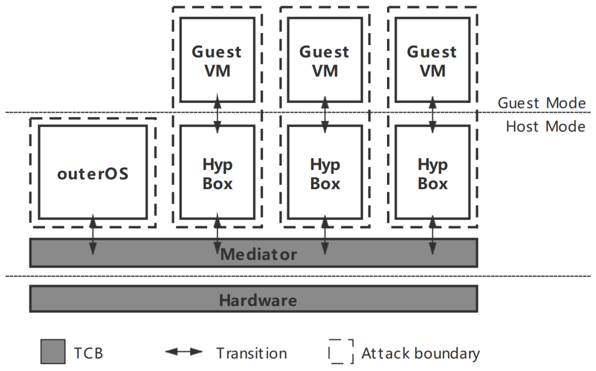

Fig. 1. Architecture overview of SecFortress.

C. System Overview

Figure 1 presents the architecture of SecFortress. We partition the virtualization layer into a mediator , an outerOS (the deprivileged host OS) , and multiple HypBoxes (hypervisor runtime instances). SecFortress minimizes the TCB of the virtualization platform to include hardware and the mediator (around 2K LOC) only, which makes formal verification possible. The mediator lies in parallel with the outerOS and hypervisor instances, but is isolated from them via virtual MMU memory protection mechanism. Meanwhile, to prevent the outerOS from accessing the hypervisor's memory or tampering with its control-flow, we design an isolated HypBox context through the CPU paging mechanism with memory access control and explicit instruction execution. In addition, each VM has its own HypBox, isolated from others, which is enforced by the mediator. All HypBoxes have the same code, but each has its own private data. To mitigate the single point of failure problem, SecFortress implements a security isolation mechanism to confine the control flow of the HypBox instance, and designs a series of policies to confine its memory access.

IV. Design

The core goal of SecFortress is to enhance the hypervisor against security threats from untrusted outerOS and VMs. To achieve this goal, we first remove the monolithic outerOS from the TCB by creating an isolated trusted context mediator that follows the principle of least privilege. Then we deploy a cross-layer isolation mechanism using the mediator between outerOS and HypBox, and among HypBoxes.

A. Mediator

1) Tamper-proof protection: The mediator is the most critical and fundamental component to SecFortress protection. If the mediator is compromised, all security mechanisms in the system will be invalidated. Thus we must ensure that it is tamper-proof. One general approach is to abstract an additional layer (either hardware or software) to run the mediator (e.g., [9], [12]), but it typically incurs high performance overhead. In order to run the mediator both securely and efficiently, we extend nested MMU [15], which creates a memory protection domain by virtualizing the MMU for monitoring all memory mapping updates within a massive system at the same privilege level. Further, nested MMU explicitly removes all instructions that may modify the MMU from non-TCB components.

TABLE I
PHYSICAL PAGE ATTRIBUTES

Attribute	Description
Owner	the entity to which the physical page belongs.
Type	the type of memory that the physical page is.
Max_use	the max number of times the physical page can be mapped.
Used	the number of times the physical page has been mapped.
Perm	access control restrictions on the physical page.

SecFortress virtualizes the MMU by configuring all kernel page-table-pages as read-only in the system initialization stage and enabling the CPU write-protection (WP) mechanism. Every time when the outerOS attempts to update kernel page-table-pages, a WP fault is generated to trap into the mediator, which performs a security check based on pre-defined policy (IV-A3). In addition to page table control, operations that can access the MMU must be monopolized by the mediator. These operations include modifications to the page table and the paging mechanism. SecFortress removes all instances of these operations from the outerOS, including aligned and misaligned instructions, such that the outerOS can neither modify the MMU states nor bypass the mediator. Besides, injecting new instructions into the outerOS could also threaten the mediator's security. Because the mediator monitors all page-table update operations, it can easily prevent instruction injection and data execution for code integrity. SecFortress maps most of the mediator code (except for sensitive instructions) as executable in both the mediator and non-TCB components for performance.

SecFortress significantly reduces the TCB of the virtualization layer by restricting the control of MMU to the mediator. On this basis, the mediator provides a series of security services (IV-A2) for cross-layer isolation.

2) Security services: The mediator provides core support for cross-layer isolation with three security services: memory protection, instruction protection, and enforced control-flow.

Memory protection is the first type of security service. Since each component has its own private data, the mediator needs to secure them. The mediator monitors the memory layout of all components through controlling the MMU, and provides secure memory allocation, memory access control, and memory update control for protecting private data.

Secure memory allocation. The mediator has an internal allocator for allocating the protected data. In addition, the mediator also allocates an inner memory pool to record the attributes of physical pages for every allocated or free memory. Table I details the attributes, which are used to control memory access and update according to related policies (IV-A3).

Memory access control. Figure 2 illustrates the memory access view of each component in SecFortress. For example, the left-most large box indicates the memory access view of the VM. i.e., each VM has full access to its own memory, but no access to others. The mediator has access to all memory and determines the memory view of each component by controlling the pagetables. Illegal memory accesses across components will cause page faults and be caught by the mediator.

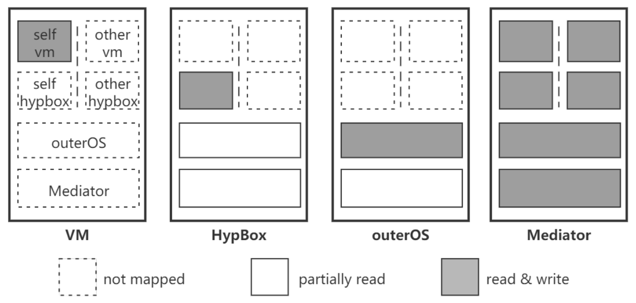

Fig. 2. Memory access view of VM, HypBox, outerOS and mediator.

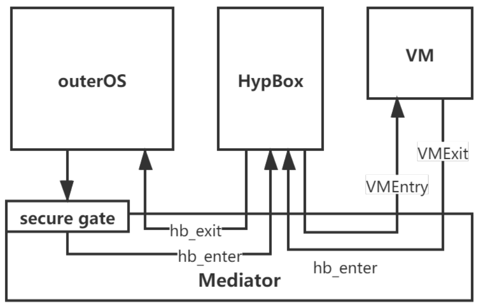

Fig. 3. The control flow of SecFortress.

Memory update control. This protection enforces memory mapping update and sensitive data update. As mentioned above, the mediator is the only component which can modify the page tables to control memory mapping updates. For example, when outerOS wants to update its page table, it needs to enter the mediator through a secure call gate explicitly. The mediator determines whether this update is valid according to the attribute information in the memory pool. Specifically, the mediator first examines the level of the page-table-page. If the current page is the last level page, the mediator then traverses the memory pool to determine whether the physical page has been assigned. The update request is considered valid if and only if this page has been assigned and its owner is the outerOS. If the current page is not the last level page, the mediator then checks whether the system enables the huge page mechanism. The update request is considered valid if and only if the huge page mechanism is enabled and the owner of the physical page is the outerOS. Besides, the mediator also checks specific bits in page table entry to ensure that this operation does not conflict with the system protection mechanism (i.e., write-protection). For sensitive data update, all sensitive data are divided into static data and dynamic data. Static data can never be modified once it is allocated, while the modification to dynamic data needs to be checked by the mediator through its page attributes (Table I).

Instruction protection is the second type of security service. In addition to memory protection, some special instructions also need to be taken seriously by the mediator. Such instructions can be classified into two categories: protection-related ones and control flow-related ones. The former ones are used to enable or disable protection mechanisms, and the latter may hijack the control flow or switch the contexts. SecFortress treats them as sensitive instructions and prohibits direct execution of them outside the mediator. Further, the mediator restricts sensitive instructions using instruction check, instruction unmap, and instruction hard-coded.

Instruction check means that the mediator does a sanity check with policy (IV-A3) after executing the protection-related instructions. Specifically, the mediator checks if specific bits in some registers (e.g., WP in CR0, SMEP in CR4) representing the security mechanisms are still set.

Instruction unmap is more restrictive than *instruction check*, used to enforce control flow-related instructions. It utilizes a dynamic instruction mapping approach to map instructions before execution and unmap them after execution. Specifically, the mediator allocates one internal physical page (4KB) for each instruction and ensures each of them has only one instance on the physical page. Before mapping an instruction page, the mediator checks the protection mechanisms and parameters to prevent a compromised component from directly accessing these instructions. The mediator then maps the page to the component. After executing instructions, the mediator does instruction checks and unmaps their pages. Since the mediator controls all memory mapping updates, any code injection attack to bypass the protection mechanism from a compromised component will be detected and stopped.

Instruction hard-coded forces the entry point of all calls to instructions in the non-TCB components to be hard-coded. The locations to invoke these instructions are built-in during the system bootup. A binary scanner is used to make sure that there are no such instructions in unexpected code regions.

Enforced control-flow is the last type of security service. When a component needs to update its memory mapping, modify sensitive data or execute sensitive instructions normally, it must switch to the mediator firstly. The mediator provides a **secure gate** for these switches. In this secure gate, the mediator first disables interrupts to prevent unexpected control flow transfers, and then disables the CPU write-protection mechanism. In addition, the mediator also switches to a secure stack for execution. After the mediator processes these operations, it enables the write-protection mechanism before returning to the ordinary contexts. This secure gate maintains the following rules to make context switches between components secure. First, all addresses that call this gate are explicitly hard-coded in non-TCB components and loaded during SecFortress bootup so that all non-TCB components can only enter the mediator through this gate. Second, interrupts are disabled to ensure that switches are atomic and cannot be redirected to other non-TCB components. Figure 3 illustrates the control flow of SecFortress.

3) *Policy enforcement:* The mediator stipulates some policies to enforce instruction and memory protection.

As mentioned above, sensitive instructions are divided into protection-related ones and control flow-related ones. Both types of instructions are monopolized by the mediator. Each instruction has only one instance in the mediator's code regions. On this basis, the mediator formulates strict policies for the execution of them. Details are shown in Table II. The mediator performs a sanity check with policies, ensuring that these instructions are not abused.

Memory protection is one of the core security services in the mediator. There are four levels policies for memory protection, corresponding to the increasing security requirements:

(P1) : Page can be read and written by its owner non-TCB component. It is mapped as read-only in other non-TCB components.

(P2) : Page can be read and written by its owner non-TCB component. It is unmapped in other non-TCB components.

(P3) : Page is mapped as read-only in its owner non-TCB component. It is unmapped in other non-TCB components.

(P4) : Page is unmapped in all non-TCB components.

It is worth noting that P4 is only used to protect the internal confidential data of the mediator. In addition, all code pages are write-protected, and all data pages are non-executable.

B. *Cross-layer Isolation*

1) Isolation between outerOS and HypBox: Preventing the outerOS from attacking HypBox is the first goal of SecFortress. Generally, the outerOS code runs at a high privilege level and controls all physical memory with the kernel master page table. A compromised outerOS can arbitrarily access HypBoxes' private memory. SecFortress isolates the outerOS from HypBoxes by creating isolated address spaces for HypBoxes and restricting the outerOS' access to HypBoxes.

Through having its own page table, each HypBox instance has a specific address space, which is separated from the outerOS. All page tables are maintained by the mediator and cannot be accessed by the outerOS. Memory areas are mapped differently in the outerOS and HypBox instances so that the former cannot access the private memory areas of the latter. Specifically, when allocating private data memory for a HypBox instance, the mediator maps these memory areas to this HypBox's address space, and unmaps them from the master page table. Memory mapping of a HypBox instance is initialized during its creation, and the mapping updates are monitored by the mediator through *secure memory allocation* and *memory update control*. Because the mediator controls all MMU operations, the outerOS cannot violate the confidentiality or integrity of HypBox instances. In addition, a HypBox instance can specify the security policy for its private data when requesting memory allocation. We classify the data segments of HypBox into six types, and each type is protected by a specific protection policy as shown in Table III.

Sensitive instructions (Table II) may directly or indirectly affect protection mechanism for HypBox instances. The mediator prohibits direct execution of these instructions in the outerOS. All instances of such instructions in the outerOS are removed by SecFortress. Executing such instructions must be monitored and tracked by the mediator. Each instruction has only one instance in the mediator's code region. For protection-related instructions, the mediator restricts them through *instruction check*. Specifically, the mediator checks if specific bits in some registers representing the security mechanisms are still set. Because the mediator code is also mapped as executable in the outerOS, these instructions are visible to the outerOS. Therefore, a compromised outerOS may disable the protection mechanisms by calling the mediator's interface or jumping to these instructions directly. Through *instruction check*, even if the outerOS clears the protection bit, the mediator can find and fix it immediately before a bad result occurs. Control flow-related instructions are restricted by the mediator using *instruction unmap*. The mediator unmaps these instructions from the outerOS and remaps them before execution. Each instruction instance is allocated on a separate 4K-page, and the mediator checks parameters before remapping it. For example, the mediator checks whether the parameter is a valid address before mapping MOV CR3.

As shown in Figure 3, when a VM exit occurs, the mediator catches this event and transfers VM exit to the associated HypBox instance. Usually, the HypBox instance directly handles this VM exit event and returns to the guest VM through the mediator. Sometimes, however, the HypBox instance needs to access physical resources or rely on functionalities in the outerOS, so switch to the outerOS' context is necessary in these scenarios. HypBox first calls the mediator's interface to trap into the mediator, which checks the VM exit reason and transfers this event to the outerOS. After the outerOS finishes processing, it calls the secure gate to return to the mediator. The mediator then verifies the results and switches back to the target HypBox. Since the mediator controls memory access and instruction execution, the outerOS can neither access a HypBox's memory without authorization nor execute sensitive instructions to bypass the protection mechanism.

2) Isolation among HypBox instances: Using untrusted VM states as input is another major security threat for the hypervisor. The design of traditional hypervisors allows a malicious VM to harm other VMs through attacking the hypervisor. To defend against such an attack, SecFortress uses the mediator to provide a robust isolation mechanism between different HypBox instances.

One HypBox has full access to its own memory, but no access to others. In addition, some memory areas of the mediator and the outerOS are mapped as read-only into the HypBox instance, so the HypBox instance is only allowed to access those areas, but cannot modify them. The memory access view of a HypBox instance is determined during its creation and should remain monitored by the mediator using *memory access control* and *memory update control* during its life cycle. The mediator maps all page-table-pages of the HypBox as read-only and enables the write protection to forbid the HypBox instance from updating its memory mappings. Instead, the memory mapping updates of the HypBox instance need to explicitly call the mediator's interface. SecFortress further enforces $W \oplus X$ in the HypBox's address space to ensure that no page in the HypBox is both writable and executable. The mediator controls HypBox's access to different data segment

TABLE II
SENSITIVE INSTRUCTIONS AND POLICY ENFORCEMENT

Type	Instruction	Consequence	Policy
Protection-related	MOV CR0	Threaten the paging and the write-protection mechanisms.	Prohibit clearing the PG and WP bits of CR0.
	MOV CR4	Threaten the supervisor-mode execution/access prevention.	Prohibit clearing the SMEP/SMAP bits of CR4.
	WRMSR	Threaten the No-execution protection.	Prohibit clearing the NXE bit of EFER.
Control flow-related	MOV CR3	Tamper with memory address space.	CR3 value must be a declared valid page table.
	MOV CR8	Threaten the external interrupts priority.	Prohibit modifying the TPL-related bits of CR8.
	VMRUN	Change the control flow to run the virtual machines.	Prohibit modifying specific VMCS regions.
	VMRESUME	Change the control flow to resume the virtual machines.	Prohibit modifying specific VMCS regions.
	VMLAUNCH	Change the control flow to resume the virtual machines.	Prohibit modifying specific VMCS regions.
	VMOFF	Turn off the VMX mode of the CPU to destroy the system.	The execution of this instruction is prohibited.

TABLE III
HYPBOX DATA TYPES AND PROTECTION POLICIES.

Date Type	Flag	Description	Policy
HypBox Normal Data	HB_RW	These data pages belong to a HypBox instance. They are readable and writable to their owner HypBox instance, but invisible to other ones and the outerOS. A HypBox instance can modify these pages without notifying the mediator.	P2
HypBox Critical Data	HB_RW_CHECK	These data pages contain secret information to manage a HypBox instance. They are readable to their owner HypBox instance, but not accessible by other ones and the outerOS. Writing to pages from one HypBox needs to be checked by the mediator.	P3
Pages storing HypBox Page Table	HB_RO	These pages contain the page table information of a HypBox instance. They are mapped as read-only in HypBox but invisible to the other ones and the outerOS. Only the mediator can write these pages.	P3
Pages storing EPT Page Table	HB_RO	These pages contain the page table content of the VM served by the current HypBox. They are mapped as read-only in its owner HypBox but invisible to the other ones and the outerOS. Only the mediator can write these pages.	P3
Guest-Related Data	HB_RW_CHECK	These data pages contain VM-related data and belong to the per-VM HypBox. Writing to these pages from the HypBox needs to be checked by the mediator based on data updates policy.	P3
Mediator-granted Data	HB_RO_GRANT	These data pages belong to the mediator. Permissions are dynamically granted to the HypBox.	P3

memory according to the policies detailed in Table III.

The execution of sensitive instructions may hijack the control flow or break the protection mechanism. In order to prevent a compromised HypBox from attacking others, the mediator restricts the HypBox from directly accessing these instructions. Access to these instructions needs to be forwarded to the mediator. The mediator uses *instruction hard-coded* to force no these instructions to appear in unexpected code regions. In addition, the mediator maps the guest memory as non-executable in the HypBox, ensuring that an attacker cannot generate new code using the guest memory. User-level code is prohibited from being executed in the kernel-level.

V. IMPLEMENTATION

We implemented a prototype of SecFortress on KVM. Our system runs on Ubuntu 18.04.5 with Linux 5.2 for Intel VT. The virtualization software at the user level is QEMU 4.0.0.

A. HypBox management

SecFortress abstracts a HypBox to encapsulate the hypervisor running environment for each VM. The mediator provides three types of interfaces that are used to securely manage HypBoxes during their life cycles. All interfaces are pre-built in the mediator and loaded at the system bootup stage. A HypBox must explicitly call interfaces during its life cycle.

The first type is used to create or destroy HypBox instances. Such interfaces include *hb_create ()* and *hb_destroy ()*. *hb_create ()* uses the VM descriptor as a parameter, and returns an integer as the HypBox handle. This handle can be used by the mediator to uniquely locate a specific HypBox instance. *hb_create ()* is called during guest VM booting stages, while *hb_destroy ()* is called when the guest VM shuts down.

The second type is context transition, used to enter or exit the HypBox. Such interfaces include *hb_enter ()* and *hb_exit ()*. When a VM exit event occurs on a guest VM, the CPU first traps into the mediator, which invokes *hb_enter ()* to enter the target HypBox instance. Moreover, when a HypBox needs to call functions of the outerOS or the mediator, it calls *hb_exit ()* to switch to the mediator. According to the exit reason, the mediator processes this request, or forwards it to the outerOS.

The third type is used to allocate or release HypBox's memory regions. Such interfaces include *hb_malloc_internal_memory ()* and *hb_free ()*. The mediator invokes *hb_malloc_internal_memory ()* to allocate page-aligned memory regions for HypBox's private data. When allocating memory regions, a flag parameter is used to specify the data type and protection policy. *hb_free ()* zeroes the HypBox's memory regions before freeing them.

B. Components isolation

SecFortress limits the outerOS access to HypBox through memory access control and paging mechanisms. In the outerOS, the master page table responsible for managing physical memory is called swapper_pg_dir. To protect HypBox's memory, SecFortress maps swapper_pg_dir pages as read-only in the outerOS. SecFortress also enforces write-protection through setting PTE.RW=0 and CR0.WP=1. If the outerOS wants to update the swapper_pg_dir, it must explicitly trap to the mediator. Any unexpected update of swapper_pg_dir will cause a WP fault, which is caught by the mediator. The mediator has an allocator for HypBox's private data. When a block of physical memory is allocated to a HypBox as private data memory, the mediator removes its mappings from swapper_pg_dir. If the outerOS or other host application tries to access HypBox's private data, the system will generate a PF exception. The mediator immediately detects it and defends against this attack. It is worth noting that if there are no relevant memory mappings in the swapper_pg_dir, the outerOS will generate an OOP which causes a system hang. To keep our system's practicability, we added a hook function instead of OOP to handle these exceptions, which generates warning based on the current context and page fault address.

There are two scenarios where HypBox needs to switch to the outerOS. The first is to handle EPT violation. EPT mechanism is based on the original CR3 page table address mapping, and furthermore, it introduces an EPT page table to implement another mapping. The original page table is responsible for guest virtual address (GVA) to guest physical address (GPA) mapping, while the EPT page table handles the mapping from GPA to host physical address (HPA). For each VM, in addition to the EPT page table, there is also a host page table, which is used by Qemu process where the guest VM is located. When a GPA-to-HPA is not existed in EPT, the VM generates an EPT violation, which switches CPU privilege to run KVM. KVM first translates GPA to host virtual address (HVA) through memory slots, and then searches HVA-to-HPA mapping in the host page table. If the HVA-to-HPA is not existed, the outerOS handles it as a host page fault with swapper_pg_dir. In our experiments, when handling EPT violation, the HypBox first traps into the mediator to switch to the outerOS. If the HVA-to-HPA mapping exists in the host page table, the mediator notifies HypBox to update the EPT page table using this HPA. If the mapping does not exist, a PF exception occurs. The mediator captures this exception and handles it using swapper_pg_dir.

The second is to handle device I/O. KVM uses Qemu to support I/O virtualization. Typically, a guest VM generates an I/O VM exit and enters the KVM. KVM then determines operation type, and passes it to Qemu. For emulated devices, SecFortress uses the trap-and-emulate approach to handle both port I/O and MMIO. SecFortress configures the I/O field in VMCS to trap I/O instructions and associated data to the mediator. In addition, The mediator controls the guest EPT and clears the mappings of virtual device I/O memory. The mediator captures I/O VM exit events, and passes only the I/O data to Qemu. After the emulation is completed, the mediator checks the results, and returns to the HypBox. For virtio devices, SecFortress uses an end-to-end I/O encryption [12] for virtio operation. For secure DMA, the mediator controls the IOMMU to track PCI devices' physical pages using paging attributes. IOMMU page tables are protected by the mediator through write-protection. Any modification to IOMMU page-table-pages will cause a WP fault. In this way, SecFortress ensures that the host handles I/O operation correctly.

C. Intervention between HypBox and VM

SecFortress enables bidirectional monitoring by interfering with the control flows between VM and HypBox. When a guest is running and a VM exit occurs, the control flow is first transferred to the mediator. The mediator checks the VM exit reason, and then sends this event to the specific HypBox. Before sending to the HypBox, the mediator checks the guest state regions in VMCS according to the VM exit reason, ensuring that no invalid parameters are passed to the HypBox. If the HypBox needs to access VM's data, it will call the mediator to check the permissions. The memory areas, that are related to VMCS in HypBox, are marked as HB_RW_CHECK during allocation. The mediator checks VMCS to ensure that they have not been tampered with before returning to the guest.

The VM image is stored on the disk in an encrypted manner. Before each startup, the mediator performs integrity checks on the VM images and decrypts the images. The key can be managed by the cloud providers' key management service (KMS) [25]. In addition, SecFortress does not support online updates. Software updates for the SecFortress virtualization layer are currently only available with offline install-packages.

VI. Evaluation

We first evaluate the security of SecFortress by analyzing real CVEs. Then we evaluate the performance overhead of SecFortress by running various benchmarks in the guest OS, e.g., SPEC CPU2017 [26] and kernel compilation to evaluate the runtime overhead, as well as IOzone [27], DD [28] and Netperf4 [29] to evaluate disk and network I/O overhead. In addition, we also measure some micro benchmarks to further understand the factors that affect performance. Our experiments are conducted on a machine with Intel Core i7-7700K processor and 16 GB physical memory. The host OS is Ubuntu 18.04.5 with Linux kernel 5.2.0. We assign four virtual CPU cores and 4GB memory to one guest VM, which runs Ubuntu 18.04.5 with Linux kernel 5.4.0.

A. Evaluation of Practical Attacks

We evaluate SecFortress' effectiveness against a compromised outerOS or malicious VMs by analyzing CVEs and identifying the cases where SecFortress secures hypervisor runtime despite any compromise. We analyzed 20 CVEs related to Linux/KVM, and the results are shown in Table IV and V. The CVEs consider two cases: a malicious VM who exploits KVM functions, and a compromised outerOS who exploits bugs in Linux/KVM. The CVEs related to our thread model could result in privilege escalation, info leakage, memory corruption, and denial-of-service in Linux/KVM.

Privilege escalation is an intermediate state, leading to arbitrary attack consequences. Attackers may exploit vulnerabilities in the hypervisor or host applications to gain host OS privilege. For these attacks, our focus is to prevent further damage to other VMs after privilege escalation. In SecFortress, the outerOS and HypBoxes are out of the TCB. Once they tries to access memory beyond its permissions, it will be captured by the mediator immediately. So even if they are compromised, they cannot cause devastating results.

Info leakage is aimed at reading the memory of sensitive data from any part of system. These attacks may be due to improper parameter handling or uninitialized memory. In SecFortress, each non-TCB component is limited to its address space and isolated from others. A compromised non-TCB component cannot read any information except its own.

Memory corruption attacks usually begin with out-of-bound access or mishandling memory mapping. In SecFortress, similar to leakage information, memory allocation and sensitive

TABLE IV
CASE STUDY: FROM THE HOST

CVE	Description	Defense	KVM	SecFortress
CVE-2019-13272	Privilege escalation: Mishandles the recording of the credentials of a process that wants to create a ptrace relationship.	An attacker obtains root permission by leveraging certain scenarios with a parent-child process relationship. Then, we emulated the deep attack to access HyperBox's page table. As expected, it triggered page fault.	No	Yes
CVE-2016-5195	Privilege escalation: Improperly handing of a copy-on-write (COW) feature to write to a read-only memory mapping.	First, an attacker gains privileges by leveraging incorrect handling of a COW feature. Then, we emulated the deep attack to access HyperBox's page table. As expected, it triggered page fault.	No	Yes
CVE-2018-18021	Privilege escalation: Mishandling of VM register state allows local users to redirect kernel control flow.	An attacker can arbitrarily redirect the control flow by leveraging incorrect KVM_SET_ON_REG ioctl handler. Then, we emulated the deep attack to modify HyperBox's page table. As expected, it triggered page fault.	No	Yes
CVE-2013-2234	Info leakage: Improperly initializing of structure members, which allows local users to obtain secret from kernel heap memory.	An attacker obtains secrets from kernel heap by reading a broadcast message from the notify interface of an IPSec key_socket. Since HyperBox has its own secure heap memory, attacker can't achieve the final goal.	No	Yes
CVE-2016-9756	Info leakage: Improperly initializing of code segment, which allows local users to obtain sensitive information.	An attacker obtains sensitive information from kernel stack memory via a crafted application. Since HyperBox has its own secure stack memory, attacker can't achieve the final goal.	No	Yes
CVE-2010-4525	Info leakage: Improperly initialize of interrupt.pad structure member, which allows local users to obtain sensitive information from kernel stack memory.	An attacker obtains potentially sensitive information from kernel stack memory via unspecified vectors. Since HyperBox has its own secure stack memory, attacker can't achieve the final goal.	No	Yes
CVE-2013-4592	Denial-of-Service: Mishandling memory slots in virt/kvm/kvm_main.c allows local users to cause memory consumption.	An attacker causes a DoS (memory consumption) by leveraging certain device access to trigger movement of memory slots. Since the mediator can manage the allocation of certain memory slot, some cases can be avoided.	No	Partially solved
CVE-2013-4129	Denial-of-Service: Improperly check of timer, which allows local users to cause a denial of service.	An attacker causes a denial of service (BUG and system crash) via vectors involving the shutdown of a KVM virtual machine. Since the bug directly shutdown the VM, it is out of our scope.	No	No

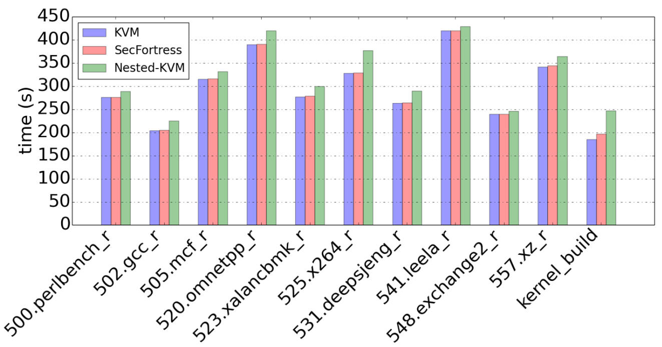

Fig. 4. CPU and memory benchmarks performance

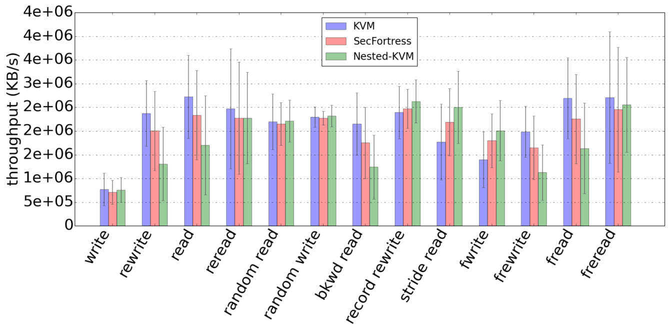

Fig. 5. Disk I/O benchmarks performance

data update are controlled by the mediator. So once a HypBox tries to perform an invalid memory access, the mediator will immediately find it and kill the guest VM.

Denial-of-Service is aimed at exploiting vulnerabilities in HypBox to crash other VMs or the entire hypervisor. For attacks that try to crash other VMs, SecFortress can check the memory and registers of guest VM states before returning to the guest VM. Any mishandling state will be fixed immediately. For attacks that try to crash the hypervisor, SecFortress ensures that these attacks can only affect its own instance by isolating each HypBox. SecFortress focuses on DoS attacks from guest and cannot defend against host DoS.

B. *Macro Benchmarks*

Runtime benchmarks. We measure the performance of all applications in the SPEC CPU2017 benchmark in a guest VM running on the original KVM (as the baseline performance), SecFortress, and nested KVM. We ran nine rounds of each benchmark with real-world workload and the average runtime of each benchmark is shown in Figure 4. Specifically, for CPU-intensive benchmarks, such as perlbench_r, gcc_r, and x264_r, there is nearly no overhead since SecFortress hardly intercepts CPU execution or introduces additional VM exits. For memory-intensive benchmarks, e.g., mcf_r, and deepsjeng_r, a slight performance overhead is observed (from 0.3% to 0.7%), since the monitoring and checking of memory operations and the control flow switching between HypBox instances and the outerOS introduce runtime overhead to SecFortress. However, since SecFortress controls most of the switch checks through the mediator, even in the worst cases, i.e., the kernel compilation, the runtime overhead is only 6.47%, which is much smaller than nested KVM (33.39%). Overall, the average runtime overhead for all the benchmarks is 0.8%.

I/O benchmarks. To evaluate the disk I/O overhead, we select IOzone benchmarks, with the settings of 4KB block size and 32MB file size. We first try 100 rounds of the benchmarks, but find the standard deviations of the throughput for each benchmark is extremely large. Then we run 1000 rounds of each benchmark, compute the average throughput and show the results in Figure 5. Overall, the average I/O performance overhead of SecFortress is 5% (nested KVM is 10%). For some test cases (e.g., stride-read, record-rewrite, and fwrite) however, the performance of SecFortress is better than that of the native KVM. We believe the reason is caching and buffering, since the frequent memory operations of SecFortress

TABLE V
CASE STUDY: FROM THE VM

CVE	Description	Defense	KVM	SecFortress
CVE-2015-7504	Privilege escalation: Heap-based buffer overflow in hw/net/pcnet.c allows guest OS administrators to execute arbitrary code.	First, an attacker executes arbitrary code via a series of packets in loopback mode. Then we emulated the deep attack to access other HyperBox's page table. As expected, it triggered page fault.	No	Yes
CVE-2013-0311	Privilege escalation: Improperly handling cross-region descriptors in drivers/vhost/vhost.c allows guest OS to obtain host OS privilege.	First, an attacker obtain host OS privileges by leveraging KVM guest OS privileges. Then, we emulated the deep attack to modify HyperBox's page table. As expected, it triggered page fault.	No	Yes
CVE-2016-9777	Privilege escalation: Improperly restricting the VCPU index allows guest OS users to gain host OS privileges.	First, an attacker gains host OS privileges via a crafted interrupt request. Then, we emulated the deep attack to modify HyperBox's page table. As expected, it triggered page fault.	No	Yes
CVE-2016-3713	Info leakage: Improperly handling MSR 0x2f8 in arch/x86/kvm/mtrr.c allows guest OS users to obtain sensitive information.	An attacker obtains sensitive information via a crafted ioctl call. We emulated the attack to read or write to the other HypBox's kvm_arch_vcpu data structure. As expected, it triggered page fault.	No	Yes
CVE-2013-1798	Info leakage: Improperly handling operations in virt/kvm/ioapic.c allows guest OS users to obtain sensitive information from host.	An attacker obtains sensitive information from host OS memory via a crafted application. Since secure data is isolated from host OS memory, attacker can't achieve the final goal.	No	Yes
CVE-2015-5165	Info leakage: Improperly RTL8139 emulation allows guest OS to read up to 65 KB of QEMU heap memory.	An attacker reads heap memory via unspecified vectors. Since HyperBox has its own secure heap memory, attacker can't achieve the final goal.	No	Yes
CVE-2017-2596	Memory corruption: Improperly emulating the VMXON instruction in arch/x86/kvm/vmx.c allows guest OS users to cause host OS memory corruption.	An attacker can cause host OS memory consumption by leveraging the mishandling of page references in the traditional design. But in our system, the VMXON will be trapped into the mediator, this attack can be prevented.	No	Yes
CVE-2015-4036	Memory corruption: Array index error in the tcm_vhost_make_tpg function allows guest OS users to cause memory corruption.	An attacker causes memory corruption via a crafted VHOST_SCSI_SET_ENDPOINT ioctl call. In our system, the attacker only influence its own HyperBox instance.	No	Yes
CVE-2014-3601	Memory corruption: Mishanding page mapping in virt/kvm/iommu.c allows guest OS users to cause host OS memory corruption.	An attacker can cause memory corruption by triggering a improper gfn value. In our system, the scope of the gfn is controlled by the mediator, therefore the memory corruption can be prevented.	No	Yes
CVE-2017-1000407	Denial-of-Service: Flooding the I/O port to crash the hypervisor.	An attacker can cause a kernel panic by flooding the diagnostic port 0x80. In our system, the attacker only influence its own HyperBox, because the HyperBoxes provide services to their own guest independently.	No	Yes
CVE-2017-1000252	Denial-of-Service: Out-of-bounds value causes hypervisor crash.	An attacker can crash the hypervisor via an out-of bounds guest_irq value. In our system, the attacker only influence its own HyperBox, because the HyperBoxes provide services to their own guest independently.	No	Yes
CVE-2014-7842	Denial-of-Service: KVM bug allows guest users to crash its own virtual machine.	An attacker can crash its guest VM via a crafted application. We regard this attack as inner attack and it's out of our scope.	No	No

TABLE VI
DISK I/O PERFORMANCE OVERHEAD USING DD

Operation	KVM	SecFortress	Nested KVM
128MB-READ (MB/s)	182.50	180.95 (0.8%)	108.05 (40.8%)
128MB-WRITE (MB/s)	83.35	77.95 (6.5%)	75.2 (9.8%)

TABLE VII
NETWORK I/O PERFORMANCE OVERHEAD

Operation	KVM	SecFortress	Nested KVM
TCP_STREAM (MB/s)	514.70	465.20 (9.62%)	174.35 (66.13%)
TCP_RR (per second)	7369.99	7268.27 (1.38%)	3153.19 (57.22%)

cause some I/O data to be accessed from memory instead of disk. Some test results of nested KVM are also for the same reason. We further use DD to evaluate I/O performance, with the configuration of 128MB block size, 1GB file size and direct mode. The read and write overheads are about 0.8% and 6.5% respectively, as shown in Table VI. For network I/O performance overhead, we use Netperf4 benchmark with TCP_STREAM and TCP_RR, configured at 100MB and 20 rounds. The average overhead is 5.5% as shown in Table VII.

C. Micro Benchmarks

To better understand the factors that cause the performance overhead, we conduct some micro benchmarks. These tests are run for 100,000 times to get the average values. We first test the number of CPU cycles added by switching to HypBox. The results show that the overhead of switching to HypBox is 2626 cycles. Then we test the number of CPU cycles added by switching to the mediator. The overhead is 235 cycles.

SecFortress introduces 2249 lines of C and 87 lines of assembly code as TCB (measured by cloc [30]). We remove the outerOS and HypBoxes out of the TCB, so as long as the mediator's code is verified to be secure, there will be no vulnerability for control-flow hijacks or ROP attacks. According to [31], current verification can be applied to more than 10K LOC, so it is reasonable to verify SecFortress' correctness, which is left as our future work.

VII. RELATED WORK

HyperLock and DeHype deconstruct KVM by assigning a separate isolated hypervisor instance to each VM. Both of them and SecFortress create an isolated context for each VM to isolate hypervisor exploitation. HyperLock and DeHype focus on removing KVM from Linux kernel TCB. Both of them do not protect hypervisor against compromised host OS. In contrast, SecFortress deprivileges the host OS and uses the mediator to protect hypervisor even if the host OS is compromised. MultiHype [32] supports running multiple hypervisors on a single physical platform, and each hypervisor can further create multiple VMs. SecFortress creates a single HypBox for each VM to ensure a smaller TCB and stronger isolation. Nexen and SecFortress both use Nested Kernel to reconstruct the virtualization platform, but Nexen does not consider the security vulnerabilities in its shared service domain. Cloudvisor [9] and Cloudvisor-D [10] protect VMs against a compromised hypervisor through nested virtualization design. In contrast to SecFortress, these systems focus on isolating the hypervisor's impact on VMs and do not harden the hypervisor itself. Using a microkernel design to reconstruct a hypervisor can effectively reduce the hypervisor's TCB. NOVA divides the hypervisor into several components and moves most of them to userspace. HypSec [12] and SecKVM [33] split a monolithic hypervisor into a small TCB corevisor and an untrusted hostvisor using ARM VE. SKEE [34] also deprives the privileged OS of controlling MMU. However, SecFortress further leverages cross-VM control protection.

In recent years, there are many works on hardware-based defense technologies. vTZ [11] protects the guest-TEE against an untrusted hypervisor through virtualizing ARM TrustZone. However, it does not support normal world VMs protection. HA-VMSI [35] prevent guest VMs from untrusted hypervisor but only supports limited virtualization features. HyperCoffer [13] uses AISE+BMT encrypted secure processor to protect VM's data. AMD SEV [36] protects guests from hypervisor by encrypting guest memory. Unlike SecFortress's target, these systems cannot protect hypervisor against attacks from malicious VMs. Intel TDX [37] is a new hardware feature designed to isolate VMs from the hypervisor and any non-TD software on the platforms by adding a secure-arbitration mode. Compared with SecFortress, Intel TDX aims to prevent virtualization platform from attacking VMs, while SecFortress focuses on the bidirectional isolation protection between VMs and hypervisor. AWS Nitro Enclaves [38], which is based on lightweight OS, creates isolated VM execution environments enclaves on the nitro hypervisor. Arm CCA [39] builds on TrustZone and introduces Realm Management Extension to protect all data and code. It focuses on protecting applications rather than the entire VM.

VIII. Conclusions

In this paper, we present SecFortress, a novel cross-layer isolation approach, to secure hypervisor runtime. SecFortress partitions the virtualization platform into a trusted mediator, an isolated outerOS, and multiple restricted HypBox instances. SecFortress deprivileges the outerOS from accessing the hypervisor's memory. Each HypBox instance is confined with the least privilege to access memory, so that it cannot violate the integrity and confidentiality of other instances even if it is compromised. We implemented a prototype of SecFortress and evaluated its security and practicability. The experimental results show that SecFortress can defeat the exploits against the host OS and VMs with negligible performance overhead.

References

[1] A. Kivity, Y. Kamay, D. Laor, U. L, and A. L, "Kvm: the linux virtual machine monitor," in *Ottawa Linux Symposium*, 2007.

[2] A. M. Azab, P. Ning, Z. Wang, X. J, X. Z, and N. C. S, "Hypersentry: enabling stealthy in-context measurement of hypervisor integrity," in *ACM CCS*, 2010.

[3] H. Moon, H. Lee, I. Heo, K. Kim, Y. Paek, and B. B. Kang, "Detecting and preventing kernel rootkit attacks with bus snooping," *TDSC*, 2017.

[4] U. Steinberg and B. Kauer, "Nova: a microhypervisor-based secure virtualization architecture," in *EuroSys*, 2010.

[5] E. Keller, J. Szefer, J. Rexford, and R. B. Lee, "Nohype: Virtualized cloud infrastructure without the virtualization," in *ISCA*, 2010.

[6] D. Williams, Y. Hu, U. Deshpande, P. K. Sinha, and H. Jamjoom, "Enabling efficient hypervisor-as-a-service clouds with eemeral virtualization," *ACM SIGPLAN Notices*, 2016.

[7] L. Shi, Y. Wu, Y. Xia, N. Da, H. C, B. Zang, and J. Li, "Deconstructing xen," in *NDSS*, 2017.

[8] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield, "Breaking up is hard to do: security and functionality in a commodity hypervisor," in *SOSP*, 2011.

[9] F. Zhang, J. Chen, H. Chen, and B. Zang, "Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *SOSP*, 2011.

[10] Z. Mi, D. Li, H. Chen, B. Zang, and H. Guan, "(mostly) exitless vm protection from untrusted vmm through disaggregated nested virtualization," in *USENIX Security*, 2020.

[11] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vtz: Virtualizing ARM trustzone," in *USENIX Security*, 2017.

[12] S. Li, J. S. Koh, and J. Nieh, "Protecting cloud virtual machines from commodity hypervisor and host operating system exploits," in *USENIX Security*, 2019.

[13] Y. Xia, Y. Liu, and H. Chen, "Architecture support for guest-transparent vm protection from untrusted hypervisor and physical attacks," in *HPCA*, 2013.

[14] Y. Wu, L. Yutao, L. Ruifeng, H. Chen, and Z. Binyu, "Comprehensive vm protection against untrusted hypervisor through retrofitted amd memory encryption," in *HPCA*, 2018.

[15] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve, "Nested kernel: An operating system architecture for intra-kernel privilege separation," *Computer architecture news*, 2015.

[16] F. Bellard, "Quick emulator," https://www.qemu.org, 2001.

[17] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Fa, N. Har, A. Gor, A. Lig, O. W, and B. A. Y, "The turtles project: Design and implementation of nested virtualization," in *OSDI*, 2010.

[18] "Common vulnerabilities and exposures," http://cve.mitre.org, 2021.

[19] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "Hypervision across worlds: Real-time kernel protection from the ARM trustzone secure world," in *ACM CCS*, 2014.

[20] Z. Wang, C. Wu, M. C. Grace, and X. Jiang, "Isolating commodity hosted hypervisors with hyperlock," in *EuroSys*, 2012.

[21] C. Wu, Z. Wang, and X. Jiang, "Taming hosted hypervisors with (mostly) deprivileged execution," in *NDSS*, 2013.

[22] "Hypervisor industry research report 2021 by type, applications, regions, market size estimation and forecast to 2025," https://www.marketwatch.com/, 2021.

[23] 2021. [Online]. Available: https://sourceforge.net/projects/tboot/

[24] "Kernel page-table isolation," https://en.wikipedia.org/wiki/Kernel_page-table_isolation, 2018.

[25] "AWS key management service," https://aws.amazon.com/kms, 2020.

[26] SPEC, "Standard performance evaluation corporation," https://www.spec.org/cpu2017/, 2017.

[27] D. Capps and W. Norcott, "Iozone filesystem benchmark," http://iozone.org/, 2019.

[28] "Device driver," https://github.com/torvalds/linux, 2021.

[29] H. N. P. Team, "Netperf," https://hewlettpackard.github.io/netperf/, 2005.

[30] A. Danial, "cloc," https://github.com/AlDanial/cloc, 2020.

[31] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, and D. Costanzo, "Certikos: An extensible architecture for building certified concurrent OS kernels," in *OSDI*, 2016.

[32] W. Shi, J. Lee, T. Suh, D. H. Woo, and X. Zhang, "Architectural support of multiple hypervisors over single platform for enhancing cloud computing security," in *CF*, 2012.

[33] S.-W. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui, "Formally verified memory protection for a commodity multiprocessor hypervisor," in *USENIX Security*, 2021.

[34] A. M. Azab, K. Swidowski, R. Bhutkar, J. Ma, W. S, R. W, and P. N, "SKEE: A lightweight secure kernel-level execution environment for ARM," in *NDSS*, 2016.

[35] M. Zhu, B. Tu, W. Wei, and D. Meng, "HA-VMSI: A lightweight virtual machine isolation approach with commodity hardware for ARM," in *VEE*, 2017.

[36] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption," https://developer.amd.com/, 2016.

[37] Intel, "Intel trust domain extensions," https://software.intel.com, 2020.

[38] Amazon, "AWS nitro enclaves," https://docs.aws.amazon.com/enclaves/, 2020.

[39] ARM, "ARM confidential compute architecture," https://www.arm.com, 2021.