HyperMI: A Privilege-level Secure Execution Environment for VM Protection against Compromised Hypervisor

xxx

Institute of Information Engineering, Chinese Academy of Sciences
School of Cyber Security, University of Chinese Academy of Sciences
{xxx}@iie.ac.cn

Abstract—Sensitive data and critical content in guest virtual machine is easy to be leaked once hypervisor in the highest privilege-level is compromised. Therefore, VM protection is important to monitor running VM and protect the memory of VM from compromised hypervisors malicious access causally. Previous efforts employ extra customized hardware or employ new architecture relying on a higher privilege level.

This paper proposes HyperMI, a novel approach that provides a privilege-level secure execution environment for running VM protection in cloud computing against compromised hypervisor. It provides HyperMI world, effective VM isolation and event-driven VM monitoring in order to prevent customers' sensitive data in VM from being leaked. HyperMI world, placed at the same privilege level with hypervisor, is a privilege-level secure isolated execution environment as TCB, and the world is used for VM isolation and VM monitoring. What's the most important, HyperMI focuses on decoupling the function of interaction between hypervisor and VM and decoupling the function of address mapping of VM from hypervisor. As a result, HyperMI can only controls page mapping of VM and EPT updating when page is allocated to VM. We have implemented a prototype for KVM hypervisor with multiple Linux as guest OSes, which can be used in commercial cloud computing industry with portability and compatibility for all kinds of CPU platforms. The security analysis shows that this approach can provide protection for VM with effective isolation and event-driven monitoring, and the performance evaluation confirms the efficiency of HyperMI.

Index Terms—Virtualization, VM Isolation, VM Security

I. INTRODUCTION

As more and more functionalities are added into hypervisor, the code bases of commodity hypervisors (KVM or Xen) have been increased to be massive lines. However, recent survey shows that commodity hypervisor incurs more vulnerabilities because of the larger code bases. On the one hand, from 2004 to now, there are 130 vulnerabilities about KVM. Some of them (e.g., CVE-2018-1087) shows high-risk vulnerabilities that can lead to privilege raising behavior and comprehensive compromised hypervisor. On the other hand, because hypervisor possess the highest privilege in the cloud environment, an attacker who compromises hypervisor could harm the whole cloud infrastructure and endanger data and computation in the cloud. For example, an attacker can deploy a complete malicious guest VM on the virtualized platform, conducts attacks to the hypervisor and further attack other VMs even the entire platform through illegal data accessing and so on. Some attacks also directly compromise hypervisor. In order to settle down all these threats, some try to detect malicious actions among frequent cloud management operations, but this kind of approach is much similar to that of looking for a needle in a haystack. Therefore, VM isolation and VM monitoring could provide a superior solution from another perspective. Current previous researches, including customized hardware, reconstructed hypervisor and software placed at a higher privilege level, provide services of protection for critical data.

Customized Hardware Some efforts (SecureME [5], Bastion [4] and Iso-x [10]) rely on customized underlying hardware to provide fine-grained protection for VM or in-VM process. Iso-X provides isolation for security-critical pieces of an application by introducing additional hardware and changes to OS. It controls memory access by introducing ISA instructions. Bastion, uses modified microprocessor hardware based on FPGA to protect the storage and runtime memory state of enhanced hypervisor against both software and hardware attacks. So that it provides hardware-protected environment and protection for security-critical OS and application modules in an untrusted software stack.

Reconstructed Hypervisor Some efforts (NoHype [13] and TrustOSV [23]) pre-allocate fixed cores or memory resource to isolate VM via reconstructing hypervisor. In the meantime, deprive some virtualization capabilities and introduce lots of modification to hypervisor. And NoHype removes the virtualization layer, while retaining the key features enabled by virtualization. TrustOSV provides protection from compromised cloud environment by removing interaction between exposed executing environment and hypervisor.

Software at A Higher Privilege Level In order to mitigate the hazard caused by the hypervisor possess at the highest privilege level, plenty of software solutions propose and introduce a higher privilege-level than the original hypervisor. Nested virtualization is one of the representative approaches, who provides a higher-privileged and isolated execution environment to run the monitor securely. The turtles project [3]

and CloudVisor [26] are examples of systems that propose nested virtualization idea to achieve isolation for protected resources. Specially, CloudVisor uses nested virtualization to decouple resource management into nested hypervisor to provide protection for VM. These approaches based on a higher privilege level would introduce lots of inter privilege level transition, severe performance overhead and a larger code base.

Meanwhile, practicality and low performance overhead are among the most prized features for cloud providers. For business architectures that are already widely deployed, it is equally important to minimize changes to existing systems or architectures. Current works can't satisfy all features simultaneously. To avoid introducing extra hardware device and address the high performance overhead of inter-privilege transition for software at a higher privilege level, some recent efforts focus on software approaches about how to achieve same privilege level isolation and protection without relying on a higher privilege level. For example, SKEE [1] introduces a secure execution environment at the same privilege level with kernel.

In this paper, we propose HyperMI, a "same privilege level" and software-based VM isolation approach on x86 platform, to provide runtime protection for guest VMs against compromised hypervisor. HyperMI introduces a secure isolation execution environment, named HyperMI world, to place security tools including event-driven VM monitoring and VM isolation module. The VM isolation module guarantees memory isolation between VMs. The VM monitoring module provides monitoring for the interaction between compromised hypervisor and each VM. Details of these three modules are as follows.

HyperMI World Inspired by the idea of "same privilege level" isolation, HyperMI world is created at the same privilege level with hypervisor without introducing severe performance overhead. Also, no part of our approach runs at a higher privilege level than the original hypervisor.

VM Isolation Since compromised hypervisor with the highest privilege can access all memory of VMs casually, HyperMI isolates memory of VMs and hypervisor, and implements access control for physical memory pages of isolated VM to avoid malicious access. This can resist remapping and double mapping attack. Firstly, HyperMI marks each page with page marking technique to guarantee each page can only be owned by one VM or hypervisor. Secondly, it deprives the address translation function of hypervisor to ensure that the page is marked with owner when the page is mapped. Finally, it can avoid double mapping and remapping attack. In order to avoid double mapping attack, the owner of the page is verified to ensure whether access or not when EPT updates resulting from page fault. In order to resist remapping attack, clear the content of page when the physical page is released.

VM Monitoring Event-driven VM monitoring provides protection for interaction between hypervisor and VM to protect states domain of VM and hypervisor. First, VM monitoring is event-driven, however, other approaches provide VM monitoring by using polling-query. This way can't detect the malicious actions during polling-query intervals, when compared to even-driven monitoring. Second, there are some especially critical structures data recording states information of VM and hypervisor need to be monitored. Once these three data structures are attacked, the security of VM can't be guaranteed. These data includes Extended Page Tables (EPT), EPT Pointer (EPTP) and Virtual Machine Control Structure (VMCS). EPT contains address mapping relationship between the Guest Physical Address (GPA) and Host Physical Address (HPA). EPTP is a CR3-like register used to identify the location of the corresponding EPT. VMCS is used in VMX operation to manage transitions into and out of VMX non-root operation. Attackers attack these data to perform a variety of attacks. Therefore, HyperMI provides monitoring for them.

Our contributions are as follows:

- A secure isolated execution environment placed at the same privilege level with hypervisor instead of relying on a higher privilege-level or customized hardware.
- An approach of isolating memory among VMs and hypervisor securely for VM by using page marking technique to avoid malicious access from compromised hypervisor.
- A unbypassable hypervisor monitoring for VMCS and EPT approach which can ensure the security of interaction between hypervisor and VM.
- A prototype based on KVM and x86 architecture with trivial performance overhead, high security and portability.

Our prototype introduce 4K SLOC (Source Lines of Code) to VM monitoring and 300 SLOC modifications to the hypervisor, VM monitoring reduces attack surface of hypervisor, and the memory isolation among hypervisor and VMs is significantly guaranteed. The experimental results show trivial performance overhead for completed and secure memory isolation approach for VM and event-driven VM monitoring.

The rest of this paper is organized as follows. Section 2 details background. Section 3 discusses our threat model. Section 4 gives an overview of the HyperMI architecture while Section 5 presents our implementation. Section 6 gives the evaluation of security and performance. Section 7 compares HyperMI to past work and Section 8 concludes.

II. BACKGROUND

A. Address translation of VM

The address translation of VM requests two page tables, guest page table and extend page table (EPT). Guest page table can finish the translation from guest virtual address (GVA) to guest physical address (GPA). EPT technique, based on hardware and managed by hypervisor, is used to support for memory virtualization at processor level and improve virtualization performance. In the meantime, EPT meets the need of translation from guest physical address (GPA) to host physical address (HPA), the real physical memory address.

B. VMCS

Virtualization technique imports two kinds of operation mode, root and non-root mode. The processor will trigger a VM exit and return to root mode form non-root mode, then handle some operations. VMCS, a data structure based on hardware, is imported to manage transitions into and out of VMX non-root operation (VM entries and VM exits) as well as processor behavior in VMX non-root operation.

III. THREAT MODEL AND ASSUMPTION

In this section, we describe threat model and assumption.

A. Threat Model

Attacker can attack hypervisor with its own vulnerability directly. Or attacker first compromises the VM, then obtains the permission of the hypervisor through the virtual machine escape attack, and attack other VMs at the runtime. On the one hand, the attacker modifies the critical interaction data in the context switching process between VM and hypervisor, including general purpose register information, privileged registers, etc. On the other hand, the attacker modifies the address mapping of the EPT of VM, causing remapping attacks and double mapping attacks. Our goal is to prevent these attacks. The specific implementation is as follows.

a) Modify the Interaction Data: For the modification of the critical interaction data of the context switching, the attacker can obtain the address and access privilege of the VMCS structure in the victim VM, modify the information of the VMCS, such as HOST_RIP, GUEST_CR0, EPTP, etc. For example, modifying the value of HOST_RIP register and writing malicious program address to the 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.

b) Modify the Address Mapping of EPT: Modification to EPT can result in malicious memory information leakage. There are two used scenes, double mapping and remapping attack.

Scene 1. 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 1. 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.

Scene 2. 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
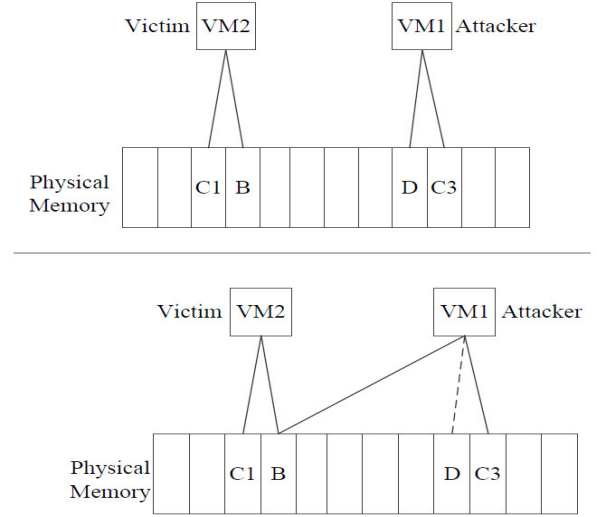

Fig. 1. The execution process of double mapping.

page A. By this way, VM1 can access the information on A used by VM2, causing information leakage.

B. Assumption

We propose some assumptions. First, we assume hardware resources are trusted including processor, buses and so on, the trusted boot based on hardware can ensure the security and integrity of bootloaders. Obviously, the TCB contains created HyperMI and hardware resources. Second, this paper does not consider denial of service attack (DOS), side channel attack and hardware-based attack. DOS attack is already trivial for a compromised hypervisor to deny service to VM [20]. Side channel attack has very limited bandwidth to leak data and is much hard to perform. Hardware-based attacks, such as cold-boot attacks and RowHammer, are harder to implement than software attacks under a certain time restriction, which is agreed with prior works [26].

IV. DESIGN

In this section, we give the overview about the HyperMI architecture.

A. Overview of HyperMI

HyperMI is designed to provide a secure isolated execution environment to protect running VM against compromised hypervisor without depending on the higher privilege level software or customized hardware.

Figure 2 shows the architecture, the overall system contains three parts: several isolated trustworthy VMs, hypervisor either in normal world or in HyperMI word, memory hardware including HyperMI region for HyperMI world and normal region for normal world.

The origin hypervisor is divided into two parts: HyperMI world which security tools can run in and normal world which hypervisor runs in. Firstly, HyperMI world is used to run security tools when hypervisor is compromised, so that these
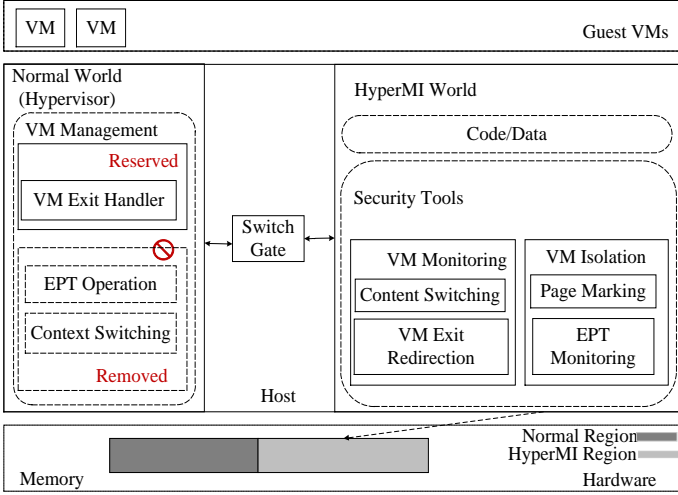
Fig. 2. The architecture of HyperMI.

Fig. 3. The control flow of HyperMI.

tools running in secure HyperMI world can resist attacks from compromised hypervisor. Secondly, operations for EPT and context switching module are deprived from normal world for security, then are put into HyperMI world, but VM exit handler module is reserved to handle VM exit in normal world. Thirdly, these two worlds can communicate with each other through the only secure channel, named Switch Gate.

While the hypervisor together with guest VMs run in the normal world, the hypervisor is forced to request HyperMI to perform four operations on its behalf: 1) switching context between the hypervisor and VMs, 2) updating EPT of VMs, 3) verifying the pages when executing swapping operations to resist double mapping attack, 4) verifying the pages when executing releasing operations to resist remapping attack. After setting up the HyperMI, the whole system is ready to create isolated executing environment. With these designs, HyperMI enforces the isolation and protection of memory used by each VM. Furthermore, HyperMI guarantees security of interaction, memory isolation between the hypervisor and VMs.

B. Control Flow of HyperMI

Figure 3 shows the control flow execution of HyperMI. In traditional system, VM interacts with hypervisor by VM EXIT/VM ENTRY to request hypervisor to handle events. The execution process of the interaction is as follows. 1) VM switches context and exits to hypervisor. 2) Hypervisor handles the VM exit request events. 3) Hypervisor executes VM resume instruction to entry VM. In HyperMI system, HyperMI intercepts the VM exit request, executes context switch, to make sure interaction data not be modified. The execution process of the interaction is as follows. 1) VM exits to HyperMI and switches context. 2) HyperMI exits to hypervisor. 3) Hypervisor handles the VM EXIT request events. 4) Hypervisor executes VM resume instruction to entry VM.
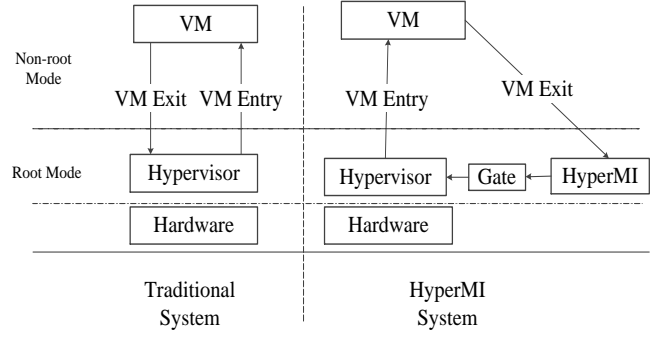
V. Implementation

We describe the implementation of key components in the HyperMI architecture below including HyperMI world, VM monitoring and VM isolation.

A. HyperMI World

HyperMI world, as a secure isolated execution environment, is created to place security tools to resist compromised hypervisor against leaking information, accessing data illegally and falsifying data. In order to make sure the isolation and security of HyperMI world, we implemented the following three strategies. Firstly, to achieve isolation, HyperMI world is created on another kernel page table. Secondly, the only and secure switch gate is required to provide switch channel for two different worlds. The switch gate is the only channel to avoid bypassing HyperMI world attack. The gate is secure by using atomic operation to avoid the address of HyperMI world leakage attack. Thirdly, some secure approaches must be adopted to protect HyperMI world against bypassing HyperMI world and breaking HyperMI world. Because attackers can load new page table and bypass HyperMI world by modifying the value of register CR3, attack running HyperMI world while the code in normal world has executive privilege.

Creation of HyperMI World HyperMI world is created to provide a secure execution environment for security tools against compromised hypervisor. To satisfy isolation, the code and data segment of HyperMI world must not be accessed by compromised hypervisor. We use two isolated address space based on two sets of page table to achieve isolation. Figure 4 describes the address space layout of two worlds through two sets of page table, the normal page table and the secure page table. On the left of Figure 4, the normal page table contains address of the normal world expect for address of HyperMI world in case of compromised hypervisor breaking

Kernel Code	RX
Kernel Data	RW
HyperMI Code	RX
HyperMI Data	RW
Switch Gate	RX

Normal World

Entry gate:
1 save kernel state
2 clear interrupt
3 write cr3
4 flush TLB
5 clear interrupt
6 jump to HyperMI world

Shared buffer:
common data and code

Exit gate:
1 flush TLB
2 write cr3
3 set interrupt
4 recover kernel state
5 jump to normal world

Switch Gate

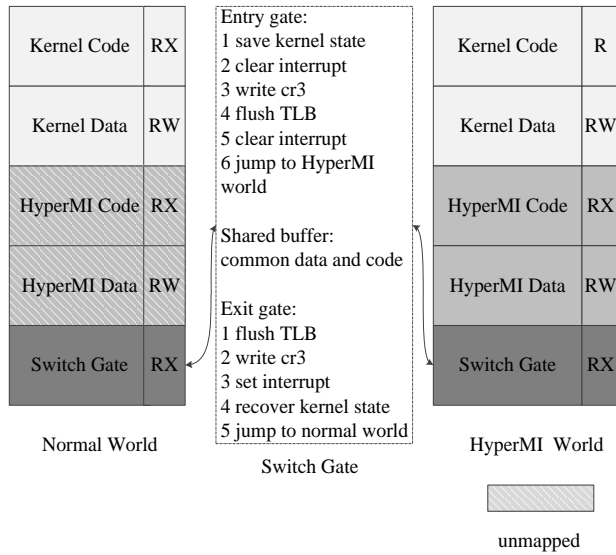| Kernel Code | R |
Kernel Data	RW
HyperMI Code	RX
HyperMI Data	RW
Switch Gate	RX

HyperMI World

unmapped

Fig. 4. An overview of address space layout.

the integrity of code and data in HyperMI world. Program running in normal world cannot access data in HyperMI world. On the right of Figure 4, all address are mapped in this secure page table. HyperMI code remains executable and HyperMI data remains writable. Kernel code is forbid to execute with reference to protection for running HyperMI world. In the middle of Figure 4, the switch gate includes entry/exit gate and shared buffer. Entry gate provides the only entrance to HyperMI world while the exit gate provides the address to return to the normal space. Shared buffer contains common data and code which the system needs to run the switch process. Common code is switch code, common data is entrance address to HyperMI world and return address to normal world. The switch gate is mapped at the same place in the normal world and HyperMI world because the page table loading code must be called by the two worlds before and after switching. Of course, the entrance address must be protected after switching to HyperMI world in case of malicious attacker accessing HyperMI world causally after trusted boot.

Obviously, HyperMI world is based on page table. There are three reasons for controlling the two set of kernel page tables: 1) To access casually or bypass HyperMI world, attacker can tamper the content of page table to map to the physical page belonging to HyperMI world or load malicious page table to CR3. 2) To execute code injection attack, attacker can close the write protection mechanism by modifying the value of CR0 register, changing the access permission bits of the memory page. Then cover the hooked functions we use, redirect the functions to their own malicious code and bypass secure monitoring of HyperMI. 3) To break HyperMI world, attacker can access HyperMI world causally when HyperMI world is running if kernel code has the execution permission. Therefore, three secure approaches against these attacks are as

follow.

For the first attack, we make some segment unmapped by creating another page table, named secure page table, which contains HyperMI code and data. And to protect the entrance address to HyperMI world from being leaked, remove all entries that map to HyperMI world from the page table in normal world. Deprive the ability of accessing CR3 of kernel in order to avoid to load illegal page table, and resist bypassing HyperMI world. For the second attack, intercept the accessing operation to CR0 and maintain the WP bit as 1. Stick to W⊕X mapping and code segment belonging to hooked functions still maintains unwritable. To go against the third attack, we set the kernel code segment as NX (non-executable) when HyperMI world is running. For more security, modify the kernel to configure all page tables as read-only by setting access permission bits of specific page table entries mapping to the memory regions of the page tables. This is necessary to prevent the page tables from being compromised by attackers. Any access permission modification to kernel page tables must cause the kernel to page fault, then we dispatch page fault to HyperMI world to handle.

Worlds Switch HyperMI creates a switch gate for switch between normal world and HyperMI world by loading a page table of the next space into CR3. And we must ensure atomicity and security during the switching process.

The switching process described in Figure 4 is as follow: 1) Save the kernel state to the stack including generic registers and interrupt enable / disable status. 2) Clear the interrupt with the CLI instruction. 3) Load the page table to the register CR3 and flush the TLB. 4) Interrupt again. 5) Jump to the HyperMI region. For the exit process, return to the normal world by performing the operations in the reverse order.

During this switching process, attackers can attack the system by violating atomicity and security. 1) Interrupt the gate's execution sequence and violate the atomicity. 2) Jump the first interrupt and get the base address of page table of HyperMI world after writing to CR3 can go against security. Therefore, interrupt policy is used to guarantee atomicity and twice interrupt is required to ensure security in case of attacker carrying out attack after getting the address of HyperMI world. Saving the kernel state can make running program return to normal world normally.

B. Security Guarantee for HyperMI World

Nevertheless, without any protection measures, the page table to load to switch to HyperMI world is not secure for three reasons: 1) Hypervisor with the highest privilege can control page table. 2) Free to execute privileged instructions. 3) Carry out DMA attack to access HyperMI world casually.

Firstly, hypervisor has full control of page tables, so it can attack the HyperMI world. Actually, protection for page tables is detailed in section V-A.

Secondly, hypervisor is still privileged and it can free to execute privilege instructions, so it can write any value to the related privileged registers. 1) Malicious attackers can close DEP mechanism by writing to CR0, close SMEP mechanism

by writing to CR4. 2) Kernel code can load a crafted page table to bypass the HyperMI world by converting a meticulously constructed address of one page table to CR3. Actually, to protect the system, HyperMI deprives sensitive privileges instructions executed by hypervisor, and dispatches the captured events to the HyperMI world. The HyperMI world can choose how to handle this event, such as issuing alerts, terminating the process, or doing nothing. The whole process is similar to the signal handling in traditional OSes.

Thirdly, it is important to focus on DMA attack. DMA operation is used by hardware devices to access physical address directly. Malicious attackers can read or write arbitrary memory regions including HyperMI world by DMA. Therefore, it is a key focus of intercepting access to physical pages belonging to HyperMI world directly by DMA operation. Fortunately, HyperMI employs IOMMU mechanism to avoid DMA attack, which can carry out access control for DMA access. Our approach adopts two policies: 1) Remove the corresponding mapping of the critical data from the page table which IOMMU uses. These critical unmapped data includes the entrance address of HyperMI, data recording Page-Mark structure used in VM isolation, VM-Mark structure used in VM monitoring and so on. 2) Intercept the address mapping functions about I/O, verify whether the address is in address space of HyperMI world, then choose to map or unmap.

C. VM Monitoring

During the VM entry/exit process, the related state information of every virtual CPU and host OS are stored in VMCS structure. And only the VMX root privilege instructions, such as VMPTRLD, VMPTRS, VMCLEAR, VMWRITE and VMREAD, can operate the VMCS. VM exit is a point to intercept accessing operation of VMCS. HyperMI intercepts and validates interaction between hypervisor and VM: 1) Interception of context switching. 2) VM exit redirection.

Interception of Context Switching Obviously, the protection to VMCS structure can't be ignored based on the fact that VMCS is structure recording all context informations of the VM and it is managed by the compromised hypervisor. The VMCS structure always records the information of privileged registers, such as HOST_CR3, EPT_POINTER, HOST_CR0, HOST_CR4, VM_EXIT_MSR_STORE_ADDR, HOST_RIP and so on. During the VM entry or VM exit, the compromised hypervisor can tamper VMCS structure, so the system can suffer these attacks: 1) Access memory region of VMCS directly. 2) Falsify the value of HOST_RIP, and the system will suffer control flow hijack. 3) Tamper the value of EPT_POINTER(EPTP), and other malicious EPT is loaded. 4) Fake the value of HOST_CR3, so the page table of host OS can be replaced.

For the first kind of attack, to prevent compromised hypervisor accessing memory region of VMCS structure, HyperMI hides the base address of VMCS structure in HyperMI world. So, hypervisor loses the ability of accessing VMCS structure. To ensure the system run normally, hypervisor must require HyperMI to return the signal information rather than real
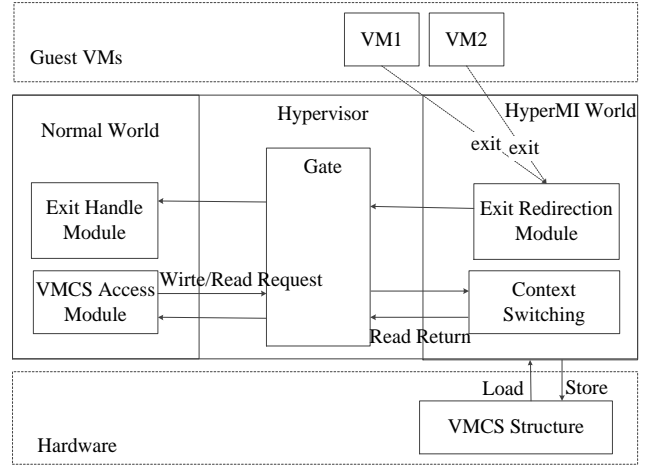

Fig. 5. An overview of VM monitoring.

address on demand or trap the functions to the HyperMI world, and avoid many functions about VMCS operations to access the address of VMCS structure directly. To avoid the last kind of attack, in addition to hiding the address, HyperMI also intercepts and validates the execution of these instructions by placing hooks at these functions (vmcs_writel, vmcs_readl et al.). So hypervisor requests HyperMI world to handle operations about VMCS and return corresponding result for legal request, as Figure 5 shows.

VM Exit Redirection VMCS structure can only be accessed during VMX root operation, therefore, the switching point between VMX Non-Root operation and VMX Root operation provide HyperMI a perfect point to perform security check. In details, HyperMI would intercept the process of VM Exit and redirect to HyperMI world to perform security check. To intercept the context switching efficiently, HyperMI chooses a novel way showed in Figure 5. Firstly, trap events of VM exit, when VM exit events happen, HyperMI saves the hypervisor states firstly for security and then dispatches exit events to hypervisor to handle. To cut down the performance overload, HyperMI set VM exit configure of conditional exit events as non-running and reduce the occurrence number of exit events.

D. VM Isolation

As known to all, when Intel VT-d is enabled, all the physical memory is managed by the hypervisor using Extended Page Tables (EPTs), EPTs must be involved in each memory access and determine the permissions of the accessed memory page frame. Besides, the hypervisor employs different EPTs to manage the corresponding physical memory area for different VMs. If the page is remapped or double mapped, the system causes page fault. Page is accessed by VM through EPT updating. it is important to control page mapping when EPT updates. To achieve the goal of VM isolation completely, we adopt page marking technique, that marks memory page with flag and track the page. However, a time point is needed for

marking all page. And it is no doubt that EPT updating is a great point because hypervisor can manage all page mapping efficiently when EPT updates. Besides, EPT must be protected in case of data being leaked because EPT plays an important role in address translation of VM. So we describe VM isolation in these two aspects, EPT interception and memory isolation for VM.

Interception of EPT Operation During the translation process of mapping GPA to HPA through EPT, a VM can suffer these attacks: 1) Compromised hypervisor can access EPT of every VM, then does whatever it wants by modifying the address mapping causing remapping and double mapping attack. 2) Load the malicious EPT and execute illegally.

Confronting with these attacks, we can see that the ultimate purpose of attackers is to access the physical memory page frames of other VM. HyperMI solves the issue using the following policies: 1) Hide the address of EPT and make it unmapped for kernel in normal world. 2) Intercept the related operations including EPT creating, loading, updating, walking and destroying to avoid leaking the address of EPT. 3) Mark respective EPT for every VM, isolate EPT among VMs, and ensure that the right and corresponding EPT is loaded for every VM.

TABLE I
VM-MARK TABLE.

VM-Mark Table			
Label	VMID	EPTID	EPT_Address
Description	The VM Identifier	The EPT Identifier	The Entry Address of EPT

For the first policy, three places should be particularly protected. The first place is the EPT creation function where the function would return a CR3-like address: EPTP value. The second place is VMCS where would record the value of corresponding EPTP. The last place is the hardware register EPTP, the work of this register is similar to that of the CR3 register, which is used to provide MMU with the location of EPT. Therefore, it is critical to intercept the EPT creating operation and protect the stored address of EPT in HyperMI world. Especially, it is important to store VMCS structure who contains the value of EPTP in HyperMI world. However, some functions, in addition to the EPT creation function, still rely on the value of EPTP. In this paper, HyperMI provides a novel approach that HyperMI returns signal information necessarily to these functions rather than true address to make the system run normally. For the second policy, some functions, EPT creating, loading, walking and destroying, need access address of EPT. HyperMI places hooks on these functions, then dispatches them to HyperMI world and handle appropriately. In the meantime, HyperMI handles double mapping to ensure that there is only one virtual address mapping to one physical memory page during the EPT updating, and handles remapping problems to ensure the content of page cleaned after page being swapped out. This will be described in detail later. For the last one, intercepting the loading EPT operation and
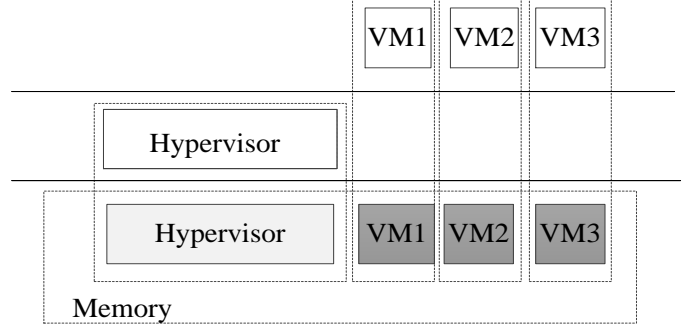

Fig. 6. Memory isolation for VM.

TABLE II
PAGE-MARK TABLE.

Page-Mark Table			
Label	OwnerID	SharedBit	Used
Description	The Owner Identifier	Shared or Unshared	Free or Used

verifying the correctness of EPT can avoid loading a wrong EPT and leaking the content of physical memory page. To ensure one EPT for one VM, HyperMI creates the VM-Mark structure stored in HyperMI world as Table 1 described, and record VMID, EPTID, EPT_Address and binds them together. VMID is created and destroyed based on hash value of image of VM. EPTID and EPT_Address is recorded as long as the EPT of current VM is created.

Memory Isolation for VM In addition the isolation EPTs mentioned above, isolating memory is another aspect that should be considered to achieve the goal of VM isolation completely, which is depicted in Figure 6. Hypervisor and every VM just can access own physical memory. Without memory isolation mechanism, compromised hypervisor and compromised VM can access memory pages of victimized VM by two ways when EPT updates: 1) Double mapping. 2) Remapping to pages with content. So some efforts are done. Monitoring EPT updating and creating Page-Mark structure described in Table 2 to record the owner of every physical memory page.

In order to against the double mapping attack in the process of EPT updating, HyperMI should finish these two tasks: verifying the owner of pages firstly, and then marking the OwnerID of Page-Mark structure for unused pages or thwart the mapping operation for used pages in case of malicious double mapping behavior. So the technique of pages allocation can divide all the pages into different catalogues: the pages of hypervisor or the pages of every VM.

To go against the remapping attack, HyperMI cleans the context of the page when the page is swapped out, so attackers can't get the context of the page by the way of remapping.

And HyperMI clear the Page-Mark structure of corresponding page. This solution is the same as the treatment method of pointer for the purpose of security protection.

VI. Evaluation

A. Security Analysis

Through the above design and implementation, we discuss in detail how HyperMI achieves HyperMI world, VM monitoring and memory isolation for VM. Based on our threat model, an attacker aims to escape from VM and further compromises the hypervisor or control other VMs. Afterwards, we discuss how HyperMI prevents the following possible attack scenarios.

Subverting memory protection across VMs attack A kind of attack is subverting memory protection across VMs. Memory of VM is managed by Hypervisor through EPT, so compromised hypervisor can conduct double mapping and remapping attack. However, HyperMI conducts all operations about EPT, and prevent double mapping and remapping attack. Firstly, double mapping happens when assigning memory pages that have already been owned by a hostile VM to a victim VM. This kind of attack is prevented by page tracking and write-protection of EPT. For each new mapping to a VM, HyperMI validates whether the page is already in use. Meanwhile, the allocated pages must be marked in the PUT table for tracking. Secondly, another challenge is page remapping attack by a compromised hypervisor from a victim VM to a conspiratorial VM. This attack involves remapping a private page to a different address space. To defeat this type of attack, HyperMI ensures that whenever an unsharable page is released, its content must be zeroed out before creating a new mapping.

DMA attack In addition, the memory can be accessed through DMA operations bypassing the MMU, except accesses by executing memory accessing instructions. DMA attack is described in detail in section V-B. Attackers can use this feature to read or corrupt arbitrary memory regions. DMA attacks are not a threat to HyperMI, because HyperMI is inherently secure against DMA using IOMMU. DMA attacks that aim at modifying the VM memory or the page tables will also be defeated.

Modify interaction data of context switch attack A potential attack is that attackers may attempt to modify a VM runtime states to disturb a VM execution. This attack is not feasible in HyperMI. The HyperMI world is enough secure to bypass, interacting between hypervisor and VM runs in HyperMI world, VMCS structure used to record context switching data is hidden in HyperMI world, so attacker cannot modify VM states during context switching. HyperMI adopts VM-Mark table to ensure that load consistent EPT for every VM, attacker cannot modify EPTP. Therefore, the VM states cannot be modified.

To further validate the security of HyperMI, we examine several real vulnerabilities. Here we only make analysis on these vulnerabilities and demonstrate that HyperMI is immune to these vulnerabilities. The first examined vulnerability showed in Table 3 is CVE-2009-2287 [7], which can fake

TABLE III
HYPERVISOR ATTACKS AGAINST.

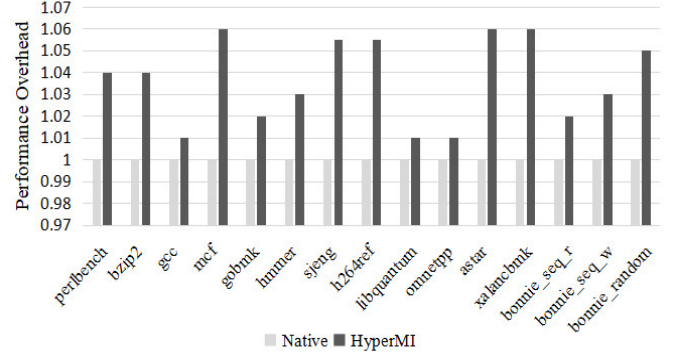| Attack	Description
CVE-2009-2287	Load a crafted CR3 value
CVE-2017-8106	Load a crafted EPT value
DMA Attack	Access HyperMI world by DMA
Code Injection Attack	Inject code and cover hooked functions to bypass HyperMI world

Fig. 7. Performance overhead.

a VM execution via a crafted page table root in KVM hypervisor. This is prevented by hiding VMCS structure in HyperMI world during the secure context switching. CVE-2017-8106 [8], loading a wrong EPT, can be prevented by verification for EPT when EPT loads. DMA attack and code injection attack can be avoided by policies described in section V-B.

B. Performance Evaluation

All experiments are done on a server with 64 cores and 32 GB memory, running at 2.0 GHz. The version of hypervisor KVM is 4.4.1. Each guest VM is with Linux kernel 4.4.1 and configured with 2 virtual cores and 2 GB memory. All experiments are done 50 times and results are from the average.

To better understand the factor causing the performance overhead, we experiment with compute-bound benchmark (SPEC CPU2006 suite) and one I/O-bound benchmark (Bonnie++) running upon original KVM and HyperMI in a Linux VM. For Bonnie++, we choose a 1000 MB file to perform the sequential read, write and random access. The experiments result described in Figure 7(the last three groups) shows relatively low cost. Most of the SPEC CPU2006 benchmarks (the first twelve groups) show less than 6% performance overhead. It's not surprising as there are few OS interactions and these tests are compute-bound. Mcf, astar, and xalancbmk with the highest performance loss allocate lots of memory, and HyperMI handles Page-Mark structure when EPT updates. This can incur worlds switching which involves fewer register access and fewer TLB flushes with PCID technique because the two worlds are at the same privileged layer. For Bonnie++, the performance loss of sequential read, write and random

access is 2%, 3% and 5%, the main reason is that HyperMI has no extra memory operations for I/O data.

VII. RELATED WORK

We describe the related work from these three aspects, integrity verification for hypervisor, resource isolation based on hardware and software, and the same privilege level isolation.

A. Protection for Hypervisor

Integrity Verification for Hypervisor In order to ensure the security of the hypervisor during trusted boot and runtime, an effective and commonly used method is to verify the integrity of the hypervisor, and reduce the attack surface. For the security of the hypervisor during trusted boot, paper [17] proposes control flow integrity protection policy, by verifying regularly control flow integrity behavior to detect rootkit attacks. However, attacker can detect the regular and bypass the detection. For runtime security of the hypervisor, HyperSafe [24] and HyperCheck [21] choose pooling-query method based on SMM to finish integrity verification of hypervisor. However, SMM doesn't support for MMU. And attackers can hide trace during polling-query intervals when comparing to event-driven monitoring.

B. Resource Isolation

Isolation Based on Hardware Some works at the hardware level complete the protection of the process by extending the virtualization capabilities. These tasks provide fine-grained isolation of processes and modules from the hardware level. Haven [2] uses Intel SGX [11], [15] 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. The effort [6] combines the advantages of ARM TrustZone and virtualization to improve system performance, and isolate critical process components securely and efficiently. H-SVM [12] utilizes the hardware extension features of the CPU, and extends SMM microcode to achieve memory resource isolation among virtual machines. It deprives ability of accessing to memory resource by replacing the source code of the original hypervisor to access memory resource. Vigilare [16] and KI-Mon [14] provide monitoring for access operations by introducing extra hardware. Vigilare provides a kernel integrity monitor that is architected to snoop the bus traffic of the host system from a separate independent hardware. It adds extra Snooper hardware connections module to the host system for bus snooping. KI-Mon monitors write operation to system bus and handles data to write in order to check rootkit attack.

Isolation Based on Software Expect for approaches based on hardware, some works [18], [19], [25] pay attention to software isolation. Pre-allocating physical resource and completed isolated environment for every VM can avoid VM cross-domain attack, and data leaking attack. NOVA [19] 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 [25] prepares backup KVM for every VM by copying KVM code, and ensures every VM run in own isolated space. Nexen [18] 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. These approaches redesign hypervisor greatly. In contrary, HyperMI adopts a feasible way to isolate VM without lots of modification to hypervisor.

C. The Same Privilege Level Isolation

Some efforts, ED-Monitor [9], SKEE [1] and SecPod [22], 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. SKEE provides a lightweight secure kernel-level execution environment for ARM, this environment is placed at the same privilege level with kernel. When kernel is compromised, attacker can't break the isolation between SKEE and the kernel, and the security of internal security tools placed at secure isolated environment is guaranteed. SecPod, an extensible approach for virtualization-based security systems that can provide both strong isolation and the compatibility with modern hardware. SecPod has two key techniques: paging delegation delegates and audits the kernel's paging operations to a secure space; execution trapping intercepts the (compromised) kernel's attempts to subvert SecPod by misusing privileged instructions.

VIII. CONCLUSION

We introduce HyperMI, 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 VM, and secure and event-driven runtime monitoring for interaction between hypervisor and VMs. This approach, which does not rely on additional hardware devices or a higher privilege level software, has fewer changes to system and fewer requirements for types of CPU hardware device. It reflects good practicality and portability. And security analysis describes protection for VM, the performance evaluation shows its efficiency by introducing negligible performance overhead. It can be implemented widely in real-world for cloud providers.

REFERENCES

[1] Azab, A., Swidowski, K., Bhutkar, R., Ma, J., Shen, W., Wang, R., Ning, P.: Skee: A lightweight secure kernel-level execution environment for arm. In: Network and Distributed System Security Symposium (2016)

[2] Baumann, A., Peinado, M., Hunt, G.: Shielding applications from an untrusted cloud with haven. Acm Transactions on Computer Systems **33**(3), 1–26 (2014)

[3] Ben-Yehuda, M., Day, M., Dubitzky, Z., Factor, M., Har'El, N., Gordon, A., Liguori, A., Wasserman, O., Yassour, B.A., Ben-Yehuda, M.: The turtles project: Design and implementation of nested virtualization. Yehuda pp. 1–6 (2007)

[4] Champagne, D., Lee, R.B.: Scalable architectural support for trusted software. In: HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture. pp. 1–12 (2010)

[5] Chhabra, S., Rogers, B., Yan, S., Prvulovic, M.: Secureme:a hardware-software approach to full system security. In: International Conference on Supercomputing. pp. 108–119 (2011)

[6] Cho, Y., Shin, J., Kwon, D., Ham, M.J., Kim, Y., Paek, Y.: Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices. In: Usenix Conference on Usenix Technical Conference. pp. 565–578 (2016)

[7] CVE-2009-2287: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2287 (2018)

[8] CVE-2017-8106: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-8106 (2018)

[9] Deng, L., Liu, P., Xu, J., Chen, P., Zeng, Q.: Dancing with wolves: Towards practical event-driven vmm monitoring. Acm Sigplan Notices **52**(7), 83–96 (2017)

[10] Evtyushkin, D., Elwell, J., Ozsoy, M., Ponomarev, D., Ghazaleh, N.A., Riley, R.: Iso-x:a flexible architecture for hardware-managed isolated execution. In: Ieee/acm International Symposium on Microarchitecture. pp. 190–202 (2015)

[11] Hoekstra, M., Lal, R., Rozas, C., Phegade, V., Cuvillo, J.D.: 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)

[12] Jin, S., Ahn, J., Seol, J., Cha, S., Huh, J., Maeng, S.: H-svm: Hardware-assisted secure virtual machines under a vulnerable hypervisor. IEEE Transactions on Computers **64**(10), 2833–2846 (2015)

[13] Keller, E., Szefer, J., Rexford, J., Lee, R.B.: Nohype:virtualized cloud infrastructure without the virtualization. Acm Sigarch Computer Architecture News **38**(3), 350–361 (2010)

[14] Lee, H., Moon, H., Jang, D., Kim, K., Lee, J., Paek, Y., Kang, B.H.: Ki-mon: a hardware-assisted event-triggered monitoring platform for mutable kernel object. In: Usenix Conference on Security. pp. 511–526 (2013)

[15] Mckeen, F., Alexandrovich, I., Berenzon, A., Rozas, C.V., Shafi, H., Shanbhogue, V., Savagaonkar, U.R.: Innovative instructions and software model for isolated execution. In: International Workshop on Hardware and Architectural Support for Security and Privacy. pp. 1–1 (2013)

[16] Moon, H., Lee, H., Lee, J., Kim, K., Paek, Y., Kang, B.B.: Vigilare: toward snoop-based kernel integrity monitor. In: ACM Conference on Computer and Communications Security. pp. 28–37 (2012)

[17] Petroni, N.L., Hicks, M.: Automated detection of persistent kernel control-flow attacks. In: ACM Conference on Computer and Communications Security. pp. 103–115 (2007)

[18] Shi, L., Wu, Y., Xia, Y., Dautenhahn, N., Chen, H., Zang, B., Guan, H., Li, J.: Deconstructing xen. In: Network and Distributed System Security Symposium (2017)

[19] Steinberg, U., Kauer, B.: Nova:a microhypervisor-based secure virtualization architecture. In: European Conference on Computer Systems, Proceedings of the European Conference on Computer Systems, EUROSYS 2010, Paris, France, April. pp. 209–222 (2010)

[20] Wang, B., Zheng, Y., Lou, W., Hou, Y.T.: Ddos attack protection in the era of cloud computing and software-defined networking. Computer Networks the International Journal of Computer & Telecommunications Networking **81**(C), 308–319 (2015)

[21] Wang, J., Stavrou, A., Ghosh, A.: Hypercheck: a hardware-assisted integrity monitor. In: International Conference on Recent Advances in Intrusion Detection. pp. 158–177 (2010)

[22] Wang, X., Chen, Y., Wang, Z., Qi, Y., Zhou, Y.: Secpod: a framework for virtualization-based security systems. In: Usenix Conference on Usenix Technical Conference. pp. 347–360 (2015)

[23] Wang, X., Qi, Y., Dai, Y., Shi, Y., Ren, J., Xuan, Y.: Trustosv: Building trustworthy executing environment with commodity hardware for a safe cloud. Journal of Computers **9**(10) (2014)

[24] Wang, Z., Jiang, X.: Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In: Security and Privacy. pp. 380–395 (2010)

[25] Wang, Z., Wu, C., Grace, M., Jiang, X.: Isolating commodity hosted hypervisors with hyperlock. In: Proceedings of the 7th ACM european conference on Computer Systems. pp. 127–140 (2012)

[26] Zhang, F., Chen, J., Chen, H., Zang, B.: Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In: ACM Symposium on Operating Systems Principles. pp. 203–216 (2011)