# HyperPS: A Virtual-Machine Memory Protection Approach through Hypervisor's Privilege Separation

| | |
|---|---|
| Journal: | *Transactions on Dependable and Secure Computing* |
| Manuscript ID | Draft |
| Manuscript Type: | Regular Paper |
| Keywords: | Virtual Machine Protection, Security, KVM |
| | |

# HyperPS: A Virtual-Machine Memory Protection Approach through Hypervisor's Privilege Separation

Kunli Lin, Wenqing Liu, Kun Zhang, and Bibo Tu,

**Abstract**—The HostOS or Hypervisor constitutes the most important cornerstone of today's commercial cloud environment security. Unfortunately, the HostOS/Hypervisor, especially the QEMU-KVM architecture, is not immune to all vulnerabilities and exploitations. Recently, researchers have put forward lots of schemes to protect Virtual Machines under the compromised HostOS/Hypervisor. However, some of these schemes rely on special hardware facilities, while other (e.g., Nested Virtualization schemes) require large modification to current commercial cloud architecture.
In this paper, we present a novel scheme, named HyperPS, to implement virtual machine protection under the compromised HostOS/Hypervisor. The key idea of HyperPS is to deprive the HostOS/Hypervisor of the privileges of managing the physical memory into an isolated and trusted execution environment. We have implemented a fully functional prototype based on the KVM in Intel x86_64 architecture. Experiment results show that HyperPS has achieved an acceptable trade-off between security and performance.

**Index Terms**—Virtual Machine Protection, Security, KVM.

## 1 Introduction

Integrating the hypervisor capabilities into a standard Linux kernel, the KVM-based virtualized environment not only benefits from all the ongoing work on the Linux kernel itself, but also shares countless vulnerabilities with it. A successful exploit, especially a privilege escalation exploit, against the Linux kernel will subvert the entire virtualization environment. It is time to consider that how to protect Virtual Machines if the HostOS or the Hypervisor is no longer trusted. Recently, researchers have proposed a lot of schemes to mitigate such a security concern.

Hardware facilities, such as Memory Protection eXtensions(MPX) [22], Encryption Instructions(AES-NI) [12] , Software Guard eXtensions(SGX) [18] and Secure Memory Encryption(SME) [14], provide efficient and strong memory isolation. These hardware facilities have already been actively used by researchers to separate the HostOS kernel components and the hypervisor component. However, most of these hardware facilities restrict access to the partition application to a narrow interface. This limitation makes it extremely tough to leverage these hardware facilities to separate the entire virtualization component from the kernel. Furthermore, most of these hardware facilities need developers to reconstruct their protected applications or rebuild the whole program from scratch. There are also some schemes that propose to separate the virtualization component from the rest of the kernel by reconstructing the HostOS or Hypervisor. Microhypervisor ( [4], [11], [15], [17], [25], [27], [35]) advocates that the hypervisor should only be responsible for core virtualization privileges to reduce its interaction with the kernel. For example, Trustvisor and Nova adopt microhypervisor to achieve virtualization privilege separation and memory protection. Nested virtualization is another representative approach. Nested virtualization ( [6], [9], [20], [23], [32], [34], [37]) introduces a higher privilege level and isolated execution environment beyond the original hypervisor. Thus, separated privileges in the nested virtualization environment can no longer be subverted by the original one. For example, CloudVisor [39] proposed to use nested virtualization to decouple resource management components into a nested hypervisor. Nevertheless, These schemes have been deemed incompatible for commercial cloud providers in that cloud providers could not endure the significant performance losses introduced by nested virtualization, let alone reconstructing their cloud environment architecture.

Extended Page-Table Technology (EPT) is introduced by Intel to support the virtualization of physical memory. When EPT is in use, guest-physical addresses are translated by traversing a set of EPT paging structures to produce physical addresses that are used to access memory. Everything in the guest virtual machine's memory cannot be safe if the HostOS/Hypervisor has been compromised. For example, a compromised HostOS/Hypervisor could remap one guest physical page frame to another physical page frame that holds the malicious code. Extended Page Table Pointer (EPTP) is a hardware register that contains the address of the base of EPT table, as well as EPT configuration information. Virtual Machine Control Structure (VMCS) is the most important structure in a virtualization environment. It manages transitions into and out of virtual-machine extensions (VMX) non-root operations (VM entries and VM exits) as well as processor behavior in VMX non-root operation. The value of EPTP is configured and managed by

- *Bibo Tu was with Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China. E-mail:tubibo@iie.ac.cn*
- *Kunli Lin, Wenqing Liu, Kun Zhang and Bibo Tu are with Chinese Academy of Sciences.*

VMCS. If the HostOS/Hypervisor has been subverted, the value of EPTP will no longer be safe. The attacker can load a new EPTP, thereby establishing a dedicated and malicious EPT Paging-structure hierarchy.

In this paper, we introduce a novel scheme, named HyperPS, to implement Virtual Machines protection against compromised HostOS or hypervisor. The key idea of HyperPS is to deprive the HostOS/Hypervisor of the privileges of managing the physical memory, and replace these privileges into an isolated and trusted execution environment alongside the HostOS kernel. Thus, even the HostOS/hypervisor has been compromised, we can still provide a strong protection for virtual machines's memories. HyperPS does not rely on any special hardware or need to reconstruct current commercial cloud architecture. Actually, HyperPS shares the same privilege level with the HostOS kernel and can be compatible to current commercial cloud architecture.

We have implemented a full funtional prototype based on the KVM in Intel x86_64 architecture. In our prototype, we separate the privilege of managing these two structures: VMCS (especially the EPTP filed in this structure) and EPT, from the HostOS/Hypervisor into an isolated and trusted execution environment. All access during VM exit (All VMX root operations) to these two structures will be hooked and be redirected to the isolated execution environment. In other word, code in the original HostOS/Hypervisor environment can no longer read or write VMCSs and EPTs directly. Our prototype modified 300 SLOC (Source Lines of Code) of the original Linux kernel and introduced approximately 4K SLOC of new kernel code. We also conducted several security and performance experiments. The results show that HyperPS has gained an acceptable trade-off between security and performance.

## 2 Motivation

Among the many services that cloud service providers need to provide to cloud tenants, ensuring the integrity and security of virtual machines is the key factor in gaining user trust. For each VM, the cloud service provider needs to ensure that the user's business in the VM will not be maliciously tampered with, and the data will not be stolen. For different VMs, cloud service providers need to ensure effective isolation between different VMs. However, in the QEMU-KVM architecture, Linux as Hostos is not only a hypervisor that is responsible for providing virtualization services, but also a normal operating system with full functions. In a real business deployment, even if cloud service providers adopt customized Linux as the Hostos, the Linux kernel still contains a large number of various subsystems that have nothing to do with virtualization. These subsystems, especially kernel drivers with complex sources, inevitably contain countless exploitable vulnerabilities. These subsystems provide the attacker with a huge attack surface. Even if the attacker does not find any exploitable QEMU or KVM vulnerabilities, he can still tamper with the virtualization component by exploiting the vulnerabilities of other Linux subsystems. Therefore, cloud service providers need a further think about how to ensure the security of VMs under the premise that the Hostos/Hypervisor has been compromised. This is the motivation of our HyperPS.

## 3 Background and Threat Model

### 3.1 Background

#### *3.1.1 QEMU-KVM*

QEMU is a generic and open-source machine emulator and virtualizer. QEMU can use other hypervisors, like XEN or KVM, to use CPU extensions for virtualization. When used as a virtualizer, QEMU achieves near native performances by executing the guest code directly on the host CPU. Kernel-based Virtual Machine (KVM) is an open-source virtualization technology that converts Linux into a type-1 (bare-metal) hypervisor. KVM is a part of the Linux kernel that shares all the Linux kernel's operating system-level components -such as the memory manager, process scheduler, security manager, and more to run VMs. Every VM is implemented as a regular Linux process, scheduled by the standard Linux scheduler, with dedicated virtual hardware like a network card, memory, and disks. KVM mainly consists of a loadable kernel module, kvm.ko, that provides the core virtualization infrastructure and a processor specific module, kvm-intel.ko or kvm-amd.ko. In a virtualization environment, KVM does not work on its own. It is only an API provided by the kernel for userspace. End users typically use KVM through QEMU, where it is present as an acceleration method. For the QEMU-KVM architecture, KVM interacts with QEMU (QEMU runs at the user space) in two ways: through device file `/dev/kvm` and through memory mapped pages. Memory mapped pages are used for bulk transfer of data between QEMU and KVM. `/dev/kvm` is the main API exposed by KVM. It supports a set of `ioctls` which allow QEMU to manage VMs and interact with them.

#### *3.1.2 VMCS*

Virtual-machine Control Structure is a data structure used in Virtual Machine eXtensions (VMX). It controls VMX non-root operations (guest virtual machine execution operations) and VMX transitions. Access to the VMCS is managed through the VMCS pointer (one per logical processor). The VMCS pointer is read and written using the instructions `VMPTRST` and `VMPTRLD`. In general, the hypervisor configures a VMCS using `VMREAD`, `VMWRITE`, and `VMCLEAR` instructions. A hypervisor could use a different VMCS for each virtual machine that it supports. For a virtual machine with multiple logical processors, the hypervisor could use a different for each logical processor. A logical processor may also maintain a number of VMCSs that are active, however, at any given time, at most one of the active VMCSs is the **current** VMCS. VMX instructions operate only on the **current** VMCS.

A compromised Hostos/hypervisor can force the guest virtual machine exit by tramper VM-Execution Control fields and VM-exit control fields in VMCS and tramper the guest virtual machine by writing Guest-State Area fields.

### 3.1.3 EPT

Intel implements its Second Level Address Translation (SLAT) as Extended Page Table (EPT). EPT allows each virtual machine to manage its page table (not the EPT), without giving access to the underlying host machine's MMU Hardware. Thus, EPT reduces the need for hypervisor to keep syncing the shadow pages eliminating the overhead. If EPT is enabled, guest-physical addresses are translated by traversing a set of EPT Paging-structures to produce physical addresses that are used to access memory. In addition to translating a guest-physical address to a physical address, EPT specifies the privileges that software is allowed when accessing the address. Attempts at disallowed accesses are called EPT violations and cause VM exits.

A compromised Hostos/hypervisor could tramper EPT Paging-structures so that the virtual machine will execute arbitrary malicious code without knowing it. Moreover, a compromised Hostos/Hypervisor could access any data in the guest virtual machine with the help of virtual machine introspection.

### 3.1.4 EPTP Management

The extended-page-table pointer (EPTP) contains the address of the base EPT table, as well as other EPT configuration information. The value of EPTP is stored in the VM-Execution Control Fields of VMCS. The hypervisor can manage and configure EPTP value in VMX root operation, while Intel also provides `VMFUNC` to allow software in VMX non-root operation to load a new value for the EPTP without VM-Exit. EPTP switching is VM function 0. EPTP switching allows software (in both kernel and user mode) in guest VM to directly load a new EPTP, thereby establishing a different EPT Paging-structure hierarchy. The value of EPTP can only be selected from a list of pre-configured values in advance by the hypervisor.

## 3.2 Attack Surface

In this section, we first present how attackers tamper virtual machines through exploiting vulnerabilities in Hostos/Hypervisor. Then, we present several typical attack models/examples targeting at VMCS and EPT. At last, we explain some additional attacks (not attacks relative to virtualization components)

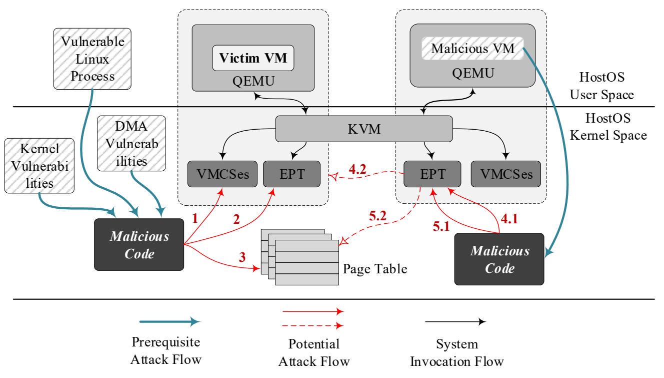

Fig. 1: The Attack Paths in the QEMU-KVM Cloud Environment

### 3.2.1 How attacker subvert VM

As illustrated in Section 3.1, Hypervisor manages VM's physical memory through EPT, and it configures and manages EPTs through the EPTP field in VMCS. Therefore, these two data structures: VMCS (especially the EPTP field) and EPT, become the key target for an adversary to tamper VMs. In this paper, we emphasize that the attacker does not settle for the denial of service attack, but attempts to inject malicious code into the victim VM. Therefore, the momory, both it is access permissions and mapping relationships, is the most important target.

As depicted in Figure 1, An adversary can exploit vulnerabilities to "jail-break" into the Hostos/Hypervisor, while he can also subvert Hostos/Hypervisor by exploiting vulnerabilities in the Hostos (both the kernel and the user process). In this paper, we hypothesize the Hostos/Hypervisor has already been compromised, we attempt to protect VMs under the compromised Hostos/Hypervisor. After compromising the Hostos/Hypervisor, the attacker will inject malicious code to subvert VMs by tampering EPTP field in VMCSes and EPTs. However, we do not restrict how attackers can compromise VMCSes or EPT. For example, an attacker can tamper any fields in VMCSes or EPT with `VMX` instructions, if he has gained the root processor privilege. An attacker can also tamper these two data structures through memory write operations. We assume that the attacker can write these two data structures either through Direct Memory Access (DMA) or through regular memory access.

As mentioned above, we emphasize memory security, so that we have detailed the various attack paths of attackers against EPT in Figure 1. An attacker can tamper with the EPTP field in VMCS directly, thereby establishing a dedicated and malicious EPT Paging-structure hierarchy, which is the Attack Path 1 in Figure 1. The attacker can also tamper with EPT Paging-structures directly (Attack Path 2). However, in some scenarios, the attacker does not have the ability to subvert the victim VM's VMCSes or EPT directly. In some cases, the attacker can only tamper with page tables, which is Path 3 in Figure 1. However, the VM's memory in the QEMU-KVM environment is not only managed by the EPT, but also managed by the Hostos's page tables (QEMU process's page tables). An attacker could subvert the victim VM by tampering with the corresponding QEMU process's page table entries. The situation would be extremely severe when the VM (QEMU process) requests a new memory area or delivers a page fault interruption (in this situation, Hostos page tables are involved because of page fault, EPT is involved because of establishing new EPT Paging-structure. In other cases, an attacker may only be able to modify the EPT Paging-structures of a malicious VM (Attack Path 4 and Attack Path 5 in Figure 1). The attacker could break the memory isolation between VM's by exploiting memory or page table vulnerabilities. For example, VMs with same guestos kernel may share the same kernel code memory frames. An attacker may subvert the victim VM by exploiting Copy-On-Write (COW) vulnerabilities in Hostos.

### 3.2.2 Attack Examples

We present several attacks to illustrate how an attack subvert VMs by through tampering VMCS and EPT.

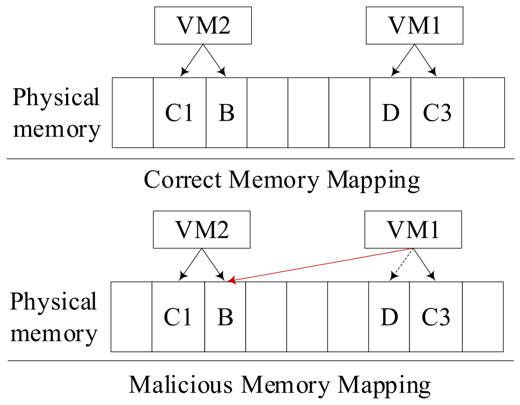

Fig. 2: Diagram of Remapping Attack

**VMCS Subversion**: Attackers can tamper fields of VMCS in VM Exit stage. For example, modifying the value of `HOST_RIP` register and writing a malicious program address to this register will cause a control flow hijacking attack. Modifying the value of privilege register, CR0, will close the DEP mechanism, and modifying the value of CR4 register will close the SMEP mechanism.

**EPT Subversion**: Malicious modification of EPT will break the isolation between virtual machines. A vicious VM can break the isolation between VMs by tampering with EPT entries, and access the memory of the victim VM at his vicious will. For example, the attacker can conduct remapping attack and double mapping attack to inspect a victim VM.

For the double mapping attack, the attacker first controls and compromises a VM, then obtains the privilege of the hypervisor through the virtual machine escape attack, and maliciously accesses the VMCS structure to obtain the value of EPTP. The attack process is as shown in Figure 2.

In this way, the EPT address of the attacker virtual machine, VM1, and the victim virtual machine, VM2, are respectively obtained. And for a guest virtual address in VM2, A, the corresponding real physical address is B. For VM1, the real physical address corresponding to the guest virtual address C is D, then D is modified to be B by modifying the value of the last page item of EPT. Then VM1 can access the data of VM2 successfully, this process is called address double mapping.

For the remapping attack, there are VM1 (attacker) and VM2 (victim). A physical page (A) used by VM2 is released after being used. After A is released, VM1 remaps to A. So that the guest virtual address of VM1 points to the physical page A. By this way, VM1 can access the information on A used by VM2, causing information leakage.

**Attacks to Kernel Page Tables**: We also take into account the fact that attacker already knows the deployment of HyperPS. As depicted in Figure 1, key components, such as Kernel Page Tables, Control Registers, to create the secure and isolated execution environment are also protected by us. More details about the creation of the secure and isolated execution environment are illustrated in Section 5.1.

### 3.3 Threat Model

In this paper, We assume that the Hostos/Hypervisor has been compromised and controlled by the powerful adversary. The adversary can turn off kernel security mechanisms, such as DEP, SMEP, SMAP, and so on. The adversary can tamper fields of VMCSs and EPTs with dedicated malicious values. The adversary can also tamper VMCSs and EPTs through DMA write or regular memory access to them.

We assume that the adversary does not possess the capability to conduct side channel attack and Hardware-based attack. We also assume that the adversary is unwilling to conduct the Denial of Service attack (DOS). In this paper, we assume that hardware resources are trusted, including processor, buses, BIOS/UEFI, and so on.

## 4 Desgin

In this section, we first propose the architecture of HyperPS. Then, in the following subsection, we elaborate on how HyperPS stripped the compromised HostOS kernel's privilege of managing guest VM's memory and the physical memory. Finally, we illustrate how HyperPS manages the guest VM's memory and physical memory to resist the compromised HostOS/Hypervisor.

### 4.1 HyperPS Overview

In a virtualization environment, the HostOS/Hypervisor deprivileges the guest VM's kernel and interposes all interactions between guest VMs and the physical memory. If the HostOS/Hypervisor has been compromised, both isolation between VMs and virtual-physical mapping relationships for a VM will inevitablely be tampered. Thus, in this paper, we present HyperPS to deprive the Hypervisor of privileges on managing physical memory.

Figure 3 depicts the details on the architecture of HyperPS. Firstly, the compromised HostOS/Hypervisor is stripped of privileges of managing physical memory. As shown in Figure 3, original functions about physical memory management (e.g., EPT operations, EPTP switching operations, and some VMCS operations) in the compromised Hypervisor are hooked into the HyperPS Space. However, hooking these functions is far from sufficient to deprive the HostOS/Hypervisor of the capabilities of managing physical memory completely. There are still some extraordinary cases in which attackers can subvert memory management data structures. For example, attackers can inject newly malicious VMX assembly code so that all hooked functions will be bypassed. The attacker can also access memory management data structures by using regular memory access if he gets their locations in advance. HyperPS, in consideration of these cases, removed VMCS and EPT from the original Hypervisor space and placed them in the HyperPS Space. Secondly, in the HyperPS Space, we introduce two new data structures, called Page-Mark structure and VM-Mark structure, to tag the mapping relations between the physical frames and the virtual machines. With the help of these two data structures, even if an attacker can tamper EPT Paging-structures with malicious value through a legal function invocation, HyperPS can still effectively detect such attacks and resist them. Lastly,as shown in Figure 3, we create a delicate kernel-level secure and isolated execution space, called

HyperPS Space, to inherit privileges of managing physical memory that originally belonged to the compromised Hypervisor. We adopt a set of techniques to guarantee the isolation of the HyperPS Space. Context switching from the compromised HostOS kernel/Hypervisor to the HyperPS Space exclusively passes through a designated Switch Gate. Actually, the Switch Gate is the only interface between the Normal Space and the HyperPS Space.

### 4.2 Privilege Separation

EPT defines a layer of address translation that augments the translation of linear addresses. When EPT is in use, the EPT interposes all memory access between guest VMs and the physical memory. Certain addresses that would normally be treated as physical addresses (and used to access memory) are instead treated as guest-physical addresses. Guest-physical addresses are translated by traversing a set of EPT paging structures to produce physical addresses that are used to access memory. If attackers have gained control over a Hypervisor, neither isolation between VMs nor virtual-physical mapping relationships in a VM are immune to them. In this paper, HyperPS deprives the Hypervisor's privileges of managing physical memory by limiting it from accessing and managing all EPTs. Figure 4 depicts the difference in the VM-Exit handling between the traditional system and the system with HyperPS.

Firstly, HyperPS removed all EPTs and VMCSes from the Hypervisor space. EPT manages all physical memory of the VM, as mentioned above, HyperPS removed them from the Hypervisor space. However, there is a hardware register, called Extended Page Table Pointer (EPTP), that contains the address of the base of EPT table, as well as EPT configuration information. Simple obliteration of EPT would make EPTP error which will crash the whole virtualization environment too. Besides, the VMCS records the value of EPTP in the VM-Execution Control Fields. The processor will load the value in that field into the hardware register when VM-Entry. Even if the attacker cannot directly tamper with EPTs, he can also subvert the guest VMs by tampering with a malicious EPTP value in the VMCS. Thus, HyperPS removed all VMCSes from the Hypervisor too.

Secondly, HyperPS hooks all VMCS operation and EPT operation functions in the Hypervisor into the HyperPS Space. In the QEMU-KVM architecture, The KVM provides a wrapper around privileged instructions and is responsible for executing privileged instructions. The QEMU does not need to deal architecture specific details, it just needs to invoke KVM functions with proper parameters. HyperPS hooks KVM's VMCS and EPT functions into the HyperPS Space. In the HyperPS Space, Context Management in the HyperPS Space Management component checks the invoked parameters and verifies if the inputted parameters are legal or not. Since VMCS is hidden in the HyperPS Space which is accessible by functions in the Hypervisor space, all context management (accessing VMCS operations) will be trapped int HyperPS Space inevitably. EPT operation functions are also hooked into the HyperPS Space too. EPT operation functions are different from VMCS operation functions, for the VMCS can only be accessed by just a few privileged instructions. Instead of hooking EPT accessing functions, HyperPS hooks all EPT operation functions. Functions about EPT creation, load, traversal, update and destruction are re-placed into the HyperPS Space. Functions that invoke these EPT operation functions are hooked into the HyperPS Space.

Lastly, HyperPS also takes DMA attacks into consideration. An attacker with the ability of arbitrary memory access by exploiting DMA vulnerabilities can tamper VMCSes and EPTs. IOMMU carries out access control for DMA access. Thus, in this paper, HyperPS employs IOMMU mechanism to resist DMA attacks. In the Hypervisor space, HyperPS found out all the critical data used by IOMMU, and removed the corresponding page table entries that map these data from the page tables. In this paper, HyperPS mainly defines the entrance address of HyperPS, the Page-Mark data, and the VM-Mark data (details about Page-Mark and VM-Mark data structure are illustrated in Section 4.3.1) as the critical data. In the HyperPS Space, HyperPS intercepts the address mapping function about I/O. At runtime, HyperPS verifies whether the address belongs to the HyperPS Space on receiving signals of executing these IO functions.

### 4.3 Privilege Management

HyperPS has removed EPTs and VMCSes from the original Hypervisor space, and functions about accessing and updating them have also been hooked into the HyperPS Space. The HostOS/Hypervisor has already been deprived of privileges of managing the physical memory directly. Actually, the HostOS/Hypervisor can not access EPTs and VMCSes with regular memory access instructions or DMA instructions. However, The HostOS/Hypervisor could still retrieve details about EPTs and VMCSes through legal functions. The attacker can also tamper with EPT Paging-structures and VMCS fields with malicious input to these hooked functions. In this paper, HyperPS present two tables : Page-Mark Table and VM-Mark Table, to tag the mapping relations between the physical memory page frames and the virtual machines. Page-Mark Table consists of Page-Mark structure, while VM-Mark Table consists of VM-Mark structure.

#### *4.3.1 EPTP Protection*

In this paper, we propose a table: VM-Mark Table to restrict Hypervisor from loading an unverified EPTP value arbitrarily. The VM-Mark Table consists of VM-Mark structures. In the Hypervisor, a new EPTP means a different EPT Paging-structure hierarchy. An attacker can map VM virtual address to a totally new physical page frame that holds malicious code. In this paper, HyperPS hooks all functions that can switch the value of EPTP and verifies if the value is authorized by HyperPS before. HyperPS relies on the VM-Mark structure to identify whether the operation of changing EPTP is legal or not.

Table 1 presents the basic architecture of the VM-Mark structure. The field `VMID` is a magic number randomly generated when the VM is created. The field `EPTID` is also a magic number randomly generated when the EPTP is allocated. In a virtualization environment, the Hypervisor can use a different VMCS for each virtual machine that it supports. However, these VMCSes share the same EPTP value for a single VM. HyperPS, thus, blinds the different

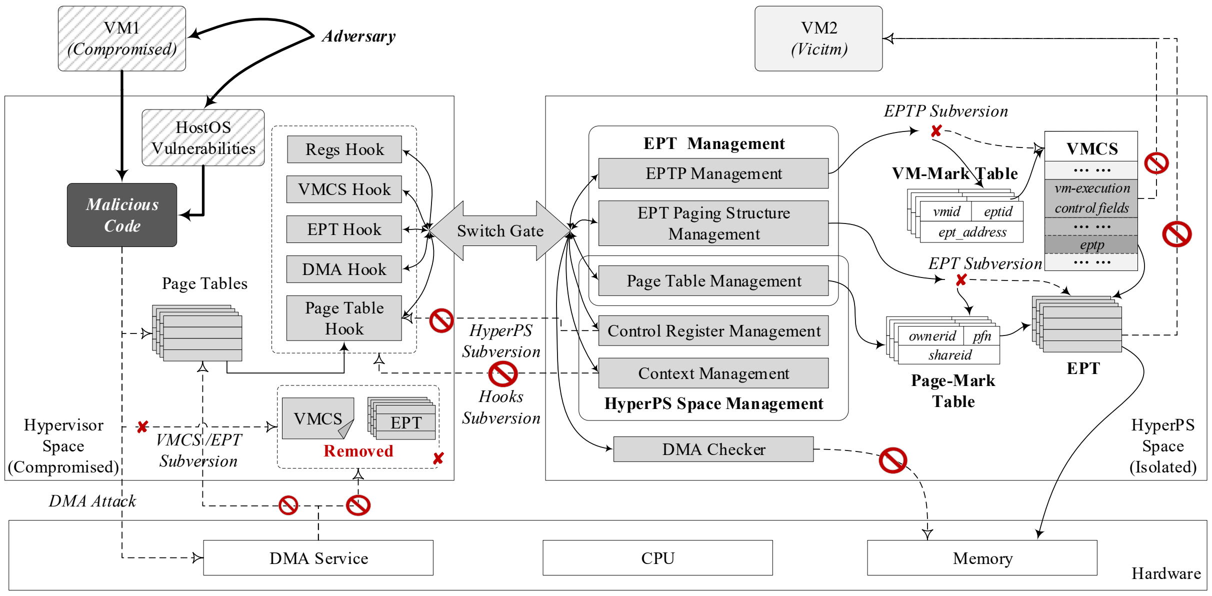

Fig. 3: HyperPS Architecture
HyperPS is committed to protect guest virtual machine under the compromised HostOS/Hypervisor. Privileges to manage VMCS and EPT are stripped from the compromised HostOS/Hypervisor into a separated and secure execution environment: HyperPS Space. The HyperPS Space shares the same processor privilege as the HostOS. HyperPS does not rely on any special hardware or a higher privilege. Updates to VMCS and EPT are abandoned by the HyperPS, if the operations are not authorized or are adjudged as harmful to virtual machine.

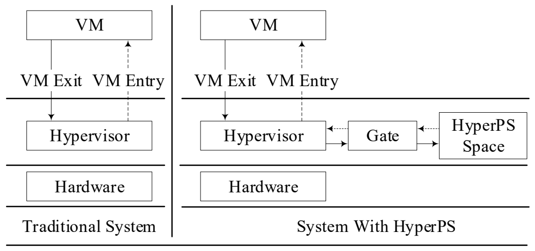

Fig. 4: Interaction Difference between the traditional system and system with HyperPS
VM-Exit handling only involves the Hypervisor in traditional system. In the system with HyperPS, VM-Exit is redirect into the HyperPS Space through the Gate. Context Management during VM-Exit and VM-Entry is managed by the HyperPS.

EPT with a VM identifier instead of each VMCS. HyperPS also takes `VMFUNC` into consideration too. EPTP switching is VM function 0. This VM function allows VM in VMX non-root operation to load a new value for the EPTP. The EPTP switching operation does not incur VM-Exit to VMX root operation. The `VMFUNC` is limited to selecting from a list of potential EPTP values configured in advance by the Hypervisor in VMX root operation. Thus, HyperPS also hooks all functions that manage the EPTP list in VMX root operation.

When the Hypervisor is going to load a new value for the EPTP or change the EPTP list in VMX root operation, functions are hooked and control flow is redirected to HyperPS Space. HyperPS iterates the VM-Mark table to check if the EPTP is authorized in advance.

TABLE 1: VM-Mark Structure

| Label | VMID | EPTID | EPT_Address |
|---|---|---|---|
| Description | The VM Identifier | The EPT Identifier | The Entry Address of this EPT |

### 4.3.2 EPT Paging-structure Protection

Values that update to EPT Paging-structures need to be verified too. In this paper, we present Page-Mark structure to record the relationships between EPT Paging-structures and the physical memory page frames. In HyperPS Space, HyperPS composes all Page-Mark structures into one table, called Page-Mark Table. HyperPS guarantees effective isolation between different virtual machines based on this Page-Mark Table.

Table 2 presents the basic architecture of the Page-Mark structure. The field `PFN` is the first address of the physical memory page frame. HyperPS fills in this field when a physical memory page frame is assigned to a VM. Usually, the Hypervisor will create a new EPT Paging-structure that maps this page frame and assigns the physical memory page frame to a guest physical memory page frame. The field `OwnerID` is the same as the value of `VMID` that identifies different VM/EPT. At the time when a physical memory page frame is assigned to a VM, HyperPS fills the field `OwnerID` with the value of the corresponding `VMID`. Kernel Same-page Merging (KSM), used by the KVM, allow for a greater guest density of identical or similar guest operating system by avoiding memory duplication. For example, different VMs with the same kernel may share the same physical memory page frames that hold the kernel code. HyperPS

TABLE 2: Page-Mark Structure

| Label | OwnerID | SharedID | PFN |
|---|---|---|---|
| Description | The Owner Identifier | The Shared Identifier | The Address of the Physical Memory Page Frame |

takes this situation into consideration too. If one physical memory page frame is shared between VMs, HyperPS fills the field `SharedID` with the one of the VM's `OwnerID`.

HyperPS has hooked all functions about EPT operations. At runtime, control flow will be redirected to the HyperPS Space when the Hypervisor invokes these functions. In the HyperPS, HyperPS index the Page-Mark structure tables with the physical memory page frame address. HyperPS ensures that any update to EPT Paging-structures must be in line with records in the Page-Mark Table. With the help of the Page-Mark Table, the attacker can no longer map a physical memory page frame that initially belongs to a victim VM to the malicious VM. The compromised Hypervisor can not map the guest VM into a dedicated physical memory page frame anymore.

With the help of VM-Mark Table and Page-Mark Table, HyperPS can effectively protect the isolation between VMs and the correct virtual-physical mapping relationships in a VM.

## 5 Implementation

### 5.1 HyperPS Space

In this paper, we create a delicate kernel-level trusted execution environment, called HyperPS Space, to inherit privileges of managing physical memory that originally belonged to the compromised hypervisor. The HyperPS Space achieves isolation and secure context switching without relying on a higher privileged layer or any special hardware.

#### *5.1.1 The Creation of HyperPS Space*

In our prototype, we create the HyperPS Space by using two sets of page tables. Figure 5 describes the address layout of HyperPS.

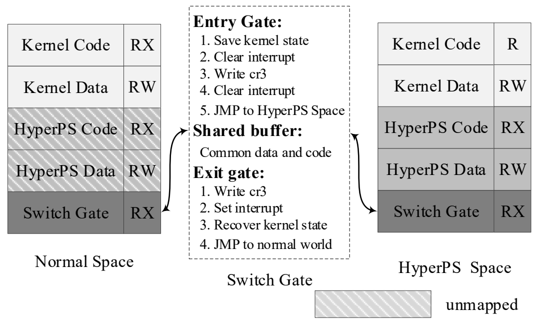

Fig. 5: Address layout of HyperPS

As shown in this figure, at the initialization stage, HyperPS has to load itself into memory firstly before it got executed. At this stage, all resources are managed by the Hostos kernel, HyperPS needs to request enormous numbers of memory allocation from the Hostos. These memories are used to hold HyperPS data and security tools that inherit compromised hypervisor's privileges of managing physical memory. In our prototype, we do not guarantee that all allocated memories are continuous in physical memory. Secondly, HyperPS creates a new page table that has already excluded the HyperPS Space. As shown in Figure 5, this dedicated page table is placed in the Normal Space and accessed by the compromised HostOS kernel. Because all page table entries that are relative to HyperPS Space have been removed from that dedicated page table, memory regions used by HyperPS are carved out of the memory ranges accessible to the HostOS kernel. Thirdly, HyperPS returns execution back to the Normal Space by loading the address of the dedicated page table into the control register CR3. Forthly, as depicted in Figure 5, we remove the execution bit in all kernel code (especially loadable kernel module code) entries of the page table used by HyperPS. HyperPS does not allow any execution of kernel code in the HyperPS Space in case attacker exploit kernel vulnerabilities in the HyperPS Space. At last, we erase the page table entry that records the base address of the HyperPS page table.

On the execution of the three steps mentioned above, HyperPS has established an execution environment. However, at this time, this execution environment does not have the conditions for isolation and security, since the kernel can still modify this memory layout or access permission.

#### *5.1.2 The Isolation of HyperPS Space*

HyperPS has to guarantee that the memory regions used by HyperPS are carved out of the memory ranges accessible to the kernel. In order to achieve the isolation of the HyperPS Space, HyperPS restricts the compromised HostOS kernel access to this space. Firstly, we removed write permissions to whole kernel page tables, so that the HostOS kernel can not modify the dedicated kernel page table in the Normal Space. In our prototype, the HostOS kernel can still navigate and examine page table entries, while write operations through legitimate page table management functions and macros are hooked and redirected into HyperPS Space. Meanwhile, direct write operations to the dedicated kernel page table will trigger an error (segment fault). Secondly, as mentioned above, HyperPS removed all page table management functions and macros and replaced them with hooks that jump to HyperPS Space, so that any modification to the dedicated kernel page table in the Normal Space will be redirected to HyperPS Space. For example, `native_set_pte()` is the function that actually implements PTE operations. Other PTE operation functions such as `set_pte()` and `native_set_pte_atomic()` all invoke the `native_set_pte()` function to accomplish their implementation. In our prototype, HyperPS hooks all these final and elemental page table management functions and macros. HyperPS makes sure that the HostOS kernel can neither tamper the virtual memory permissions of the HyperPS Space, nor can it establish new mapping relationships to the physical memory frames related to the HyperPS Space. In other word, the HostOS kernel page table is exclusively writable to HyperPS only. Lastly, HyperPS deprives the HostOS kernel of executing certain Control-Register-Relative functions so that it cannot direct the CPU to use alternative page tables other than the dedicated one we put in the Normal Space. For example, `native_write_cr3()` is written with inline assembly language to load the page table's physical address into the register CR3. HyperPS instrumented this function to prevent the adversary from using an unverified page table. HyperPS also intercept the accessing operation to Control Register: CR0 to prevent the adversary from disabling the WP protection mechanism. Actually, HyperPS hooked `native_write_cr0()`, `native_write_cr2()`, `native_write_cr3()`, and `native_write_cr4()` to HyperPS Space. By enforcing these three steps, the HostOS kernel running in the Normal Space can neither modify the dedicated host kernel page table nor change the control register configurations to use unverified page tables. As a result, the HostOS kernel cannot violate the isolation provided to HyperPS, since HyperPS deprives the HostOS kernel of privileges of accessing control registers and page tables and retains these privileges in the HyperPS Space.

## 5.2 Secure Switching

Switch Gate is designed to allow secure switching between HyperPS Space and the Normal Space, and it is the only interface between the Normal Space and the HyperPS Space. In this paper, HyperPS adopts a technique similar to the one presented in SKEE [3]. Our Switch Gate can achieve atomic, deterministic and exclusive too. As shown in the middle block of Figure 5, the Switch Gate mainly consists of three parts: the Entry Gate, the Exit Gate and the Shared Buffer. The Entry Gate provides the only entrance for HyperPS Space. The Entry Gate will save the HostOS kernel state into the Shared Buffer first. Secondly, it will clear and disable interrupts. Interrupts could enable the compromised HostOS kernel to crash the atomic execution of Switch Gate. Then HyperPS will enter the HyperPS Space by loading the page table used by HyperPS Space into register Cr3. The address of this page table is visible to HostOS kernel, however, as mentioned above, because all page table management functions and control register management functions have been hooked by HyperPS, the compromised HostOS kernel cannot subvert the HyperPS Space by using unverified page tables. Lastly, we clear interrupt with `CLI` instruction again before we execute any instruction in the HyperPS Space. The Exit Gate is the only interface that can divert execution back to Normal Space. It can only be invoked by HyperPS from the HyperPS Space. On receiving the invocation to Exit Gate, HyperPS will load the dedicated HostOS kernel page table into register CR3, recover interrupts and the saved kernel state. The Shared Buffer contains data and code shared by the Normal Space and the HyperPS Space. For example, as mentioned above, the entrance address of HyperPS Space and the return address to the Normal Space are stored in this buffer. HyperPS also stores the saved kernel state in this buffer.

However, TLB becomes a subtle issue in the implementation of our Switch Gate. TLB is a special cache used to keep track of recently used PTE transactions. Given a virtual address, the processor wll first examine the TLB if a PTE is present (TLB hit). If a match is found in the TLB, the processor retrieves the corresponding physical memory frame number from TLB directly. If the mapping is not cached by the TLB, the processor retrieves the physical memory page number from page table residing in the main memory and updates the new PTE into TLB. Thus, we need to flush TLB after we update both dedicated HostOS kernel page table and the page table used by HyperPS. In our prototype, we borrowed the idea in SecPod that clears the global bits in both the dedicated HostOS kernel page table and the HyperPS Space page table. By doing so, the TLB will always contain fresh address mappings after invoking the function `flush_tlb_all()`.

## 5.3 Privilege Deprivation and Management

In HyperPS, the HostOS has been deprived of the privileges of managing physical memory. This gives HyperPS complete control over the guest VM's virtual-physical memory mapping and protection. In our prototype, we hook all KVM's VMX and EPT functions to redirect EPT management to our HyperPS Space.

### *5.3.1 VMCS Hook*

Based on Intel manuals, VMCS can only be manipulated by several privileged instructions: `VMPTRLD`, `VMCLEAR` `VMPTRLD`, `VMREAD`, `VMWRITE`, `VMLAUNCH`, and so on. In the QEMU-KVM architecture, the KVM provides a wrapper around these privileged instructions. The QEMU does not need to deal architecture specific details, it just needs to invoke the wrapper functions in KVM with proper parameters. In Linux, all VMX operation functions are defined in the file `vmx_ops.h`. For example, `vmcs_writel()` wraps the privileged instruction `VMWRITE`, it will update the specified field of a VMCS with a given value. `__vmcs_readl()` is one of wrapper functions to the privileged instruction `VMREAD`, this function reads a field from a VMCS and returns field value. In our prototype, we implement all VMX functions in Normal Space with hooks to HyperPS Space to provide all VMX services. In particular, we prevent the HostOS/Hypervisor from loading an unverified EPTP. HyperPS interposes the execution of `vmcs_writel()`, and the Context Management in the HyperPS Space Management component checks the invoked parameters and verifies if the write to `VM-Execution Control Fields.EPTP` is legal or not. HyperPS also forbids arbitrary write to the `VM-Function Controls` field in `VM-Execution Control Fields`. The field `VM-Function Controls` contains the physical address of the 4-KByte EPTP list. Invalid value in the EPTP list may lead to an unverified EPTP value.

### *5.3.2 EPT Hook*

In this paper, because EPTs have been removed from the Normal Space, HyperPS needs to hooks EPT functions that access EPTP and EPT Paging Structures to HyperPS Space, and implement these services in the HyperPS Space. In QEMU-KVM architecture, the KVM is the component that maintains EPTs, while QEMU is responsible for requesting memory allocation. Thus, in our prototype, we dig into KVM to find out all functions that process EPTPs and EPTs directly. Figure 6 depicts how HyperPS handles all EPT management functions.

**EPT Creation**: The KVM will create and initialize MMU along with the creation of vcpu. If the EPT is supported and

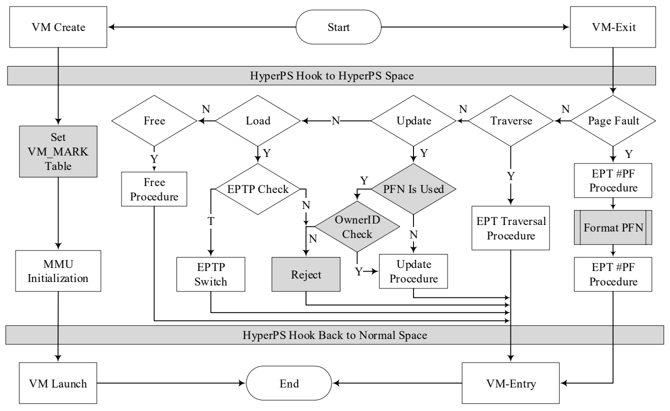

Fig. 6: HyperPS EPT Procedure

enabled by the hardware, The KVM module will invoke the function `init_kvm_tdp_mmu` to initialize the MMU. Actually, the function `init_kvm_tdp_mmu()` is just responsible for registering all kinds of EPT management functions to the inputted vcpu management structure: `vcpu->arch.mmu`. For example, the EPT page fault function is initialized to `kvm_tdp_page_fault()`, meanwhile inject page fault function is initialized to `kvm_inject_page_fault()`. In our prototype, we hooked all these registered functions into HyperPS Space. However, it is far from sufficient to hook these functions only. We need to dig into the internal of these functions to find out all surplus functions that access VMCS or EPT directly.

Besides, in this paper, we implement a new structure VM-Mark Table to guarantee the legitimate relationship between the VM and the EPT. During the creation of EPT, the function `vmx_load_mmu_pgd()` is responsible for writing the newly created EPTP to the corresponding VMCS. In our prototype, as shown in Figure 6, we hooked this function into the HyperPS Space. In the HyperPS Space, we also implemented our protection that stores and synchronizes the corresponding EPTP value and VM identifier to VM-Mark Table.

**EPT Page fault**: The KVM adopts a similar page table construction process to the kernel page table that fills EPT paging structures by handling EPT page fault. As mentioned above, if the EPT page fault occurs, the KVM invokes the function `kvm_tdp_page_fault()` to handle this exception. Typically, the KVM is not responsible for the allocation of physical pages. It is responsible for synchronizing the allocated physical page frame number (PFN) to the corresponding EPT Paging-structure. We classify EPT operations in EPT Page Fault into Update. In a virtualization environment, there is a memory reclamation technique, called Ballooning, that is used by the hypervisor to allow the physical host system to retrieve memories from certain guest virtual machine and share them with others. However, Ballooning may be abused to facilitate side channel attacks. In this paper, we take such kind of situation into consideration. As depicted in Figure reffig:impept, HyperPS will format the allocated physical memory frame before mapping it in the EPT while executing the function `kvm_tdp_page_fault()`. More details are illustrated in Section 5.4.

**EPT Traverse**: Because all EPTs have been moved from the Normal Space to the HyperPS Space, though HyperPS does not interpose the traversal of EPT, HyperPS also needs to instrument EPT traversal functions, so that they can access EPTs. However, we do not need to re-implement EPT traversal services in the HyperPS Space. In our prototype, we just implement a hook function that will invoke the Switch Gate to retrieve the corresponding EPTP value or EPT Paging Structure value. For example, the function `shadow_walk_init()` is responsible for initializing the struct `kvm_shadow_walk_iterator` structure which will record the value of EPTP and be used in the traversal of EPTs. We added the hook function inside the function `shadow_walk_init()` to access EPT in HyperPS Space.

**EPT Load**: In this paper, we propose a VM-Mark table to tag the mapping relations between the EPT and the corresponding virtual machine. HyperPS will be engaged to check whether the loaded EPT corresponds to the virtual machine every time the virtual is going to executing VM-Entry. `vcpu_enter_guest()` is an arch-specific function in KVM to start up a virtual machine. As depicted in Figure 6, we placed a hook before executing the function `vmx_load_mmu_pgd()` inside this function. The function `vmx_load_mmu_pgd()` is the final function that load the EPT. If the EPTP (the base address of the EPT) matches an item in the VM-Mark Table, HyperPS will allow the subsequent operations, such as EPTP Switch and EPT Load. Otherwise, HyperPS will terminate the execution of the corresponding virtual machine.

**EPT Update**: In this paper, we propose a new structure: Page-Mark Table to record the relationships between EPT Paging-structures and the physical memory page frames. HyperPS guarantees that physical page frames with write permission can only be accessed with one determinate VM. Physical page frames shared by different machines can only be read-only. Besides, HyperPS guarantees that virtual machines participating the memory sharing are authenticated and recorded by us. Figure 6 depicts the rough process of how HyperPS handles EPT Update. As mentioned above, in this paper, in addition to changing the permissions of EPT entries, HyperPS also regards the allocation of a new EPT Paging-structure as EPT Update. In our prototype, we do not get involved in new physical memory page frame allocation. We focus on operations to EPTs only. The function `__direct_map()` in KVM is the core function to build new EPT entries. HyperPS places hooks inside this function to perfrom security checks. If the allocated physical frame number is untracked in the Page-Mark Table, HyperPS will first invokes the function mentioned above to format this allocated physical page. Then, HyperPS hooks the function `kvm_mmu_get_page()`, the function `link_shadow_page()`, and the function `mmu_set_spte()` into the HyperPS Space. In the HyperPS Space, HyperPS will initialize a new EPT Paging-structure, and link it to the EPT.

## 5.4 KSM Handle

Kernel Same-page Merging (KSM), used by the KVM Hypervisor, allows KVM guests to share identical memory pages. KSM enables the KVM Hypervisor to examine two or more already running virtual machines and compare their memory. If any memory region or pages are identical, KSM reduces multiple identical memory pages to a single page. The KSM adopts two red-black trees: the unstable tree and the stable tree, to implement its service. The unstable tree holds pointers to pages that have been found to be unchanged for a period of time. The stable tree holds pointers to all the merged pages (KSM pages), sorted by their contents. All pages in the stable tree are write-protected. If the contents of the page is modified by a guest virtual machine, a new page is created for that guest. Once a merged page has been recorded into the stable tree, the merged page pointer will never be removed from the stable tree until all users have either modified or unmapped it. At runtime, for each page scanned, the KSM proceeds to search a match first in the stable tree that only contains already shared pages. If a match is found in the stable tree, KSM will merge this scanned page with the KSM page found in the stable. If no match is found in the stable tree, KSM will then search the unstable tree. If a match is found in the unstable tree, the page is merged with the page in the unstable tree, and the resulting KSM merged page is added to the stable tree.

In this paper, as mentioned in Section 4.3.2, we also take KSM into consideration. `stable_tree_insert()`, `stable_tree_append()` are the functions in KSM to add the merged page into stable tree. Firstly, As mentioned above, we placed hooks in these stable tree operation functions to trap execution to HyperPS Space. In our prototype, the Page-Mark Table is used to record all physical page frames used by the VM. If a page is merged by KSM, HyperPS fills the `SharedID` with the corresponding VM identifier. Secondly, Merging page results in EPT update too. HyperPS needs to get involved in the page merging process too. `try_to_merge_one_page()` and `try_to_merge_two_page()` are the function in KSM to merge two pages into one. For EPTs have been moved to HyperPS Space and EPT management functions are hooked to HyperPS Space too, we also hook these functions to HyprePS Space to synchronize the memory mapping changes to EPT.

## 6 Evaluation

In this section, we first analyze the security guarantees provided by HyperPS. Then, we evaluate the performance overhead by running a set of benchmarks on both standard KVM and HyperPS.

### 6.1 Security Analysis

We first evaluate the security of HyperPS by analyzing how HyperPS can resist various attacks. We organize these attacks from two perspectives: virtual-machine-related data structures corruption, HyperPS Space violation.

#### *6.1.1 Virtual-machine-related Data Structure Corruption*

HyperPS has deprivileged the HostOS/Hypervisor and interposed all interactions between HostOS and the physical memory. As illustrated above, VMCS and EPT are the two ultimate structures that adversaries want to tamper with.

TABLE 3: Hypervisor Attacks Against HyperPS.

| *Attack* | *Description* |
|---|---|
| VMCS Attack | Load a crafted GUEST_CR3 value |
| EPT Attack (CVE-2017-8106) | Load a crafted EPT value |
| DMA Attack | Access HyperPS Space by DMA |
| Code Injection Attack | Inject code and cover hooked functions to bypass HyperPS Space |

The adversary can maliciously modify the fields in these two structures with regular memory access, if he gained the location of them in HostOS in advance. However, in our prototype, HyperPS has removed these two data structures from the HostOS/Hypervisor and placed them in the HyperPS Space. The adversary can not modify these two data structures with regular memory access.

As shown in Table 3, we constructed a typical attack (named VMCS Attack) that attempts to modify the field `Guest_CR3` in VMCS with regular memory access. As we expected, Because the VMCS has been removed from the Normal Space, this attack failed with information: Segment Fault.

EPTs are the most critical data structures that manage the VM's physical memory. As illustrated in Section 3.2.2, we focus on resisting the Double-Mapping Attack and Remapping Attack if the attacker gets control over EPTs. A successful Double-Mapping attack will assign the memory pages that have already been owned by a hostile VM to the victim VM. The Remapping attack also manipulates memory mapping relationships. As illustrated in Section 3.2.2, a successful Remapping attack malicious modifies the victim VM's EPT Paging-Structures to another dedicated page frame. In HyperPS Space, because of the Page-Mark Tables and write-protection of EPTs, HyperPS can easily resist these attacks. In our prototype, all page table operation functions in the HostOS have been hooked into HyperPS Space. At runtime, all page table modification operations will be checked under the Page-Mark Table. Illegal page table modification or illegal memory share between VMs will be abandoned by HyperPS. We implemented a real-world attack to examine if HyperPS can resist these attacks. The `CVE-2017-8106` listed in Table 3 allows a privileged KVM guest user to access EPT. This attack can also conduct attack by invoking a single-context `INVEPT` instruction with a `NULL` EPTP. HyperPS succeed in detecting the execution of this attack, and HyperPS terminated the malicious VM's execution. Because HyperPS has hidden the address of EPT in the HyperPS Space, malicious access to EPT incurred EPT access fault. HyperPS has hooked all VMCS operation functions. Any change to EPTP and EPTP List is supervised by HyperPS. Attackers can not change the EPTP with a malicious value.

#### *6.1.2 HyperPS Space Violation*

HyperPS Space is a delicate kernel-level secure and isolated execution space that inherits HostOS's privileges of managing physical memory. In this paper, we implement HyperPS Space by using two sets of HostOS page tables and page-table-operation hooking.

The first kind of attack attempts to use regular memory access to maliciously modify the page tables, including the dedicated HostOS kernel page table and the HyperPS Space page table. For example, if HyperPS exposes the HyperPS page table base address, then a compromised kernel can compromise the isolation by changing page table entries. Thus security tools in the HyperPS Space will be tampered with. In our prototype, as depicted in Figure 5, HyperPS removed the corresponding page table entries that are relative to HyperPS page table base address, code and data. The compromised HostOS can not accurately find this base address from the entire address space. Direct access to the original addresses that are allocated to HyperPS in the Normal Space will fault because of Segment Fault. Besides, HyperPS also set the kernel code non-executable when the processor executes instructions in the HyprePS Space. This can prevent the attacker from exploiting kernel vulnerabilities in the HyperPS Space.

The attacker can also abuse control registers to crash HyperPS Space. For example, the register `CR0` controls the W$\oplus$X privilege of code, the register `CR4` controls if the SMEP mechanism is enabled or not. If an attacker takes over these control registers, he can crash HyperPS Space easily. In our prototype, HyperPS has hooked all control-register-operation functions into HyperPS Space. Values that are written to control registers are supervised by HyperPS. HyperPS will reject any malicious modification.

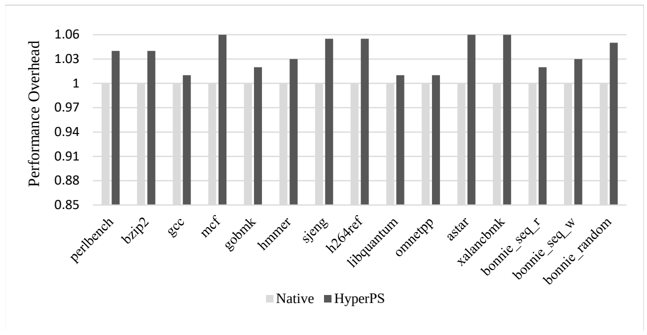

Fig. 7: SPECCPU 2006 and Bonnie++ Performance Measurement Results

## 6.2 Performance Evaluation

In this section, we evaluate the performance of our HyperPS prototype. All the experiments were conducted on a physical server with a 2.0 GHz 64 core Intel processor and 32 GB memory. The Host system runs Ubuntu 16.04 LTS with a kernel version of 4.4.1. We created more than five virtual machines. The guest is configured with 1GB of memory and runs Ubuntu 16.04 LTS Service with a kernel version of 4.4.1.

In our prototype, the HostOS kernel is modified so that HyperPS Space is initialized during the boot up sequence. This includes creating a new memory page table for HyperPS, allocating memory pages, as well as creating Page-Mark table and VM-Mark table. This process introduces security verification for pages according to Page-Mark Table, and security access to VMCS in HyperPS Space during VM Exit/Entry sequence. The kernel is modified to place hooks upon some functions.

To evaluate the performance of HyperPS, we experimented with SPECCPU 2006, Bonnie++ and HyperBench.

TABLE 4: execution time of vm operation(s).

| *Test Case* | *VM Create* | *VM Destroy* |
|---|---|---|
| No_HyperPS | 11.79 s | 1.75 s |
| With_HyperPS | 12.97 s | 1.89 s |
| Efficiency | 1.1 | 1.08 |

The SPECCPU 2006 measure HyperPS's impact to compute-intensive workloads, such as gcc; The Bonnie++ quantifies the performance overhead introduced by HyperPS for file systems; and the HyperBench measure the overall cloud performance under HyperPS, especially impact to the KVM Hypervisor. All the experiments were repeated fifty times in the cloud environment with HyperPS and in the original cloud environment (the baseline). In these fifty experiments, we set up different workloads (different numbers of guest VMs with different memory sizes) to simulate different environments in the real cloud environment. Besides, we also measured the VM load time to evaluate the impact on VM initialization. The average results of all the experiments are reported.

**Benchmarks Performance** We firstly experiment with SPECCPU 2006 to measure the performance overhead introduced by HyperPS. SPECCPU 2006 consists of several compute-intensive benchmarks, stressing the system's processor and memory subsystems. The experiment results are list in Figure 7. As shown in this figure, for the majority of these applications, such as perlbench, bzip2, gcc, gobmk, libquantum and ometpp, HyperPS introduced at most 3% performance overhead. However, there are still several benchmarks, such as mcf, astar and xalancbmk whose results are higher than average. They caused a performance loss of 6%. Most of these workloads have large and fluctuating memory footprints. It's not surprising as the HyperPS takes over physical memory management. Frequent memory request incurs HyperPS involvement to index the Page-Mark Table and verifies the legality of EPT paging-structure updates. We can conclude that, for most compute-intensive applications without large memory footprints, HyperPS has a similar performance as the baseline cloud environment. For applications with large memory footprints, We believe that this performance impact is entirely acceptable for trading this performance loss for higher security.

We then experiment with Bonnie++ to measure the performance overhead introduced by HyperPS to file systems. Bonnie++ sequentially reads/writes data from/to a particular file in different ways. The read/write granularity varies from a character to a block. Furthermore, we also test the time cost of the random access. In our experiments, the target file on which Bonnie++ performs a series of test operations is 1000MB in size. Figure 7 shows the Bonnie++ measurement results which show the performance overhead on the file system is acceptable too. The performance loss of sequential read, write and random access is 2%, 3% and 5%. Actually, HyperPS does not interpose much on memory operations for I/O data.

HyperBench is a benchmark suite that focuses on measuring cloud performance, especially on measuring HostOS/Hypervisor's capabilities. The HyperBench designs 15 hypervisor benchmarks covering CPU, memory, and I/O. The operation in each benchmark triggers hypervisor-level events, which examines the platform's ability in the target area. We used HyperBench to evaluate the performance loss introduced by HyperPS to the HostOS/Hypervisor. The results are shown in Figure 8. For half of the test cases, HyperPS does not introduce performance overhead or little performance loss which is less than 1 percent. Inter-Processor Interrupt (IPI) is a special type of interrupt by which one processor may interrupt another in a multiprocessor system. HyperPS introduces 4% performance overhead in this test suit. The virtual IPI delivery requires two VM exits and needs to write to VMCS. Because the VMCS has been removed in the Normal Space, access to VMCS requires the involvement of HyperPS. HyperPS also poses a higher impact on the cold-access test suit (about 5 percent). As mentioned above, the KVM constructs EPT with page faults. Data accessed in the cold-access benchmark is not loaded into physical memory in advance. Thus, the EPT does not map these data before. In our prototype, HyperPS interposes EPT paging-structure initialization and EPT Page Fault handle procedure. All EPT Updates are completed in the HyperPS Space. This is the reason why HyperPS introduced performance loss in this test suit. As shown the Figure 8, compared with the cold-access benchmark, the performance overhead introduced by HyperPS in the hot-access benchmark is obviously smaller. set-pt is another benchmark that operates on EPT. HyperPS introduced about 4 percent more performance overhead than the baseline because all EPT operations are hooked into the HyperPS Space.

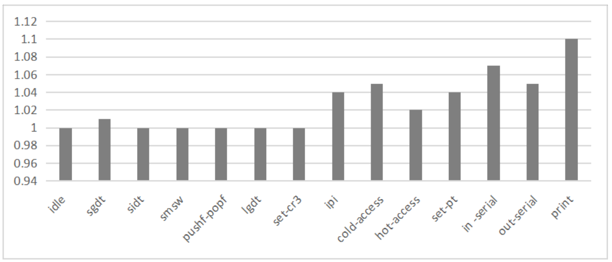

Fig. 8: Virtualization Performance Measurement Results using HyperBench

**VM Load Time** The load time of a VM is a critical aspect of performance to be considered because it influences user experience. We design experiments to evaluate the performance impact of HyperPS for VM loading. We measure the impact of completely booting and shutdowning a VM (configured with 2 VCPU and 512MB memory). As Table 4 shown, the booting time is suffered a 1.1 times slowdown under HyperPS, shutdown time is suffered a 1.08 times slowdown, due to the extra overhead of worlds switching and Page-Mark table accessing in HyperPS Space. Such overhead is worth for HyperPS.

## 7 Related work

We describe the related work from these three aspects, reconstructed hypervisor, customized hardware, and the same privilege-level isolation.

### 7.1 Customized Hardware

Some works at hardware level complete the protection of the process by extending the virtualization capabilities or measure the integrity of the Hypervisor [21] [16] [1] [24] [30] [38] [2] [29]. These tasks provide fine-grained isolation of processes and modules from the hardware level. Haven [5] uses Intel SGX [13], [19] to isolate cloud services from other services and prevent cross-domain access. SGX provides fine-grained protection at the application space instead of hypervisor space, and needs developers spend time reconstructing code and dividing code into trusted part or untrusted part. SGX has requirement for version of CPU and is applied on a few platforms. Datasafe [**?**] provides dynammic instantiations of secure data compartments and continuously tracks and propagates hardware tags to identify sentive data by enforcing unbypassable output control. HyperCoffer [36] is a hardware-software framework to protect the integrity of a guest VM that only trusts the processor chip. HyperCoffer introduced a new mechanism called VM-Shim, which runs in-between a guest VM and the hypervisor. Some solutions provide protections by using additional hardwares, such as PCI device, TPM, etc. HyperCheck [30] [38] leverages SMM and a PCI device to securely generate and transmit the full state of the protected machines to an external server, so that it can provide protection to the integrity of hypervisor.

### 7.2 Reconstructed Hypervisor

Except for approaches based on hardware, some works ( [26] [26] [33]) pay attention to software isolation. Pre-allocating physical resource and completed isolated environment for every VM can avoid VM cross-domain attack, and data leakage attack. NOVA [26] divides hypervisor into micro-hypervisor and user hypervisor running in root mode, adopts an idea which is similar to fault domain isolation to guarantee an isolated user hypervisor for every VM. The drawback of this approach is the lack of fractional traditional hypervisor functions. HyperLock [33] prepares backup KVM for every VM by copying KVM code, and ensures every VM run in own isolated space. Nexen [26] reconstructs the XEN hypervisor into one privileged security monitor, one component for shared service, backup XEN code and data for every VM, to resist attacker from exploiting known XEN vulnerabilities. Some solutions [7], [28] propose to insert another layer below the hypervisor to measure and guarantee the integrity of the hypervisor. GuardHype [8] mediates the access of third-party hypervisors to the hardware virtualization extensions, effectively acting as a hypervisor for hypervisors. CloudAuditor [32], Nosv [23], CloudViosr-D [20], and the Turtles Project [6], reconstruct the Hypervisor by introducing a higher privilege level than the origin hypervisor.

These approaches redesign hypervisor greatly. In contrary, HyperPS adopts a feasible way to isolate VM without lots of modification of hypervisor.

### 7.3 The Same Privilege Level Isolation

Some efforts, ED-Monitor [10], SKEE [3] and SecPod [31], [10], adopt the same privilege-level idea to avoid performance overhead of inter-level translation. ED-Monitor presents a novel approach that enables practical event-driven monitoring for compromised hypervisor in cloud computing, adopts "the same privilege level" protection against an untrusted hypervisor. The created monitor is placed at the same privilege level and in the same space with hypervisor. It relies on the mutual-protection of a unique pair of the techniques: Instrumentation-based Privilege Restriction (IPR) and Address Space Randomization (ASR). At the high level, IPR intercepts the most privileged operations in the Hypervisor and transfers these operations to ED-monitor, while ASR hides ED-monitor in the address space from the Hypervisor. SKEE can only be applied to limited levels of system software in comparison with HyperPS. First, targeting ARM's 32-bit architecture, SKEE capitalizes mainly on Translation Table Base Control Register (TTBCR) for dynamic page table activation. To be more specific, SKEE creates separate page tables for the secure world and activates it in a timely manner by modifying the N field of TTBCR. However, as this hardware feature is only defined in the kernel privilege level on AArch32, SKEE is not commonly applicable to different levels of system software, such as hypervisors. SecPod, an extensible approach for virtualization-based security systems that can provide both strong isolation and the compatibility with modern hardware. The biggest difference between SecPod and HyperPS is that SecPod creates the secure isolation environment for every VM. SecPod solves the problem of VM address mapping with the assistance of shadow page table (SPT) technology.

We neither adopt software at a higher level than the hypervisor, nor use customized hardware. Inspired by the same privilege-level, we propose HyperPS Space placed at the same privilege-level with hypervisor. HyperPS is independent on multi-platforms and practical for cloud providers.

## 8 Conclusion

We introduce HyperPS, an approach that enables x86 platforms to support a secure isolated execution environment at the same privilege-level with hypervisor. The environment is designed to provide memory isolation protection for VMs under the condition that the HostOS/hypervisor has already been compromised. This approach, which does not rely on additional hardware devices or a higher privilege level software. HyperPS also does not rely on special types of processor architecture or special kernel version. HyperPS reflects good practicality, portability, and independence on multi-platforms. Besides, evaluation shows that HyperPS does not introduce unacceptable negligible performance overhead. We support that HyperPS can be implemented widely in real-world for cloud providers.

## References

[1] Ahmed M Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C Skalsky. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 38–49, 2010.

[2] Ahmed M Azab, Peng Ning, and Xiaolan Zhang. Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 375–388, 2011.

[3] Ahmed M Azab, Kirk Swidowski, Rohan Bhutkar, Jia Ma, Wenbo Shen, Ruowen Wang, and Peng Ning. Skee: A lightweight secure kernel-level execution environment for arm. In *NDSS*, volume 16, pages 21–24, 2016.

[4] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44, 2009.

[5] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *Acm Transactions on Computer Systems*, 33(3):1–26, 2014.

[6] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The turtles project: Design and implementation of nested virtualization. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, Vancouver, BC, October 2010. USENIX Association.

[7] Muli Ben-Yehuda, Michael D Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The turtles project: Design and implementation of nested virtualization. In *Osdi*, volume 10, pages 423–436, 2010.

[8] Martim Carbone, Diego Zamboni, and Wenke Lee. Taming virtualization. *IEEE Security & Privacy*, 6(1):65–67, 2008.

[9] Bandan Das, Yang Y Zhang, and Jan Kiszka. Nested virtualization: State of the art and future directions. *URl: http://www. linux-kvm. org/images/3/33/02x03-NestedVirtualization. pdf*, 2014.

[10] Liang Deng, Peng Liu, Jun Xu, Ping Chen, and Qingkai Zeng. Dancing with wolves: Towards practical event-driven vmm monitoring. *Acm Sigplan Notices*, 52(7):83–96, 2017.

[11] Dawson R Engler, M Frans Kaashoek, and James O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. *ACM SIGOPS Operating Systems Review*, 29(5):251–266, 1995.

[12] Shay Gueron. Intel advanced encryption standard (aes) instructions set. *Intel White Paper, Rev*, 3:1–94, 2010.

[13] Matthew Hoekstra, Reshma Lal, Carlos Rozas, Vinay Phegade, and Juan Del Cuvillo. Cuvillo, "using innovative instructions to create trustworthy software solutions," in hardware and architectural support for security and privacy. In *6 IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS.*, 2013.

[14] David Kaplan, Jeremy Powell, and Tom Woller. Amd memory encryption. *White paper*, 2016.

[15] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, 2009.

[16] Hojoon Lee, Hyungon Moon, Daehee Jang, Kihwan Kim, Jihoon Lee, Yunheung Paek, and Byung Hoon Kang. Ki-mon: a hardware-assisted event-triggered monitoring platform for mutable kernel object. In *Usenix Conference on Security*, pages 511–526, 2013.

[17] Jochen Liedtke. On micro-kernel construction. *ACM SIGOPS Operating Systems Review*, 29(5):237–250, 1995.

[18] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, pages 1–9. 2016.

[19] Frank Mckeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–1, 2013.

[20] Zeyu Mi, Dingji Li, Haibo Chen, Binyu Zang, and Haibing Guan. (mostly) exitless {VM} protection from untrusted hypervisor through disaggregated nested virtualization. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1695–1712, 2020.

[21] Hyungon Moon, Hojoon Lee, Jihoon Lee, Kihwan Kim, Yunheung Paek, and Brent Byunghoon Kang. Vigilare: toward snoop-based kernel integrity monitor. In *ACM Conference on Computer and Communications Security*, pages 28–37, 2012.

[22] Ramu Ramakesavan, Dan Zimmerman, and Pavithra Singaravelu. Intel memory protection extensions (intel mpx) enabling guide, 2015.
[23] Jianbao Ren, Yong Qi, Yuehua Dai, Yu Xuan, and Yi Shi. Nosv: A lightweight nested-virtualization vmm for hosting high performance computing on cloud. *Journal of Systems and Software*, 124:137–152, 2017.
[24] Joanna Rutkowska and Rafał Wojtczuk. Preventing and detecting xen hypervisor subversions. *Blackhat Briefings USA*, 2008.
[25] Jonathan S Shapiro, Jonathan M Smith, and David J Farber. Eros: a fast capability system. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 170–185, 1999.
[26] Lei Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. Deconstructing xen. In *NDSS*, 2017.
[27] Udo Steinberg and Bernhard Kauer. Nova: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems*, pages 209–222, 2010.
[28] Raoul Strackx and Frank Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 2–13, 2012.
[29] Jakub Szefer and Ruby B Lee. Architectural support for hypervisor-secure virtualization. *ACM SIGPLAN Notices*, 47(4):437–450, 2012.
[30] Jiang Wang, Angelos Stavrou, and Anup Ghosh. Hypercheck: a hardware-assisted integrity monitor. In *International Conference on Recent Advances in Intrusion Detection*, pages 158–177, 2010.
[31] Xiaoguang Wang, Yue Chen, Zhi Wang, Yong Qi, and Yajin Zhou. Secpod: a framework for virtualization-based security systems. In *Usenix Conference on Usenix Technical Conference*, pages 347–360, 2015.
[32] Zhe Wang, Jin Zeng, Tao Lv, Bin Shi, and B. Li. Cloudauditor: A cloud auditing framework based on nested virtualization. *2016 IEEE 3rd International Conference on Cyber Security and Cloud Computing (CSCloud)*, pages 50–53, 2016.
[33] Zhi Wang, Chiachih Wu, Michael Grace, and Xuxian Jiang. Isolating commodity hosted hypervisors with hyperlock. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 127–140, 2012.
[34] Orit Wasserman and Red Hat. Nested virtualization: shadow turtles. In *KVM forum*, 2013.
[35] Andrew Whitaker, Marianne Shaw, and Steven D Gribble. Scale and performance in the denali isolation kernel. *ACM SIGOPS Operating Systems Review*, 36(SI):195–209, 2002.
[36] Yubin Xia, Yutao Liu, and Haibo 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. IEEE, 2013.
[37] Ben-Ami Yassour, Muli Ben-Yehuda, and Orit Wasserman. Direct device assignment for untrusted fully-virtualized virtual machines. 2008.
[38] Fengwei Zhang, Jiang Wang, Kun Sun, and Angelos Stavrou. Hypercheck: A hardware-assistedintegrity monitor. *IEEE Transactions on Dependable and Secure Computing*, 11(4):332–344, 2013.
[39] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *ACM Symposium on Operating Systems Principles*, pages 203–216, 2011.