(Mostly) Exitless VM Protection from Untrusted Hypervisor through Disaggregated Nested Virtualization

Zeyu Mi, Dingji Li, Haibo Chen, Binyu Zang, Haibing Guan
Shanghai Key Laboratory for Scalable Computing Systems,
School of Software, Shanghai Jiao Tong University

Abstract

Today's cloud tenants are facing severe security threats such as compromised hypervisors, which forces a strong adversary model where the hypervisor should be excluded out of the TCB. Previous approaches to shielding guest VMs either suffer from insufficient protection or result in suboptimal performance due to frequent VM exits (especially for I/O operations). This paper presents CloudVisor-D, an efficient nested hypervisor design that embraces both strong protection and high performance. The core idea of CloudVisor-D is to disaggregate the nested hypervisor by separating major protection logics into a protected *Guardian-VM* alongside each guest VM. The *Guardian-VM* is securely isolated and protected by the nested hypervisor and provides secure services for most privileged operations like hypercalls, EPT violations and I/O operations from guest VMs. By leveraging recent hardware features, most privileged operations from a guest VM require no VM exits to the nested hypervisor, which are the major sources of performance slowdown in prior designs. We have implemented CloudVisor-D on a commercially available machine with these recent hardware features. Experimental evaluation shows that CloudVisor-D incurs negligible performance overhead even for I/O intensive benchmarks and in some cases outperforms a vanilla hypervisor due to the reduced number of VM exits.

1 Introduction

One premise of multi-tenant clouds is that the cloud will guarantee the privacy and integrity of tenants' virtual machines (VMs). However, this premise is severely threatened by exploits against the usually-vulnerable hypervisor (including the management VM or the host OS). In fact, with the code size and complexity of the hypervisor continually increasing, the number of discovered security vulnerabilities of the hypervisor increases as well. As shown in Table 1, the total number of uncovered security vulnerabilities in the Xen hypervisor [18] has increased from 32 in 2012 to 303 in 2019.

Year	Xen	KVM	VMWare
2012	32	16	18
2013	50	19	16
2014	32	20	14
2015	54	15	9
2016	35	12	24
2017	47	13	21
2018	29	9	31
2019	24	7	21

Table 1: The numbers of vulnerabilities discovered in Xen [8], KVM [5] and VMWare [7] from 2012 to 2019.

There have been several software approaches to shielding a VM from an untrusted hypervisor, which can be mainly classified into the "in-the-box" or "out-of-the-box" approaches. The "in-the-box" approach attempts to harden the hypervisor layer using various techniques such as the hypervisor decomposition [22, 54, 58], the control flow integrity [63] and minimizing the hypervisor layer [33]. However, while such an approach can thwart attackers exploiting the hypervisor vulnerabilities to a certain extent, they cannot eliminate the risks of exploiting hypervisor vulnerabilities.

The "out-of-the-box" approach exploits a nested hypervisor to deprivilege the commodity hypervisor and securely interposes all interactions between guest VMs and the hypervisor to protect privacy and integrity. Specifically, CloudVisor [72] introduces a small nested hypervisor underneath the Xen hypervisor and securely isolates the Xen hypervisor and its VMs. It uses cryptographic approaches to guaranteeing the privacy and integrity of guest data. However, this design is at the cost of notably increased VM exits to the nested hypervisor. For instance, these numerous VM exits bring up to 54.5% performance overhead for I/O intensive workloads.

Recently, there have been increasing interests to leverage the secure hardware modules like Intel SGX [13, 47] to guarantee the security and privacy of applications executing in an untrusted hypervisor [19, 28, 53, 61]. Such an approach can provide reliable protection agasint an stronger threat model which contains the adversary controlling hardware. However, two facts limit its usage for VM protection in a virtualized environment. First, the SGX enclaves are only available to run in user mode, preventing its use to provide a VM containing both user and kernel mode. Second, the hardware limitations (e.g., limited EPC memory at 128/256 MB) usually incur significant performance overhead for memory intensive workloads (sometimes 3X [15, 50, 61]).

In this paper, we present CloudVisor-D, a design that securely and efficiently shields VMs from a compromised hypervisor. Like prior solutions such as CloudVisor, CloudVisor-D leverages nested virtualization to protect the privacy and integrity of guest VMs. However, CloudVisor-D tackles the deficiency of nested virtualization through a disaggregated design by decomposing the nested hypervisor functionality into a tiny nested hypervisor (RootVisor) in the privileged mode and a set of Guardian-VMs in the non-privileged mode. Such a disaggregated design provides one Guardian-VM for each guest VM and offloads most protection logics to each Guardian-VM, while the tiny RootVisor is responsible for isolating all the Guardian-VMs from the commercial hypervisor (SubVisor) and guest VMs. Note that a Guardian-VM is not a full-fledged VM but only contains a few service handlers and is invisible to the SubVisor. Thus, it consumes a very small amount of resources.

Recent hardware advances (e.g., VMFUNC and virtualization exception) enable the self-handling of VM exits and efficient EPT switching in the guest mode. Based on these new hardware features, a Guardian-VM can handle offloaded VM operations without VM exits. Assisted by the Guardian-VM, the guest VM is able to directly invoke the hypercall handling functions in the SubVisor without trapping into the RootVisor. By utilizing the virtualization exception, normal EPT violations are converted to exceptions in the guest mode, which are then redirected to the SubVisor by the Guardian-VM for processing.

However, it is non-trivial to handle VM operations securely in the guest mode. A VM or the SubVisor may maliciously switch EPT to bypass or even attack the Guardian-VM. Even if there are some existing solutions [27,39,44,49] that try to defend against this type of attack, none of them defeats the new variant of attack we encounter since these solutions assume that the attacker is not able to modify the CR3 register value, which is not the case in CloudVisor-D. CloudVisor-D provides a series of techniques to defend against this attack. First, the RootVisor creates an isolated environment to make Guardian-VMs tamperproof. Second, each Guardian-VM enforces that it interposes all communication paths in the guest mode between a guest VM and the SubVisor. The complete mediation is achieved by using the *dynamical EPTP list manipulation* technique and the *isolated Guardian-VM page table* technique.

Based on the tamperproof and complete mediation properties, a Guardian-VM can handle VM operations without trusting guest VMs and the SubVisor. Specifically, a Guardian-VM requires that the corresponding VM can only invoke functions within a limited range, which is listed in a *jump table*. Moreover, it provides a shadow EPT to the SubVisor for each guest VM and carefully checks the updates made to the shadow EPT by the SubVisor before copying them back to the real EPT. Finally, the Guardian-VM also protects the privacy and integrity of their guest VMs' I/O data.

We have implemented CloudVisor-D based on the Xen 4.5.0 and deployed it on a commodity Intel Skylake machine. The code size of CloudVisor-D (including the RootVisor and Guardian-VM) is roughly equal to that of CloudVisor, which means it does not increase the TCB size. Our evaluation shows that CloudVisor-D significantly improves the performance of nested virtualization. Specifically, the EPT violation handling achieves 85% speedup compared with CloudVisor. Further, CloudVisor-D can efficiently support PV (Para-Virtualization) VMs. It introduces negligible overheads for most benchmarks compared with a vanilla Xen and in some cases outperforms the vanilla Xen due to the reduced number of VM exits.

Contributions. To summarize, this paper makes the following contributions:

- A disaggregated nested virtualization design to shield VMs from an untrusted hypervisor which reduces a large number of VM exits.
- A set of techniques to achieve the same level of security as the nested virtualization.
- Implementation and evaluation of our design on a commercially available machine.

2 Motivation & Background

2.1 Attack Surface of Virtualization Layer

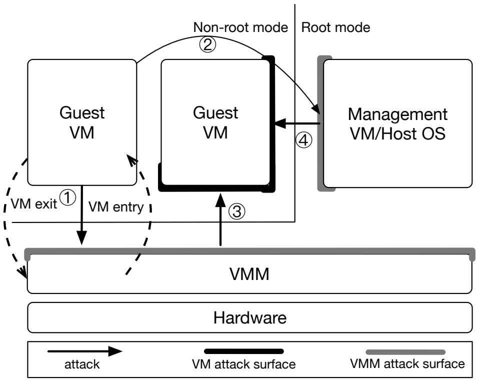


Figure 1: The attack surface in a typical cloud.

Multi-tenant cloud usually adopts virtualization to provision multiple guest VMs atop a single physical machine to maximize resource usage [18,62]. As such, the virtualization layer becomes a key target for attackers to compromise guest VMs. An attacker can exploit vulnerabilities to "jail-break" into the hypervisor, which is Step ① in Figure 1. Such a threat does exist given a large number of vulnerabilities discovered every year with the increasing complexity of the hypervisor layer (Table 1). The attacker can also exploit vulnerabilities to tamper with the host OS (in the case of hosted virtualization) or the management VM (in the case of hostless virtualization) (Step ②). After compomising the hypervisor or the host OS, the attacker can gain control of all other guest VMs (Step ③ and ④).

Operation	Control Flow in Xen	Control Flow in CloudVisor	Times
Hypercall	VM → Xen→ VM	VM → CloudVisor → Xen → ... → CloudVisor → VM	⩾ 2X
EPT violation handling	VM → Xen → VM	VM → CloudVisor → Xen → ... → CloudVisor → VM	2 - 6X
DMA operation	VM → Xen → Dom0 → Xen → VM	VM → CloudVisor → Xen → CloudVisor → Dom0 → ... → CloudVisor → Xen → CloudVisor → VM	⩾ 2X

Table 2: Overhead analysis of VM operations.

2.2 Overheads of Nested Virtualization

To protect guest VMs from the untrusted hypervisor, the nested virtualization approach tries to exclude the hypervisor layer out of the trusted computing base (TCB) and thus provides stronger protection from the vulnerable hypervisor layer. Here, we use CloudVisor [72] as an example to illustrate the details of the nested virtualization and its overheads. One design advantage of CloudVisor is that it separates security protection from resource management. Such separation allows CloudVisor to focus on protection and keep its TCB small while the untrusted hypervisor's TCB is enlarged as more functionalities are continuously added to it.

CloudVisor introduces a tiny nested hypervisor in the most privileged level (root mode) and deprivileges the Xen hypervisor and the host OS (Dom0) to the guest mode (non-root mode). The nested hypervisor interposes all communications between the Xen hypervisor and guest VMs. CloudVisor guarantees that the Xen hypervisor is unable to access a guest's memory and disk storage. Therefore, CloudVisor effectively resolves the threats in the untrusted hypervisor. Yet, the nested virtualization incurs a large number of VM exits and introduces large overhead for I/O operations involving excessive VM exits [72].

Table 2 lists a set of example operations which are commonly used in a virtualized system.

Hypercall: Each hypercall firstly gets trapped into CloudVisor, which forwards this hypercall into the Xen hypervisor for processing, as shown in Figure 2 (a). During this process, the hypervisor may execute sensitive instructions (e.g., CPUID) or access guest's memory, either of which will cause a VM exit. When the hypervisor finishes processing, it tries to resume the guest and triggers another VM exit into CloudVisor. Therefore, as shown in Table 2, a hypercall in CloudVisor introduces at least twice as many ring crossings as that in Xen, causing non-trivial overheads for each hypercall.

EPT Violation: The control flow of EPT violation handling in CloudVisor is similar to the hypercall operation, as shown in Figure 2 (a). One EPT violation first traps the VM into CloudVisor, which then lets Xen handle this violation. CloudVisor disallows Xen to access guests' memory by configuring its EPT (extended page table). During the handling of the guest's EPT violation, any modification to the guest's EPT causes a new EPT violation, which is trapped to CloudVisor and handled by it. In the worst case, modifying the whole 4-level EPT pages causes 4 extra ring crossings. As shown in Table 2, there are at most 6 times as many ring crossings as that in Xen for EPT violation handling.

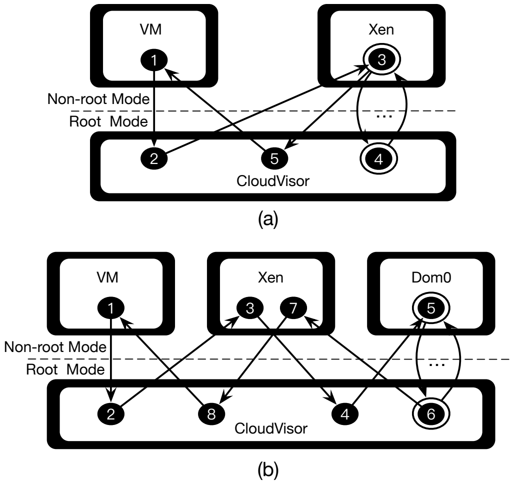


Figure 2: Figure (a) shows the control flows of hypercall operation and EPT violation handling in CloudVisor. Figure (b) shows the control flow of I/O operation in CloudVisor.

I/O Operation: CloudVisor only supports emulated I/O devices. It intercepts all interactions among guest VM, Xen hypervisor and Dom0 to do encryption or decryption (Figure 2 (b)). Therefore, it causes at least twice ring crossings. Since the Dom0 is untrusted and unable to access guest's memory, it triggers one VM exit when it reads (writes) data from (to) the guest memory when handling I/O. That means the whole I/O operation causes more than twice as many ring crossings as that in Xen, as shown in Table 2.

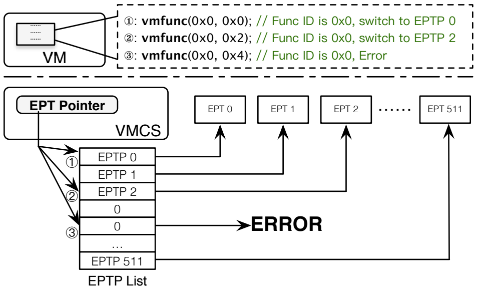
2.3 Advances in Hardware Virtualization

There are two trends in the recent advances of the Intel hardware virtualization technology [1]. The first is the lightweight context switch. Current hardware supports a VMFUNC [2] instruction that provides VM functions for non-root guest VM to invoke without any VM exits. EPTP switching is the only VM function currently supported by the hardware, whose function ID is 0. It allows a VM to load a new value for its EPTP and thus establishes a new EPT, which controls the subsequent address translation from GPA (guest physical address) to HPA (host physical address). The EPTP can only be chosen from an EPTP list configured in advance by the hypervisor.

The procedure for using VMFUNC is as follows. In the preparation stage, the hypervisor allocates an EPTP list (a

[1] We do not find any similar hardware trends on other platforms like ARM and AMD. But the CloudVisor-D approach is applicable to these platforms when similar hardware features are available.

4-KBytes page), which contains at most 512 valid EPTP entries. Then the address of the list is written into the guest's VMCS (Virtual Machine Control Structure). During run time, the guest invokes the VMFUNC instruction and uses an EPTP entry index as the parameter. Afterwards, the hardware searches the list and installs the target EPT. If the index is larger than 511 or the selected EPTP entry points to an invalid EPT structure, a VM exit occurs and notifies the hypervisor. Figure 3 is an example of the VMFUNC workflow. When Line ① and Line ② are executed, the EPT pointer in the guest's VMCS will be changed to the EPTP0 and EPTP2. If the argument of VMFUNC is an index pointing to an invalid EPT structure as Line ③ shows, it will trigger a VM exit waking up the hypervisor.



```
①: vmfunc(0x0, 0x0); // Func ID is 0x0, switch to EPTP 0
②: vmfunc(0x0, 0x2); // Func ID is 0x0, switch to EPTP 2
③: vmfunc(0x0, 0x4); // Func ID is 0x0, Error
```

Figure 3: The workflow of VMFUNC.

The EPTP switching function has four essential characteristics. First, the EPTP switching provided by VMFUNC is faster than a VM exit (134 cycles vs. 301 cycles on an Intel Skylake Core i7-6700K processor). Second, when the VPID (Virtual-Processor Identifier) is enabled, VMFUNC will not invalidate any TLB entry. The TLB entries of one EPT are different from those of other EPTs [27]. Thus, there is no need to flush the TLB after invoking VMFUNC. Third, the VMFUNC instruction can be invoked at any protection ring in non-root mode, including Ring 3 (user mode). Fourth, the VMFUNC instruction only changes the EPTP value and does not affect other registers, especially the CR3 register, program counter and stack pointer.

The second trend is to allow a guest to handle its own VM exits. One significant sign of this trend is the new virtualization exception (VE) [2]. If the VE feature is enabled, an EPT violation can be transformed into an exception (Vector 0x14) without any VM exit. Before using the VE, the hypervisor configures the guest's VMCS to enable virtualization exception support and registers a VE information page into VMCS. The guest kernel should prepare a corresponding handler for the new exception and register it into IDT (Interrupt Descriptor Table). During runtime, most EPT violations will be transformed into virtualization exceptions. The VE handler can know the GPA and GVA (guest virtual address) that cause this exception by reading the VE information page, which is filled by the hardware.

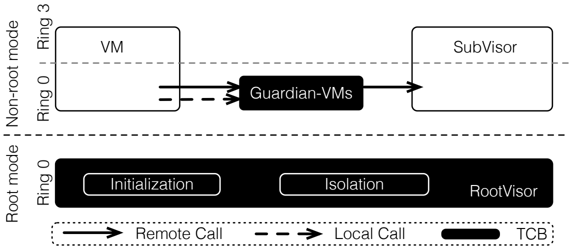


Figure 4: The architecture of CloudVisor-D.

3 CloudVisor-D Approach

3.1 System Overview

For the sake of performance and security, CloudVisor-D has two main goals:

- **Goal-1:** To reduce the number of VM exits caused by the nested virtualization.

- **Goal-2:** To achieve the same level of security as the nested virtualization.

Prior nested virtualization designs intercept all communications between guest VMs and the hypervisor to limit the hypervisor's ability to directly read or write guest VMs' CPU registers, memory pages and disk storages. It consequently incurs large overheads, as we have demonstrated in Section 2.2. The main contribution of CloudVisor-D is to delegate intensively used VM operations to an agent (the Guardian-VM) for each VM in non-root mode to reduce the large number of VM exits (**Goal-1**). CloudVisor-D provides a para-virtualization model for guest VMs to invoke these operations proactively.

Figure 4 is the architecture of CloudVisor-D. CloudVisor-D architecture consists of a tiny nested hypervisor (we call it RootVisor in our paper) in root mode and a set of Guardian-VMs in Ring 0 of non-root mode. The hypervisor is deprivileged to non-root mode and called SubVisor for convenience. The tiny RootVisor has full system privilege and manages all the important data structures such as EPTs. It also sets up a Guardian-VM for each guest VM. All interactions between a guest VM and the SubVisor pass through the corresponding Guardian-VM or the RootVisor. The Guardian-VM is responsible for forwarding and checking most VM operations in non-root mode while the RootVisor is occasionally awakened up to handle some inevitable VM exits in root mode such as external interrupts.

A Guardian-VM is not a full-fledged VM but only contains some service handlers. It supports two kinds of interfaces for guest VMs: the remote call and the local call. Neither of the interfaces causes any VM exit. By using the remote call, a guest can request the SubVisor's services with the help of the Guardian-VM, including the hypercalls and EPT violation handlers. By using the local call, a guest can request the local helper functions in the Guardian-VM. We provide I/O related helper functions that encrypt, decrypt and check data integrity of I/O data.

To achieve (**Goal-2**), we regard CloudVisor-D as a reference monitor [14], which means it should satisfy the following two security properties [29, 30] [2].

- **Tamperproof:** CloudVisor-D isolates the RootVisor and each Guardian-VM and makes their states (including memory and CPU registers) unmodifiable by the corresponding guest VM and the SubVisor.
- **Complete Mediation:** CloudVisor-D (including the tiny RootVisor and the Guardian-VM) interposes all communications between guest VMs and the SubVisor.

To support the **tamperproof** property, CloudVisor-D guarantees the authenticated booting procedure of the RootVisor by leveraging the trusted platform module (TPM) [16] and users could remotely attest the integrity of the RootVisor. Furthermore, the memory address spaces of the RootVisor and all Guardian-VMs are isolated from guest VMs and the SubVisor (Section 4.1).

To enforce the **complete mediation** property, we propose a series of techniques (Section 4.4) to ensure that all communications in non-root mode have to be intercepted and checked by the Guardian-VM while the RootVisor intercepts and monitors the left communication paths that cause VM exits.

Based on the two properties, a Guardian-VM is able to handle VM operations securely in non-root mode. First, one Guardian-VM provides to its VM a limited number of local and remote calls that the VM can invoke (Section 4.5). Second, we introduce a technique to handle EPT violations securely in non-root mode, which guarantees that updates to a VM's EPT by the SubVisor should be verified by the Guardian-VM before coming into effect (Section 5). Finally, Guardian-VMs protects the privacy and integrity of their guest VMs' I/O data (Section 6).

3.2 Threat Model and Assumptions

The only software components CloudVisor-D trusts are the RootVisor and the Guardian-VMs. It also trusts the cloud provider and the hardware platform it runs on. CloudVisor-D distrusts the vulnerable commodity hypervisor, which may try to gain unauthorized access to the guest's CPU states, memory pages, and disk data. CloudVisor-D does not trust the guest VM either since the guest VM can misbehave like trying to escalate its privilege level and attacking other colocated VMs and even the hypervisor. We assume that the guest does not voluntarily reveal its own sensitive data and has already protected sensitive network data via encrypted message channels such as SSL. Finally, we do not consider physical attacks as well as side-channel attacks between different VMs[3].

4 Guardian-VM

In the traditional nested virtualization, a guest VM frequently interacts with the SubVisor to ask it to do VM operations, which forces the VM to trap into the SubVisor. These operations include hypercalls, EPT violation handling and I/O operations. CloudVisor-D provides a Guardian-VM for each guest VM to help them request SubVisor's services without VM exits.

When the RootVisor is booted, it downgrades the SubVisor to non-root mode and creates a SubVisor-EPT for the SubVisor. Then the address translation of SubVisor is controlled by page table (from GVA to GPA) and SubVisor-EPT (from GPA to HPA). The RootVisor removes all its own memory from the SubVisor-EPT to isolate its physcial address space from the SubVisor. The SubVisor is unaware of the existence of the SubVisor-EPT.

Although the SubVisor is in non-root mode, it is still allowed to create guest VMs. When creating a VM, the SubVisor sets up all management data structures for this VM, including an EPT. After that, the SubVisor executes a privileged instruction (i.e., VMLAUNCH in the x86 architecture) to start this new VM, which causes a VM exit trapping the SubVisor to the RootVisor. The RootVisor will not install the EPT initialized by the SubVisor for the guest VM. Instead, the RootVisor treats the original EPT as a shadow EPT and creates a new EPT (called Guest-EPT) by copying all address mappings from the shadow EPT. Therefore, the Guest-EPT maintains the same GPA to HPA mappings as the shadow EPT. Then SubVisor also initializes all other necessary data structures for the VM. After finishing the initialization, the SubVisor installs the Guest-EPT for the guest VM while leaving the shadow EPT unused. The shadow EPT is made read-only for the SubVisor by configuring the SubVisor-EPT. We will discuss more details about the shadow EPT in Section 5.

When the RootVisor initializes a VM, it builds a Guardian-VM for this VM as well. The Guardian-VM has its own ETP called Guardian-EPT. The RootVisor maps code and data pages into the Guardian-VM space by configuring this Guardian-EPT. To isolate the memory of the VM and its Guardian-VM from the SubVisor, the RootVisor not only removes all mappings associated with the memory of the VM and its Guardian-VM from the SubVisor-EPT, but also makes the Guest-EPT and Guardian-EPT inaccessible to the SubVisor.

In the following subsections, we first introduce how CloudVisor-D achieves the **tamperproof** property in Sec-

[2] In fact, the reference monitor model has a third property called "verifiable". Due to the small TCB of CloudVisor-D, it is feasible to completely test and verify CloudVisor-D, which is our future work.

[3] We do not consider recent side-channel attacks like Meltdown [42], Spectre [34] and L1TF [4]. These attacks can be effectively prevented by CPU vendors' microcode patches, which are orthogonal to the CloudVisor-D apporach.

tion 4.1. Then we deconstruct the **complete mediation** property into two more detailed invariants in Section 4.2. Section 4.3 elaborates two attacks that break the two invariants respectively. Section 4.4 explains two techniques that CloudVisor-D uses to enforce the two invariants and further achieve the **complete mediation** property. Finally, we briefly discuss the jump table mechanism in CloudVisor-D.

4.1 Isolating Environment for Guardian-VM

To support the **tamperproof** property, each Guardian-VM runs in an execution environment isolated from its corresponding VM and the SubVisor. Because the RootVisor ensures that the Guest-EPT and the SubVisor-EPT do not contain any memory mappings belonging to the Guardian-VM, neither the guest VM nor the SubVisor is able to access the physical address space of the Guardian-VM. Furthermore, each Guardian-VM also owns a separate stack, which will be installed when a VM or the SubVisor switches into the Guardian-VM. This stack is inaccessible to the guest VM and SubVisor, which ensures that data stored in the separate stack cannot be modified, especially for the runtime states and function arguments. To protect the data in registers, the Guardian-VM clears most general registers to avoid privacy leakage and retains necessary register values (e.g., general registers containing SubVisor function arguments) before switching between a guest VM and the SubVisor.

4.2 Deconstructing the Complete Mediation Property

A guest VM communicates with the SubVisor through two paths. The first one starts with a VM exit and traps to the RootVisor, which then forwards the control flow to the SubVisor. The other path is forwarded by a Guardian-VM to the SubVisor in non-root mode. The **complete mediation** property requires that CloudVisor-D interposes both of the two communications paths. The path in root mode is mediated by the RootVisor, which is enforced by existing techniques [20, 72]. For the communication path in non-root mode, we propose the following invariants which can help achieve the complete mediation property.

- **Invariant 1.** A guest VM must switch to its Guardian-VM before switching to the SubVisor, and vice versa.

- **Invariant 2.** A guest VM (or the SubVisor) enters the Guardian-VM only through the predefined entry points (gates).

Invariant 1 requires that a Guardian-VM intercepts all the communications in non-root mode. Invariant 2 further specifies that a guest VM or SubVisor enter the Guardian-VM only through legal gates, which means they cannot directly jump into other code pages of the Guardian-VM.

4.3 New Attacks to Bypass or Compromise Guardian-VMs

However, it is difficult to enforce these invariants. A straightforward design of the Guardian-VM would enable two types of attacks that break these two invariants respectively. The first attack allows a malicious VM to bypass the Guardian-VM in non-root mode and execute any instructions in the SubVisor, which breaks the **Invariant 1** property. This attack also allows a malicious SubVisor to bypass the Guardian-VM and attack VMs. Specifically, the attacker invokes a self-prepared VMFUNC instruction to maliciously bypass the Guardian-VM by directly switching from one physical space to the target physical space and execute sensitive instructions in the target space. The second attack breaks **Invariant 2** and is simpler than the first one. This attack targets the Guardian-VM and uses techniques similar to the first attack, which bypasses the Guardian-VM's predefined gates and compromises the Guardian-VM.

We first use an example to illustrate the basic procedure of the first attack. We suppose that the guest OS is an attacker, and its purpose is to bypass the Guardian-VM and directly execute any instructions in the SubVisor-EPT (victim). If the attacking direction is reversed, that is, the attacker is the SubVisor and the victim is a guest VM, the attacking procedure is similar. Figure 5 shows an example of the first attack, which consists of the following four steps.

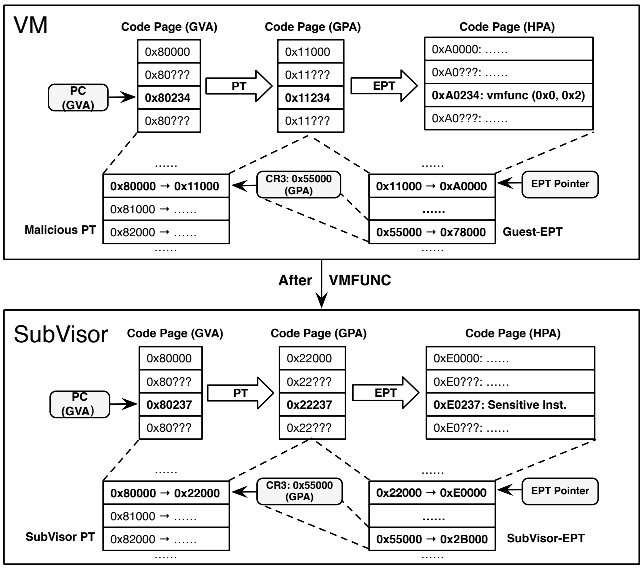


Figure 5: An example of the first attack. All addresses in this figure are used for illustration and do not have any practical meaning.

- **Step 1: Guessing the SubVisor's page table base address.** The attacker guesses the SubVisor's page table base address. The page table controls the mapping from

GVA to GPA, which is managed by the SubVisor. Since the SubVisor usually uses a statically allocated page table which is initialized during system booting, the base address of the page table is easy to guess if the attacker is familiar with the source code of the SubVisor. In Figure 5, the base address of the SubVisor page table is 0x55000.

- **Step 2: Creating a malicious page table.** In the VM's physical address space, the attacker then creates a malicious page table whose base address value (GPA) is equal to that of the SubVisor's page table [4]. Hence, the base address of the malicious page table is 0x55000 in Figure 5. This base address is translated to the malicious page table in the Guest-EPT and to the SubVisor page table in the SubVisor-EPT. The malicious page table consists of four-level page table pages, but each level has only one page. These page table pages translate the GVA of a code page (0x80000 in this example), which contains a self-prepared VMFUNC instruction. The VMFUNC instruction's virtual address is deliberately set to the value just before the GVA of the target instructions in the SubVisor's space, which is 0x80237 in Figure 5.

- **Step 3: Switching EPTs.** The attacker writes the base address of the malicious page table into the CR3 register in non-root mode and executes the self-prepared VMFUNC instruction to bypass the Guardian-VM and switch to the SubVisor-EPT. Here we understand why the attacker needs to guess the SubVisor's page table base address at Step 1. After switching to the SubVisor-EPT, an incorrect value in the CR3 register will be translated to an illegal page table. The illegal page table may contain meaningless GPAs that cause numerous EPT violations, which wake up the RootVisor.

- **Step 4: Executing target instructions.** In the SubVisor-EPT, the GPA in the CR3 register is translated to the HPA of the SubVisor's page table (0x2B000 in this example). Thus, all the GVA of the subsequent instructions will be translated by the SubVisor's page table. Finally, the target instructions are executed.

The second attack is similar to the first one since the attacker also uses the above four steps. The only difference is that the attacking target is Guardian-VM. The attacker similarly crafts a malicious page table and puts the self-prepared VMFUNC instruction just before the GVA of the target instructions in the Guardian-VM. Therefore, the attacker can bypass the predefined gates of the Guardian-VM and breaks **Invariant 2**.

Previous works have proposed many solutions to defend against these attacks. SeCage [44] and EPTI [27] set the code pages belonging to the attacker EPT to non-executable in the victim EPT. SeCage further puts a security checker at the beginning of each sensitive function page. SkyBridge [49] takes another defense solution that first replaces all illegal VMFUNC instructions and then makes code pages non-writable so that the attacker cannot insert self-prepared VMFUNC instructions.

Nevertheless, none of these defenses works in the CloudVisor-D scenario. All of these methods depend on one assumption which is not held in CloudVisor-D: the attacker runs in Ring 3 which means she cannot modify the page table or the CR3 register value. In CloudVisor-D, both of the guest OS and the SubVisor can freely modify their page tables and even CR3 register values. Therefore, previous defenses are unable to defeat this new variant of the attack in CloudVisor-D. Furthermore, CloudVisor-D has one stricter requirement that the guest VM (or the SubVisor) should switch to the Guardian-EPT before the SubVisor-EPT (Guest-EPT).

4.4 Enforcing the Complete Mediation Property

To defeat these attacks and enforce the **complete mediation** property, we propose two techniques that satisfy the two invariants respectively. To enforce Invariant 1, we propose a technique called *dynamic EPTP list manipulation*, which guarantees that both the guest VM and the SubVisor have to enter the Guardian-VM before switching to the target EPT. Another technique to satisfy Invariant 2 is called *isolated Guardian-VM page table*. By using this technique, the malicious guest VM or the SubVisor cannot directly jump into the middle code pages of the Guardian-VM since the base address of the Guardian-VM page table exceeds the GPA ranges of the guest VM and SubVisor.

4.4.1 Dynamic EPTP List Manipulation

A strawman design. One straightforward solution to enforce Invariant 1 is to control the executable bits dynamically in the Guest-EPT and the SubVisor-EPT. Since the Guardian-VM has access to the SubVisor-EPT and corresponding Guest-EPT, it can initialize all code pages in the SubVisor-EPT to non-executable. Hence, the guest OS has to switch to the Guardian-VM and enable the SubVisor's execution privilege before switching to the SubVisor. That gives the Guardian-VM a chance to do the security check. This solution supports fine-grained privilege control, which means it can create multiple SubVisor-EPTs and Guest-EPTs for different vCPUs and enable the executable bits in one SubVisor-EPT/Guest-EPT for one vCPU while keeping other vCPUs' SubVisor-EPTs/Guest-EPTs non-executable. Furthermore, the privilege control can be accelerated by just modifying the L4/L3 EPT entries. However, this solution is infeasible even if it looks reasonable because it requires frequent EPT synchronizations among vCPUs and thus brings

[4] The attacker just puts the page table at a specific GPA. She cannot modify the Guest-EPT.

about a large number of costly TLB shootings [12] for one multi-vCPU VM.

```
 1: Enter_GUARDIAN-VM(func_index, arguments)
 2:         VMFUNC(0x0, 0x1) // install Guardian-EPT
 3:         Guardian_CR3 = Guardian_Info_Page[0]
 4:         Install Guardian_CR3 to CR3 register
 5:         Install Guardian-VM stack
 6:         Push registers
 7:         DISPATCH_REQUESTS(func_index, arguments)
 8:         Pop  registers
 9:         Restore guest stack
10:         Restore guest page table                       Entry
11:         VMFUNC(0x0, 0x0) // install Guest-EPT          Page
12:
13:
14: DISPATCH_REQUESTS(func_index, arguments)
15:         type = VERIFY_REQUESTS (func_index, arguments)
16:         if (is_remote_call == type) then
17:               HANDLE_REMOTE_CALL (func_index, arguments)
18:               CHECK_UPDATES
19:         else if (is_local_call == type)
20:               HANDLE_LOCAL_CALL (func_index, arguments)
21:         else reject the request
22:
23: HANDLE_REMOTE_CALL(func_index, arguments)
24:         EPTP_LIST = Guardian_Info_Page[1]
25:         EPTP_LIST[0] = 0
26:         EPTP_LIST[2] = SubVisor-EPT
27:         func_pointer = jump_table[func_index]
28:         CALL_HYPER_FUNC(func_pointer, arguments)
29:         EPTP_LIST[2] = 0
30:         EPTP_LIST[0] = Guest-EPT
31:
32: HANDLE_LOCAL_CALL(func_index, arguments)
33:         func_pointer = jump_table[func_index]          Main
34:         func_pointer(arguments)                        Page
35:
36:
37: CALL_HYPER_FUNC(func_pointer, arguments)
38:         Install SubVisor page table
39:         Install SubVisor stack
40:         VMFUNC(0x0, 0x2) // install SubVisor-EPT
41:         func_pointer(arguments)
42:         VMFUNC(0x0, 0x1) // install Guardian-EPT
43:         Guardian_CR3 = Guardian_Info_Page[0]
44:         Install Guardian_CR3 to CR3 register           Exit
45:         Restore Guardian-VM stack                      Page
```

Figure 6: The pseudo code of the Guardian-VM.

CloudVisor-D design. Fortunately, we observe that the VMFUNC instruction causes a VM exit if the target EPTP entry in the EPTP list points to an invalid EPT. Therefore, by controlling the EPTP entry in the EPTP list, we propose a new technique called the *dynamic EPTP list manipulation* to ensure that both the guest VM and the SubVisor switch to the Guardian-VM before switching to the other EPT, which thus enforces **Invariant 1**. The intuition behind this technique is that the Guardian-VM dynamically puts and clears the base address of the SubVisor-EPT (or the Guest-EPT) in the EPTP list before entering and after leaving the SubVisor (or the guest VM).

Figure 6 is the pseudocode of the Guardian-VM and Line 24-26 show this technique. The RootVisor shares the EPTP list page with the Guardian-VM, whose address is written in a Guardian-VM private data page by the RootVisor (**Guardian_Info_Page**). By default, most entries in the EPTP list are zero except Entry 0 and 1, which point to the Guest-EPT and the Guardian-EPT respectively. Before calling the SubVisor function, the Guardian-VM clears Entry 0 and then writes the base address of the SubVisor-EPT into Entry 2. When it returns from the SubVisor, the Guardian-VM reversely clears Entry 2 and writes the base address of the Guest-EPT into Entry 0. By using this technique, any illegal EPT switch bypassing the Guardian-VM encounters an EPTP entry with the zero value which causes a VM exit and wakes up the RootVisor to stop the attacker. This technique requires no EPT modification and thus avoids TLB flushing. Furthermore, the VMCS is a per-CPU structure which allows applying the technique to each vCPU independently.

4.4.2 Isolated Guardian-VM Page Table

We do not prevent the attacker from guessing the base address of the Guardian-VM page table. Instead, we prevent installing the Guardian-VM page table. To do that, the RootVisor puts the Guardian-VM page table at a GPA which exceeds the maximum GPA used by the guest VM and the SubVisor. Theoretically, an EPT can support 256TB physical memory that is usually not used up in practice. For example, the maximum GPAs for the SubVisor and guest VMs are smaller than 16GB on our test machine and the RootVisor puts the Guardian-VM page table pages at the GPA larger than 16GB.

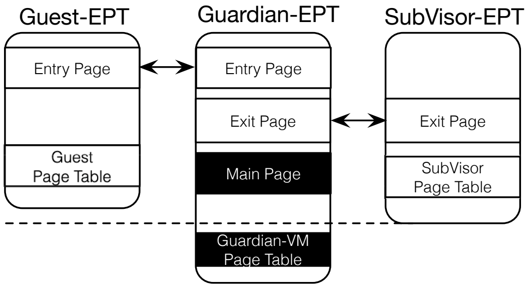


Figure 7: The memory mappings for code pages and page tables.

Figure 7 depicts the memory mapping of code pages. The entry page and the exit page are the two Guardian-VM code pages shared with the Guest-EPT and the SubVisor-EPT respectively. The main page is a private code page of the Guardian-VM. The page table pages used to translate the entry page are shared by the Guest-EPT and the Guardian-EPT. However, the guest does not have the permission to modify these page table pages, which are mapped as read-only in the Guest-EPT. The page table pages used to translate the exit page are similarly mapped into the Guardian-EPT and the SubVisor-EPT. The base address of the Guardian-VM page table is written into the Guardian_Info_Page and the Guardian-VM installs this page table in the entry page, as shown in Figure 6.

This technique effectively prevents the attacker from jumping into the middle of the Guardian-VM. Suppose that there is a malicious VM and it knows the base address of the Guardian-VM page table, it has to create one malicious page table which maps one code page containing at least a VMFUNC instruction. However, the VM is unable to configure the malicious page table whose base address (GPA) is not mapped in the Guest-EPT. Any access to that GPA wakes up the RootVisor. Therefore, a guest VM has to invoke the Enter_GUARDIAN-VM function to enter its Guardian-VM and the SubVisor can enter the Guardian-VM only via returning to the CALL_HYPER_FUNC function.

4.5 Jump Table

CloudVisor-D guarantees that a guest VM invokes a limited range of functions specified in a fixed list, which we call the jump table. The jump table contains the functions in the SubVisor (remote calls) and the local helper functions in the Guardian-VM (local calls). Each entry in the jump table comprises a function pointer and information about its arguments, such as the argument count and their value ranges. The table is not mapped in the Guest-EPT or the SubVisor-EPT so that neither the guest nor the SubVisor can modify it. To invoke a remote call or local call, the guest should provide the index of the function it is calling and corresponding arguments. When processing a guest request, the Guardian-VM verifies the function index and arguments that the guest provides. If the index is out of jump table's range or the number and the value ranges of the arguments do not satisfy those recorded in the jump table, it will reject this request. Otherwise, the Guardian-VM calls a local helper function or redirects it to call a SubVisor function.

5 Memory Virtualization in Non-root Mode

CloudVisor-D handles EPT violation in non-root mode without triggering any VM exit. To achieve this goal, CloudVisor-D leverages the virtualization exception (VE) and converts an EPT violation to a VE in the guest. The guest then issues a remote call of the Guardian-VM to call the EPT violation handler in the SubVisor, which also resides in non-root mode.

When a VE happens, the guest's VE handler is called. By reading the VE information page, it gets the violation GPA and exit qualification. The exit qualification is a technical term used in the Intel manual [2], which describes information about the access causing the exception, such as whether the violation is caused by a data read or write. Then the handler calls a remote call to invoke the EPT violation handler of the SubVisor.

We design a secure guest EPT update mechanism to handle the EPT violation securely in non-root mode: (1) The Guardian-VM grants the write permission of the guest's shadow EPT to the SubVisor by modifying the SubVisor-EPT; (2) The Guardian-VM switches to the SubVisor-EPT and calls the SubVisor's EPT violation handler; (3) The SubVisor traverses the shadow EPT to handle this violation and returns; (4) The Guardian-VM revokes the shadow EPT permission from the SubVisor; (5) The Guardian-VM traverses the shadow EPT to check the updates made by the SubVisor and notifies the RootVisor if anything abnormal is detected; (6) The Guardian-VM applies the updates to the Guest-EPT. Please note that all the above EPT modifications by the SubVisor are made to the shadow guest EPT, which is not actually used by the guest VM. Only after being checked by the Guardian-VM can these updates come into effect.

When checking the updates made by the SubVisor, the Guardian-VM sees the EPT pages that are associated with the violated address and omits other pages. This could boost the checking procedure since there are at most four EPT pages that are used to translate the violated address. The Guardian-VM validates the page ownership when checking the updates. For example, if the SubVisor tries to maps another VM's page to this VM, the Guardian-VM rejects these updates and notifies the RootVisor.

We do not invoke INVEPT here to flush the corresponding TLB entries after handling the EPT violation. This is reasonable because we only consider the EPT violation situation, where all TLB mappings that would be used to translate the violated address are invalidated by the hardware before the VE handler is called [2]. For instance, one read-only TLB entry exists for one page and any write operation to the page triggers one VE which flushes the stale read-only TLB entry before invoking the VE handler.

CloudVisor-D focuses on the EPT violation scenario which increases privileges (e.g., change non-present to present or read-only to writable). It does not shoot down other TLB entries in a multi-core VM to boost the VE handling procedure. The stale TLB entries on other cores only cause extra VEs if accessed by other cores. Furthermore, the Guardian-VM optimizes the VE handling of the stale TLB entries by directly returning to the guest VM without forwarding the VE to the SubVisor.

Other EPT management operations: The SubVisor may modify guest VM's EPT for other management purposes, such as memory deduplication and NUMA page migration. These management operations are handled like CloudVisor, which still trigger EPT violations and trap into the RootVisor.

Faking VE Attack: One guest VM may issue a fake VE by intentionally making a remote call to invoke the SubVisor EPT handling procedure. The fake VE lures the Guardian-VM to map other VMs' or the SubVisor's pages into the attacker's EPT and make these pages accessible to the guest. However, the Guardian-VM disallows such modifications to the attacker's EPT since it checks page ownership before modifying any page mapping and will not grant one page to the attacker if it belongs to other VMs or the SubVisor.

6 I/O Protection

It is critically important to protect the privacy and integrity of the virtual disk of a guest virtual machine. The most straightforward strategy is to encrypt the whole disk in the guest kernel level, like LUKS [25]. However, the malicious SubVisor can steal the encryption key, peek into or tamper with the plaintext in memory. Further, it also mandates the guest VM with the support of LUKS, which is not always available. Therefore, CloudVisor-D provides the full virtual disk encryption support efficiently and mostly-transparently

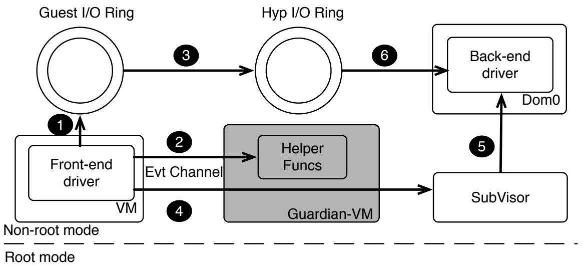


Figure 8: The PV disk I/O request handling process in CloudVisor-D.

at the cloud level. To support para-virtualization I/O model which is widely used in today's cloud environment, we insert two lines of code into the PV front-end driver in the guest OS that call helper functions in Guardian-VM. These helper functions encrypt (or decrypt) I/O data of the guest and update (or verify) the hash values of the disk. CloudVisor-D uses the AES-XTS algorithm in Intel AES-NI to encrypt and decrypt disk data with a 128-bit AES key. The key is generated by the tenant and encrypted by a platform public key provided by CloudVisor-D. Then the user passes the encrypted key to CloudVisor-D through the network. Afterwards, the key cipher-text is decrypted and maintained inside the CloudVisor-D memory.

Overall Control Flow: Figure 8 is our solution for PV I/O protection. When the front-end driver is initializing, CloudVisor-D creates a SubVisor I/O ring for the back-end driver in the SubVisor. The SubVisor I/O ring is editable by the SubVisor, while the original one is inaccessible to it. Suppose the front-end I/O driver is ready to issue an I/O write request. Before it pushes the request into I/O ring, it invokes the Guardian-VM's sending helper function via a local call, which allocates a new buffer and copies the data of the request into the buffer (This copy is omitted for the read request). Then the Guardian-VM encrypts all pages in the copied buffer and updates corresponding hash values of related sectors. Finally, it writes the new buffer into the SubVisor I/O ring and modifies the SubVisor-EPT to change these new buffer pages' permission to *writable*. Next, the front-end driver pushes the request to the ring and invokes a remote call to send an event to the back-end driver under the help of the Guardian-VM. When the front-end driver receives a virtual completion interrupt from the back-end driver, it invokes the receiving helper function via the other local call to process the response and revoke the buffer permission from the SubVisor-EPT. If it is a read request, the Guardian-VM also copies data from the buffer into the guest OS request pages, and decrypts the data in these pages.

Data Integrity: We compute a 256-bit SHA-256 hash value for each disk sector and use the Merkle tree [48] to organize the hash values of all disk sectors. This hash tree is stored in a hash file and loaded into a shared memory of CloudVisor-D by Xen management tool (xl) when we boot a guest VM. Even though a compromised xl program may modify the hash value of storage, CloudVisor-D can detect that situation since the hash values are generated based on the decrypted sector data which xl is unable to access without the AES key passed by the user.

DMA Attack: An attacker may access sensitive memory or even inject code into CloudVisor-D memory by leveraging DMA operations. To defend against this attack, CloudVisor-D controls IOMMU and makes protected memory regions inaccessible to the SubVisor by manipulating the mapping from device address to HPA. The IOMMU page table for the storage device controlled by the SubVisor only contains physical addresses that do not belong to any VMs. Each time a new VM is booted, the RootVisor removes mappings related with this new VM from the IOMMU page table for the device. Therefore, when the malicious SubVisor issues a DMA request to write or read VM memories, an IOMMU page fault triggers, which notifies the RootVisor.

7 Security Analysis

7.1 CloudVisor-D as a Reference Monitor

CloudVisor-D is actually a reference monitor which mediates all communications between guest VMs and the SubVisor. There are two necessary and sufficient requirements for a secure reference monitor, which are tamperproof and complete mediation. In this section, we first explain how CloudVisor-D satisfies these two requirements.

Property 1 (tamperproof): The RootVisor is trusted during its lifetime. The integrity of the RootVisor is guaranteed by the authenticated boot of TPM, by which users can attest whether the RootVisor is trusted. After booted, potential attackers cannot modify the RootVisor's code or data since it has an isolated address space, which is inaccessible to the SubVisor and VMs. The RootVisor also has the full privilege of the hardware and prevents attackers from disabling key hardware features like the virtualization feature.

Property 2 (tamperproof): Guardian-VMs are tamperproof during its lifetime. Based on Property 1, the trusted RootVisor can securely load a trusted Guardian-VM when booting a guest VM. The RootVisor also checks its integrity when finishing the booting process. During run time, the guest VM and the SubVisor do not have the privilege to modify the memory and EPT of the Guardian-VM. Therefore, a malicious VM or SubVisor is unable to touch any sensitive memory states of a Guardian-VM directly. However, since Guardian-VMs accept inputs from untrusted VMs and SubVisor, the Guardian-VM and the RootVisor must protect themselves from malicious inputs, which may exploit a stack overflow vulnerability and then mount a ROP attack. Memory bugs are unavoidable for software written in C/C++ languages. However, due to the small TCB of Guardian-VM, it is relatively easy to verify that Guardian-VMs are free of these memory vulnerabilities. Furthermore, we have used three static analysis tools (Facebook infer v0.15.0 [10], CBMC v5.3 [36] and Cppcheck v1.72 [23]) to check the current implementation of CloudVisor-D. Both Facebook infer and Cppcheck found some instances of three types of bugs (uninitialized variables, possibly null pointer dereferences, and dead stores) while CBMC did not report any bugs. We have fixed all the reported bugs. However, none of these tools could prove that the implementation of CloudVisor-D is bug-free. We plan to use formal verification methods to verify CloudVisor-D or completely rewrite it by using high-level and secure languages like Rust [46] in the future.

Property 3 (complete mediation): CloudVisor-D intercepts all communications There are two types of paths that a VM or the SubVisor can communicate with each other. The first is via the VM exits which are then forwarded by the RootVisor, which is the traditional and slow path. The other one is through the Guardian-VM. An attacker may try to bypass Guardian-VMs by directly switching from a VM to the SubVisor. This attack is prevented by controlling the EPTP list entries and the isolated Guardian-VM page table. Thus, the only way to enter the SubVisor in non-root mode is through the Guardian-VM, which accepts a limited range of functions recorded in the jump table. A VM may refuse to call the interface provided by Guardian-VM. But it is in an isolated EPT environment, which means this behavior only results in its own execution failure, not affecting other VMs or the SubVisor.

7.2 Defend VMs against an Untrsuted Hypervisor

Due to the tamperproof and complete mediation properties of CloudVisor-D, we ensure that a guest VM (or the SubVisor) cannot tamper with CloudVisor-D nor bypass it, and any communication path between VMs and the SubVisor is mediated by CloudVisor-D. In this section, we explain how CloudVisor-D protects guest VMs based on the secure reference monitor concepts.

Protecting CPU states for guest VMs The CPU registers of one VM can only be modified by the RootVisor or its Guardian-VM. CloudVisor-D will clear unnecessary register values when switching between VMs and the SubVisor. The SubVisor cannot compromise the normal execution of guest VMs since it is forbidden from directly changing the CR3, RIP and RSP registers.

Protecting Memory states for guest VMs CloudVisor-D prevents a malicious SubVisor (or a malicious guest VM) from accessing the memory of any VMs by controlling the EPTs to enforce the memory isolation. The SubVisor may try to modify the guest's EPT and maps the guest's memory into the SubVisor's EPT when it handles EPT violations. This can also be defeated since any modification to the shadow guest EPT made by the SubVisor is checked by the Guardian-VM which prevents such dangerous mappings. The SubVisor could attempt to leverage a DMA capable device to access the VM memory and even compromise CloudVisor-D. This is prevented by controlling IOMMU to make the protected memory regions inaccessible for the SubVisor.

Protecting Disk I/O states for guest VMs CloudVisor-D also guarantees the privacy and integrity of guest VMs' disk I/O data. The SubVisor is able to access the disk image file directly. But the image contains encrypted data, which is meaningless if not decrypted. Furthermore, CloudVisor-D protects the encryption key in its memory and registers, and the attacker cannot steal the key to decrypt the I/O data. The SubVisor may also modify the encrypted disk file, which could be detected by CloudVisor-D by comparing the hash values.

8 Evaluation

This section evaluates CloudVisor-D's overall performance and scalability by answering the following questions:

Q1: What is the implementation complexity of CloudVisor-D?

Q2: Does CloudVisor-D improve the performance of the micro-architectural operations (e.g., hypercalls)?

Q3: How do real-world applications perform under CloudVisor-D?

Q4: Does CloudVisor-D achieve good I/O performance?

Q5: How does CloudVisor-D perform when running multiple instances of guest VMs?

Q6: Can CloudVisor-D defend against malicious VMs or SubVisor?

8.1 Methodology

Name	Description
apache	Apache v2.4.7 Web server running ApacheBench v2.3 with the default configuration, which measures the number of handled requests per second serving the index page using 100 concurrent clients to send 10,000 requests totally
mysql	MySQL v14.14 (distrib 5.5.57) running the sysbench oltp benchmark using 6 threads concurrently to measure the time cost by an oltp test, the size of oltp table is 1000000 and the oltp test mode is complex mode
memcached	memcached v1.4.14 using the memcslap benchmark on the same VM, with a concurrency parameter of 100 to test the time it takes to load data
kernel compile (kbuild)	kernel compilation time by compiling the Linux 4.7.0 from scratch with the default configuration using GCC 4.8.4-2
untar	untar extracting the 4.7.0 Linux kernel tarball compressed with gzip compression using the standard tar utility, measuring the time cost
hackbench	hackbench v0.39-1 using unix domain sockets and 100 process groups running with 500 loops, measuring the time spent by each sender sending 500 messages of 100 bytes
dbench	dbench v4.0 using different numbers of clients to run I/O Read/Write tests under empty directories with default client configuration repeatedly

Table 3: Description of real applications.

In this section, we demonstrate the efficiency of CloudVisor-D by comparing it with the vanilla Xen hypervisor (v4.5.0). Our test machine is equipped with an Intel Skylake Core i7-6700K processor, which has 4 cores and 8 hardware threads with the hyper-threading enabled. The storage device is a 1TB Samsung 860 EVO SATA3 SSD.

All the benchmarks we used and their setup details are described in Table 3. The Dom0 is Debian 8.9 and the kernel is Linux 4.4.80. We used Ubuntu 16.04 for the guest virtual machine and Linux 4.7.0 as its kernel. The guest has 1 (a UP VM) or 2 (an SMP VM) vCPUs, 2GB virtual memory and 30GB virtual disk. All multicore evaluations were done using two vCPUs bound to two physical CPUs. To ensure the evaluation results measured at the same CPU clock, we disabled the CPU frequency scaling.

8.2 Status Quo and Complexity

To answer the first question (**Q1**), we have built a prototype of CloudVisor-D on an Intel Skylake machine. CloudVisor-D uses the Intel AES-NI [2] for encryption and leverages IOMMU to defend against DMA attacks (Section 6). Table 4 shows the breakdown of CloudVisor-D TCB, which is measured by the sloccount tool [6]. The code sizes of the RootVisor and Guardian-VM are 4,174 and 1,656 respectively. The sum is roughly equal to that of CloudVisor, which means CloudVisor-D does not increase the TCB size.

	Functionality	LOC
RootVisor	VMCS Manipulation	1,742
	Memory Management	1,397
	Exit Handlers	583
	Other	452
Guardian-VM	Reference Monitor	429
	Encryption	574
	Hash Integrity	653
CloudVisor-D	Total	5,830

Table 4: The breakdown of CloudVisor-D TCB.

8.3 Micro-architectural Operations

Operation	**Xen**	**CloudVisor**	**CloudVisor-D**	**Speedup**
Hypercall	1758	4681	1810	61.3%
EPT violation handling	5374	66301	9929	85.0%
Virtual IPI	11214	21344	13331	37.5%

Table 5: Micro-architectural operation overhead measured in cycles.

To answer the second question (**Q2**), we quantified the performance loss of micro-architectural operations of the hypervisor on an SMP virtual machine. Table 5 presents the costs of various micro-architectural operations in an SMP VM. The results are measured in cycles.

Hypercall is an operation commonly used by the guest kernel to interact with the hypervisor. To measure its performance, we call a **do_vcpu_op** hypercall to check whether a vCPU is running or not. In the Xen hypervisor, this hypercall causes two VM ring crossings: a VM exit and a VM entry. Even if CloudVisor-D causes more EPT switches, it can achieve similar performance via the efficient remote calls. A hypercall in CloudVisor incurs almost 3 times as many cycles due to a large number of ring crossings as we have analyzed in Section 2.

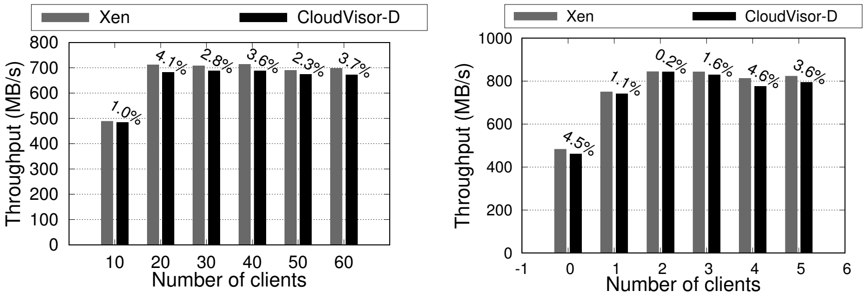
EPT violation handling is the total cost of switching to the SubVisor, handling the EPT violation and returning to the guest. We invalidated one GPA in the guest EPT and measured the procedure of reading a value in the address, which involves an EPT violation handling. The result is an average of 5,000 tests. The cost of this operation in CloudVisor-D is larger than that in Xen due to the manipulation of EPT in Guardian-VM introduced in Section 5. In CloudVisor, the SubVisor causes two VM ring crossing each time it modifies the guest EPT, which introduces multiple VM ring crossings when handling EPT violations. Therefore, it performs the worst, which is nearly 10 times worse than Xen and CloudVisor-D.

Virtual IPI is the cost of issuing an IPI to another vCPU. We pinned two vCPUs to different physical CPUs. Virtual IPI is an important operation intensively used in the multi-core machines. The measured time starts from sending an IPI in one vCPU until the other vCPU responds. In Xen hypervisor, a virtual IPI is implemented by sending an event using the event channel to the SubVisor, which then injects a virtual interrupt to the target vCPU. CloudVisor-D replaces the **do_event_channel_op** hypercall with a remote call to allow one vCPU to send an event without any VM exit. Yet, we did not optimize the virtual interrupt sending procedure which is our future work. Even if CloudVisor-D is slower than Xen, it is significantly faster than CloudVisor due to the efficient remote calls.

8.4 Applications Performance

To answer **Q3**, we measured CloudVisor-D with real-world applications which have various execution characteristics. Since CloudVisor only supports emulated I/O devices, it is unfair to directly compare it with CloudVisor-D, which supports a PV I/O device model. Moreover, the vanilla Xen has been shown to outperform CloudVisor. Therefore, we directly compared CloudVisor-D with the vanilla Xen, which is sufficient to demonstrate CloudVisor-D performance.

Figure 11(a) shows the result of the performance comparison of CloudVisor-D on real applications with the vanilla Xen hypervisor in a uniprocessor VM. CloudVisor-D performs similarly to the vanilla Xen hypervisor across all workloads. The maximum overhead is not larger than 5%. We also evaluated these applications in an SMP VM. Figure 11(b) shows the normalized performance of real applications in an SMP VM. For these real-world applications, CloudVisor-D still incurs negligible overhead. It even performs better than the vanilla hypervisor, especially for the memcached benchmark. Benchmarks such as memcached incur many event channel communications in an SMP setting which is optimized by CloudVisor-D by using the efficient remote calls. To check the impact of this optimization, we ran a guest VM with and without using the **do_event_channel_op** remote call and compared their performance. As shown in Table 6, a guest without **do_event_channel_op** remote call suffers from severe performance degradation, which means the **do_event_channel_op** remote call improves performance



(a)

(b)

Figure 9: Throughput for dbench in UP (left) and SMP (right) VMs using from 10 to 60 concurrent clients. The numbers above each bar are the CloudVisor-D overhead compared with the vanilla Xen. (Higher is better)

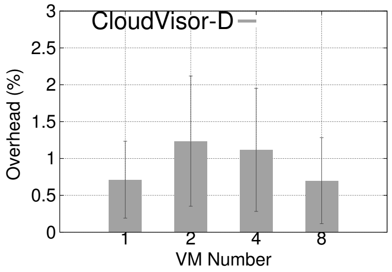


Figure 10: Performance overhead for kernel building in CloudVisor-D compared to the vanilla Xen for different number of concorrent VMs. (Lower is better)

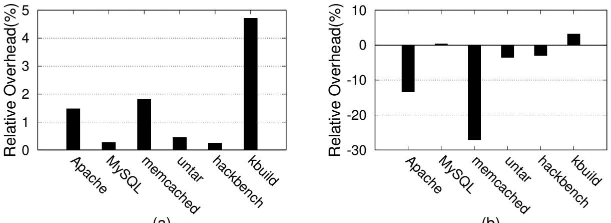


Figure 11: Performance overhead for real applications in UP (left) and SMP (right) VMs. The data on the left bar shows the relative overhead compared to a vanilla Xen hypervisor (Lower is better). In the right figure, the bar below the line (with zero overhead) represents that CloudVisor-D outperforms the vanilla hypervisor while the bar above the line means that CloudVisor-D is slower than the vanilla.

a lot for the memcached benchmark. Specifically, it improves the performance of CloudVisor-D for about 78.32% (27.05%+51.27%).

Experiment	Vanilla Xen	CloudVisor-D–	CloudVisor-D
Time (seconds)	7.613	11.516	5.554
Speedup	0	-51.27%	27.05%
VM exits	1,691,758	4,572,269	63,909

Table 6: The performance impact of remote calls on memcached. CloudVisor-D– means the guest does not invoke do_event_channel_op remote call while CloudVisor-D means the guest uses this remote call.

The reason for this speedup is as follows: memcached is a multi-threaded application and has no problem saturating many cores. In an SMP VM, one vCPU frequently sends virtual IPIs to another vCPU, which is implemented by the event channel mechanism. With the help of the remote calls, CloudVisor-D reduces numerous VM exits caused by invoking **do_event_channel_op** hypercall, resulting in much less unnecessary scheduling. Moreover, a vCPU will not send any virtual IPI if it detects the target vCPU is not idle, which further avoids VM exits caused by virtual IPIs. We found that CloudVisor-D decreases the number of VM exits from 1,882,098 to 60,921 compared to the vanilla Xen hypervisor, as shown in Table 6. Therefore, memcached in CloudVisor-D achieves better performance than that in the vanilla Xen.

Overheads of a Guardian-VM. Each tenant VM only requires one Guardian-VM, which is not a complete VM but only a few service handlers. A Guardian-VM is invoked on demand. It introduces only 108KB memory for one vCPU (116KB for two vCPUs), costs at most 3.39% CPU cycles when running real-world apps used in our paper.

8.5 I/O Performance

To answer **Q4**, we studied how CloudVisor-D behaved in the worst-case I/O scenario by using dbench v4.0 [1]. dbench is a widely-used I/O-intensive benchmark. In our evaluation, the sysstat [11] tool reveals that I/O activities (including file system time and waiting for the block device) account for 87.99% of the total workload time. Figure 9(a) demonstrates the result of I/O performance overhead on dbench in a UP VM by changing the number of concurrent clients. When the number of concurrent clients is smaller than 20, the throughput does not reach its limit which is approximately 710 MB/s. The overhead for storage I/O is smaller than 5% for all cases. Since dbench is a worst-case I/O scenario benchmark, the result demonstrates that even in the worst case, CloudVisor-D can provide acceptable I/O performance. The I/O performance in an SMP VM is similar to that in a UP VM, as shown in Figure 9(b). CloudVisor-D achieves negligible overhead across different concurrency levels.

8.6 Performance of Multiple VMs

Finally, to answer the scalability question (**Q5**), we demonstrated how CloudVisor-D performs by running kbuild under the different numbers of VMs. Figure 10 shows the performance overhead of concurrently running kbuild on the different number of VMs. All these VMs are protected by

CloudVisor-D. The result is an average value of 10 runs. Each VM has one vCPU, 512MB memory and one 15GB virtual disk. In CloudVisor-D, most VM operations are delegated to the Guardian-VMs and each guest VM has its own Guardian-VM, which is not shared by others. Therefore, CloudVisor-D incurs negligible overhead on multiple VMs. Considering the small overhead of this experiment, the worse performance in the case of 2 VMs could be attributed to runtime variation.

8.7 Security Evaluation

According to the CVE analysis for the Xen hypervisor in Nexen [54], the consequences of different attacks can be classified into DoS (we do not consider this), privileged code execution, information leakage, and memory corruption. CloudVisor-D can be used as a last line of defense such that it does not directly fix security vulnerabilities but instead prevents exploitation of them from having harmful effects.

We conducted two experiments to show that CloudVisor-D can protect guest VMs against memory writes (or reads) from the malicious SubVisor, which is usually the ultimate goal of many attack means. In the first experiment, the malicious SubVisor tries to read or write one VM's memory page. The guest reserves one page and then the malicious SubVisor modifies the page. This attack succeeds in the vanilla Xen but fails in CloudVisor-D in which any access to the VM's memory triggers one EPT violation caught by the RootVisor. In the second experiment, the malicious SubVisor modifies the VM's EPT, maps one code page into the VM's physical memory space and maps the page into the VM's virtual space. Similar to the previous attack, this one succeeds in the vanilla Xen but fails in CloudVisor-D.

We also conducted two more experiments to show that the Guardian-VM can defeat the malicious EPT switching attack. First, we simulated a malicious VM that bypasses the Guardian-VM and executes code in the SubVisor. The VM installs a malicious page table whose base address value identical to that used in the SubVisor and then invokes a VMFUNC to switch to the SubVisor-EPT directly. However, since the target EPTP entry is 0 in the EPTP list, this attack fails when the VM invokes the VMFUNC instruction that triggers one VM exit. In the second attack, the malicious VM leverages the four steps (Section 4.3) to jump to the middle of the Guardian-VM. But the attack fails when it tries to configure the malicious page table which triggers one VE. The Guardian-VM then notifies the RootVisor to terminate the VM.

9 Discussion

VMFUNC and Virtualization Exception in Modern Hypervisors. Modern hypervisors (e.g., Xen and KVM) have already used the VMFUNC instructions and virtualization exception (VE) in various use cases. The first typical use case for using VMFUNC and VE is to monitor VM behaviors [9] (Virtual Machine Introspection, VMI) and track memory accesses by restricting the type of access the VM can perform on memory pages. Once the monitored VM violates the memory permission configured in its EPT, one VE triggers a handler which then uses a VMFUNC instruction to switch to a monitoring application's EPT. Another use case of VMFUNC and VE is to boosting network function virtualization (NFV) [3]. In NFV, each network function resides in a different VM. NFV heavily depends on inter-VM communications. To boost the NFV communication, one network function uses the VMFUNC instruction to switch to an alternate EPT and directly copy network data to another VM's memory. These use cases do not conflict with CloudVisor-D because CloudVisor-D only occupies 3 EPTP entries in the EPTP list, leaving 509 free entries for other usages, like boosting VMI and NFV.

Directly Assigned PCIe Devices. The current version of CloudVisor-D provides no support for SR-IOV devices. Fortunately, many cloud providers disabled SR-IOV devices due to the incompatibility with live VM migration. However, the design of CloudVisor-D can be extended to protect VMs if using directly assigned PCIe devices and SR-IOV. First, the RootVisor leverages the IOMMMU to limit the physical space each assigned device can access. The physical function of the SubVisor is limited by the IOMMU page table as well, which means it cannot freely access other VMs' spaces. Second, before writing data into the assigned device, a guest OS should invoke a helper function in its Guardian-VM to encrypt the data. For reading data, the guest OS first issues a DMA request to move encrypted data from the device to a private memory buffer, and then invokes a helper function in the Guardian-VM to decrypt the data.

10 Related Work

Hardware-based Secure Computation: Secure architectures have been extensively studied during the last decades [21, 37, 38, 40, 41, 43, 45, 51, 55, 59, 59, 60, 67–71]. Besides, different mainstream processor manufacturers recently presented their products that support memory encryption. AMD (SEV [32]) and Intel (SGX [13, 47]) have presented their memory encryption products to the market respectively. Researches proposed to leverage Intel SGX to shield software [15, 19, 24, 28, 52, 57] or harden the SGX itself [53, 56]. Haven [19] and SCONE [15] use SGX to defend applications and weakly isolated container processes from software and hardware attacks. Ryoan [28] provides an SGX-based distributed sandbox to protect their sensitive data in data-processing services. M2R [24] and VC3 [52] allow users to run distributed MapReduce in the cloud while keeping their data and code secret.

Defending against Untrusted Hypervisor: Many studies have considered how to defend guest VMs against possibly untrusted hypervisor. One prominent solution is to leverage architectural support to remove the hypervisor out of TCB. For example, H-SVM [31] modifies hardware to intercept each Nested Page Table (NPT) update from the hypervisor to guarantee the confidentiality and integrity of the guest VM. HyperWall [60] forbids the hypervisor from accessing the guest's memory by modifying the processor and MMU. Another approach is to decompose the hypervisor and move most of its part to the non-privileged level. NOVA [58] proposes a microkernel-like hypervisor. Xoar [22] decomposes the Dom0 into nine different service VMs to achieve stronger isolation and smaller attack surface. Similarly, Nexen [54] deconstructs Xen hypervisor into a shared privileged security monitor and several non-privileged service slices to thwart vulnerabilities in Xen. HyperLock [64] and DeHype [66] isolate the hypervisor from the host OSs. HypSec [38] leverages the ARM virtualization extension and TrustZone technique to decompose a monolithic hypervisor into a small trusted corevisor and a big untrusted hypervisor, which effectively reduces the TCB.

Even though we also propose a disaggregated design, CloudVisor-D is different from the previous solutions in three ways. First, CloudVisor-D separates the tiny nested hypervisor, not the commodity hypervisor which has been totally excluded out of the TCB. Second, while previous solutions require intensive modifications to the commodity hypervisor, CloudVisor-D makes much fewer modifications (less than 100 LOC) to the commercial hypervisor and is completely compatible with it. Finally, CloudVisor-D utilizes new x86 hardware features to efficiently and securely connect the isolated parts, which boosts the nested virtualization in the x86 architecture.

Researchers also proposed to leverage the same privilege protection for untrusted hypervisor, to harden the hypervisor itself by measuring integrity [17] or enforcing control-flow integrity [63] of the hypervisor. However, these approaches are best effort ones and do not exclude the commodity hypervisor out of the TCB.

Nested Virtualization: Traditional nested virtualization [20] uses "trap and emulate" model to capture any trap of the guest and forward it to the hypervisor for processing. CloudVisor-D puts frequent normal VM operations to an agent in non-root mode to replace the heavy "trap and emulate". Different from turtles project [20], CloudVisor [72] distrusts the hypervisor and prohibits it from accessing security-sensitive data of guest VMs. Since nested virtualization technology incurs unacceptable overheads, Dichotomy [65] presents the ephemeral virtualization to reduce this overhead, but it does not intend to defend against the malicious hypervisor.

VMFUNC-based Systems: Even though there are some previous researches that leverage VMFUNC to implement user-level memory isolation [27, 35, 44] or efficient communication facilities [26, 39, 49], all these systems assume that a malicious VMFUNC user cannot modify the CR3 register, which is not the case in CloudVisor-D. We propose a new variant of the malicious EPT switching attack and a series of techniques to defeat it. Furthermore, CloudVisor-D is the first design to utilize this hardware feature to build a disaggregated nested hypervisor to defend VMs against an untrusted hypervisor efficiently.

11 Conclusions

CloudVisor-D is a disaggregated system that protects virtual machines from a malicious hypervisor. It leverages nested virtualization to deprivilege the Xen hypervisor and offloads most VM operations to secure Guardian-VMs without the intervention of the tiny nested hypervisor (RootVisor). CloudVisor-D has been implemented for Xen-based systems and introduces negligible overhead.

12 Acknowledgments

We sincerely thank our shepherd Vasileios Kemerlis and all the anonymous reviewers who have reviewed this paper in the past two years. We also would like to thank Xinran Wang, Weiwen Tang, Ruifeng Liu, and Yutao Liu. This work was supported in part by the National Key Research & Development Program (No. 2016YFB1000104), the National Natural Science Foundation of China (No. 61525204, 61772335), and research grants from Huawei and SenseTime Corporation. Haibing Guan is the corresponding author.

References

[1] Dbench filesystem benchmark. https://www.samba.org/ftp/tridge/dbench/.

[2] Intel 64 and ia-32 architectures software developer's manual volume 3c. https://software.intel.com/en-us/articles/intel-sdm.

[3] Intel corporation, extending kvm models toward high-performance nfv. https://www.linux-kvm.org/images/1/1d/01x05-NFV.pdf.

[4] Intel developer zone. l1 terminal fault. https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault.

[5] Kvm cve. https://nvd.nist.gov/vuln/search.

[6] Sloccount. https://dwheeler.com/sloccount/.

[7] Vmware advisories list. https://www.vmware.com/security/advisories.html.

[8] Xen cve. https://xenbits.xen.org/xsa/.

[9] Xen project blog. stealthy monitoring with xen altp2m. https://blog.xenproject.org/2016/04/13/stealthy-monitoring-with-xen-altp2m.

[10] Facebook open source. facebook infer. https://fbinfer.com, 2019.

[11] Sysstat: Performance monitoring tools for linux. http://sebastien.godard.pagesperso-orange.fr/, 2019.

[12] N. Amit. Optimizing the tlb shootdown algorithm with page access tracking. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, Berkeley, CA, USA, 2017.

[13] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for cpu based attestation and sealing. 2013.

[14] J. P. Anderson. Computer security technology planning study. Technical report, Anderson (James P) and Co Fort Washington PA, 1972.

[15] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. Scone: Secure linux containers with intel sgx. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, Berkeley, CA, USA, 2016.

[16] W. Arthur and D. Challener. *A practical guide to TPM 2.0: using the Trusted Platform Module in the new age of security*. Apress, 2015.

[17] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. Hypersentry: Enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, New York, NY, USA, 2010.

[18] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, New York, NY, USA, 2003.

[19] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *11th USENIX Symposium on Operating Systems Design and Implementation*, Broomfield, CO, Oct. 2014.

[20] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The turtles project: Design and implementation of nested virtualization. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, Berkeley, CA, USA, 2010.

[21] D. Champagne and R. B. Lee. Scalable architectural support for trusted software. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, Jan 2010.

[22] 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 *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, New York, NY, USA, 2011.

[23] Cppcheck. Cppcheck a tool for static c/c++ code analysis. http://cppcheck.sourceforge.net/, 2019.

[24] T. T. A. Dinh, P. Saxena, E.-C. Chang, B. C. Ooi, and C. Zhang. M2r: Enabling stronger privacy in mapreduce computation. In *Proceedings of the 24th USENIX Conference on Security Symposium*, Berkeley, CA, USA, 2015.

[25] C. Fruhwirth. Luks on-disk format specification version 1.2.3. 2018.

[26] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA, July 2019.

[27] Z. Hua, D. Du, Y. Xia, H. Chen, and B. Zang. EPTI: Efficient defence against meltdown attack for unpatched vms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, Boston, MA, July 2018.

[28] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, Berkeley, CA, USA, 2016.

[29] T. Jaeger. Operating system security. *Synthesis Lectures on Information Security, Privacy and Trust*, 1(1):1–218, 2008.

[30] T. Jaeger. Reference monitor. In *Encyclopedia of Cryptography and Security*, 2011.

[31] S. Jin, J. Ahn, S. Cha, and J. Huh. Architectural support for secure virtualization under a vulnerable hypervisor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, New York, NY, USA, 2011.

[32] D. Kaplan, J. Powell, and T. Woller. Amd memory encryption. 2016.

[33] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. Nohype: Virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, New York, NY, USA, 2010.

[34] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[35] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the Twelfth European Conference on Computer Systems*, New York, NY, USA, 2017.

[36] D. Kroening and M. Tautschnig. Cbmc – c bounded model checker. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[37] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang. Architecture for protecting critical secrets in microprocessors. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, Washington, DC, USA, 2005.

[38] S.-W. Li, J. S. Koh, and J. Nieh. Protecting cloud virtual machines from hypervisor and host operating system exploits. In *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA, Aug. 2019.

[39] W. Li, Y. Xia, H. Chen, B. Zang, and H. Guan. Reducing world switches in virtualized environment with flexible cross-world calls. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, New York, NY, USA, 2015.

[40] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2000.

[41] D. Lie, C. A. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, New York, NY, USA, 2003.

[42] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Conference on Security Symposium*, Berkeley, CA, USA, 2018.

[43] Y. Liu, Y. Xia, H. Guan, B. Zang, and H. Chen. Concurrent and consistent virtual machine introspection with hardware transactional memory. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 416–427. IEEE, 2014.

[44] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2015.

[45] W. Mao, H. Chen, J. Li, and J. Zhang. Software trusted computing base, May 8 2012. US Patent 8,176,336.

[46] N. D. Matsakis and F. S. Klock, II. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, New York, NY, USA, 2014.

[47] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, New York, NY, USA, 2013.

[48] R. C. Merkle. Protocols for public key cryptosystems. In *1980 IEEE Symposium on Security and Privacy*, pages 122–122. IEEE, 1980.

[49] Z. Mi, D. Li, Z. Yang, X. Wang, and H. Chen. Skybridge: Fast and secure inter-process communication for microkernels. In *Proceedings of the Fourteenth EuroSys Conference 2019*, New York, NY, USA, 2019.

[50] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein. Eleos: Exitless os services for sgx enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems*, New York, NY, USA, 2017.

[51] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin. Using address independent seed encryption and bonsai merkle trees to make secure processors os- and performance-friendly. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, Washington, DC, USA, 2007.

[52] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2015.

[53] J. Seo, B. Lee, S. M. Kim, M. Shih, I. Shin, D. Han, and T. Kim. Sgx-shield: Enabling address space layout randomization for SGX programs. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017.

[54] L. Shi, Y. Wu, Y. Xia, N. Dautenhahn, H. Chen, B. Zang, and J. Li. Deconstructing xen. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017.

[55] W. Shi and H.-H. S. Lee. Authentication control point and its implications for secure processor design. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, Washington, DC, USA, 2006.

[56] M. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: eradicating controlled-channel attacks against enclave programs. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017.

[57] M.-W. Shih, M. Kumar, T. Kim, and A. Gavrilovska. S-nfv: Securing nfv states by using sgx. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, New York, NY, USA, 2016.

[58] U. Steinberg and B. Kauer. Nova: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems*, New York, NY, USA, 2010.

[59] G. E. Suh, D. Clarke, B. Gassend, M. v. Dijk, and S. Devadas. Efficient memory integrity verification and encryption for secure processors. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, Washington, DC, USA, 2003.

[60] J. Szefer and R. B. Lee. Architectural support for hypervisor-secure virtualization. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2012.

[61] C.-C. Tsai, D. E. Porter, and M. Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, Berkeley, CA, USA, 2017.

[62] H. Wang, P. Shi, and Y. Zhang. Jointcloud: A cross-cloud cooperation architecture for integrated internet service customization. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, June 2017.

[63] Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2010.

[64] Z. Wang, C. Wu, M. Grace, and X. Jiang. Isolating commodity hosted hypervisors with hyperlock. In *Proceedings of the 7th ACM European Conference on Computer Systems*, New York, NY, USA, 2012.

[65] D. Williams, Y. Hu, U. Deshpande, P. K. Sinha, N. Bila, K. Gopalan, and H. Jamjoom. Enabling efficient hypervisor-as-a-service clouds with ephemeral virtualization. In *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, New York, NY, USA, 2016.

[66] C. Wu, Z. Wang, and X. Jiang. Taming hosted hypervisors with (mostly) deprivileged execution. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*, 2013.

[67] Y. Wu, Y. Liu, R. Liu, H. Chen, B. Zang, and H. Guan. Comprehensive vm protection against untrusted hypervisor through retrofitted amd memory encryption. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 441–453. IEEE, 2018.

[68] Y. Xia, Y. Liu, and H. Chen. Architecture support for guest-transparent vm protection from untrusted hypervisor and physical attacks. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 246–257, Feb 2013.

[69] Y. Xia, Y. Liu, H. Guan, Y. Chen, T. Chen, B. Zang, and H. Chen. Secure outsourcing of virtual appliance. *IEEE Transactions on Cloud Computing*, 5(3):390–404, July 2017.

[70] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin. Improving cost, performance, and security of memory encryption and authentication. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, Washington, DC, USA, 2006.

[71] F. Zhang and H. Chen. Security-preserving live migration of virtual machines in the cloud. *Journal of network and systems management*, 21(4):562–587, 2013.

[72] F. Zhang, J. Chen, H. Chen, and B. Zang. Cloudvisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, New York, NY, USA, 2011.