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

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

***Abstract*—Underlying hypervisor provides all kinds of services for VMs including resource allocation, resource management and so on. Sensitive data and critical content in guest virtual machine is easy to be leaked once hypervisor is compromised. For example, VM escape attack that attackers escape from a guest VM and attack directly the host OS. Cross-domain attack is that a malicious VM attacks another VM who shares the same hypervisor. Therefore, VM isolation is important that protect the memory of VM from malicious access causally. Previous efforts employ extra customized hardware which is not convenient for cloud providers to adopt widely or employ new architecture relying on a higher privilege level which introduces high performance overhead.**

This paper proposes HyperMI, a novel approach that provides protection for VM in cloud computing against compromised hypervisor. In order to prevent customers' sensitive data in VM from being leaked as long as the hypervisor depended on is attacked. It provides a privilege-level secure isolated execution environment as TCB, named HyperMI world, used for effective VM isolation and event-driven VM monitoring. Unlike other approaches of protecting VM, HyperMI world is placed at the same privilege level with the compromised hypervisor, incurring low overhead. The approach provides the runtime protection for VM by using a novel way that isolate memory among VMs and hypervisor securely. The event-driven VM monitoring can ensure the accuracy which polling-inquire can't provide. The key of HyperMI is decoupling the function of monitoring interaction between hypervisor and VM into VM monitoring module and decoupling the function of memory isolation from hypervisor into VM isolation module. As a result, HyperMI controls page mapping and accessing, isolates memory completely when VM's new mapping because of frequent page fault. 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 a large number 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 and Xen) have been increased to be hundreds of thousands of lines. However, recent survey has shown that commodity hypervisor incurs more vulnerabilities because of the larger code bases. From 2004 to now, there are 130 vulnerabilities about KVM [**?**]. Some of them (e.g., CVE-2018-1087 [**?**]) have been shown to be 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. Some 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 customizated 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. Bastion, uses modified microprocessor hardware to provide protection for security-critical OS and application modules in an untrusted software stack.

Reconstructed Hypervisor To achieve the goal of VM isolation, some efforts including 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 higer-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. However, there is no perfect approach to meet all the requirements at the same time.

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. Some 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 as kernel on ARM platform.

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 isolation between VMs. The VM monitoring module provides monitoring for the interaction between compromised hypervisor and each VM. Firstly, inspired by the idea of "same privilege level" isolation, HyperMI world is created at the same privilege level with hypervisor without introducing high severe performance overhear. Also, no part of our approach runs at a higher privilege level than the original hypervisor. Secondly, for the VM isolation module, since compromised hypervisor with the highest privilege can access all memory of VMs casually, HyperMI deprives the address translation capability of the compromised hypervisor. It modifies the memory management function to isolate every physical memory page with page marking technique, and guarantees that one physical memory page can only be used and owned by a guest VM. Meanwhile HyperMI verifies the correctness of owner for every page to prevent casual mapping. Thirdly, event-driven VM monitoring provides protection for VM isolation by trapping corresponding functions. Some approaches provide VM monitoring using polling-query, however, polling polling-query can't detect the malicious actions during polling-query intervals, when compared to even-driven monitoring. For VM Isolation, there are some especially critical data structures need to be monitored, such as Extended Page Tables (EPT), EPT Pointer (EPTP) and Virtual Machine Control Structure (VMCS). EPT is an important data structure that contains address mapping relationship between the Guest Physical Address and Host Physical Address. EPTP is a CR3-like register used to identify the location of the corresponding EPT. VMCS is another important data structure used in VMX operation to manage transitions into and out of VMX non-root operation. As long as these three data structures are attacked, the security of VM can't be guaranteed. Therefore, HyperMI provides VM monitoring for them.

Our main contributions are as follows:

- A secure isolated execution environment placed at the same privilege level with hypervisor.
- An approach of isolating memory securely for VM by using page marking technique against compromised hypervisor.
- A non-bypassable hypervisor monitoring approach which can ensure the security of interaction between hypervisor and VM.

We propose and implement a prototype based on KVM and x86 architecture. Our prototype efforts 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.

II. Background

A. Memory Management Unit

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. Virtual Machine Control Structure (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.

C. TLB

A Translation Lookaside Buffer (TLB) is a hardware component that concentrates on decreasing the overhead of address mapping by caching recently used virtual-to-physical address mappings. When lots of tasks run in a system concurrently, cached TLB entries are flushed at every context switch between tasks to prevent data from being leaked. However, such frequent flushes for all TLB would incur a substantial increase of TLB miss rates. To decrease redundant TLB flushes for every switch, there is a feature which tags every TLB entry with an identifier, named Process-Context Identifiers (PCID) on x86 platform. Only TLB entries that are associated with the current PCID are available to the CPU.

III. Threat Model and Assumption

In this framework, we pay attention to the runtime protection for VM. We consider host OS and hypervisor are at high risk of being attacked. Only VM monitoring and VM isolation running in HyperMI world can be fully trusted. And we assume hardware resources are trusted including processor, buses and so on. So the TCB contains created secure monitoring and hardware resources.

A. The information leakage attack

The attacker obtains the permission of the hypervisor through the virtual machine escape attack, and attack the VM 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. The specific implementation is as follows.

For the modification of the critical interaction data of the context switch, the attacker can obtain the address and access authority of the VMCS structure in the victimized VM, modify the information of the VMCS, such as HOST_RIP, HOST_CR3, GUEST_CR3, GUEST_CR0, EPTP, etc. Modifying the value of 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.

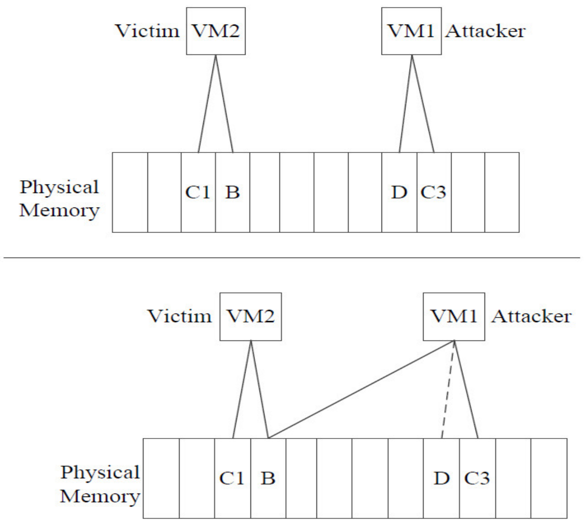

Fig. 1. The execution process of double mapping.

For the double mapping attack, the attacker first breaks through a VM, and then obtains the privilege of the hypervisor through the virtual machine escape attack, and maliciously accesses the VMCS structure to obtain the EPTP. The attack process is as shown in Figure 1. In this way, the EPT addresses 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 remap 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.

B. Assumption

We do 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, this is similar to prior works [26]. And we don't consider the attack incurred by interaction between VM and software with vulnerabilities. We assume the trusted boot can ensure the security and integrity of bootloaders.

Our approach can avoid these attacks including accessing any memory pages by page remapping or double mapping, malicious supervisor accessing to memory from hypervisor, malicious DMA (direct memory access) accessing and memory escalation attacks etc. Besides, protection for critical interaction data between hypervisor and VM can prevent attacker from implementing data leakage attack, modifying special privilege register, even control flow hijack.

IV. Design and Implementation

In this section, we give the overview about the HyperMI firstly and then detail every module.

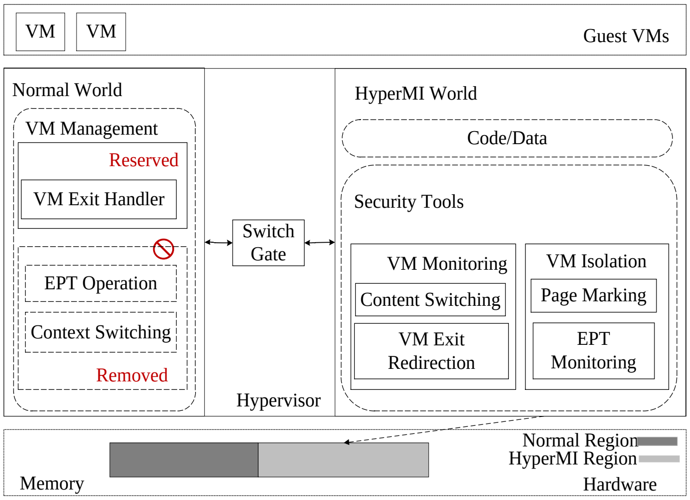

Fig. 2. The architecture of HyperMI.

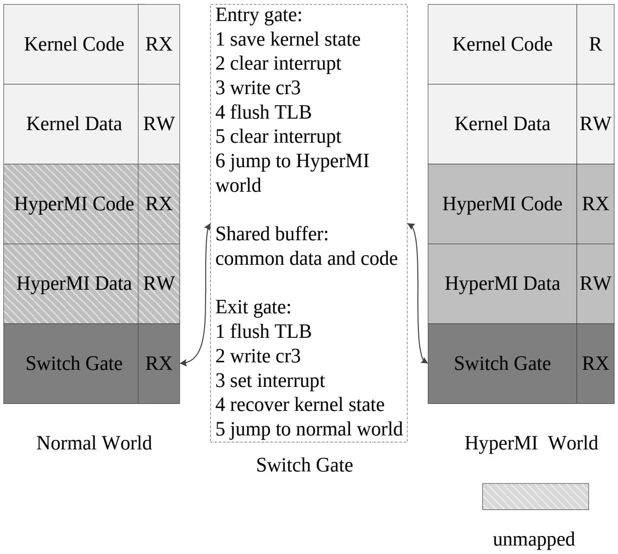

Fig. 3. An overview of address space layout.

A. Overview

HyperMI is designed to provide a secure isolated execution environment to protect running VM against compromised hypervisor without depending on the higher privilege level. The approach achieves 3 objects: 1) A secure isolated environment at the same privilege level with hypervisor to host security tools with relatively trial performance overhead. 2) Completed secure memory isolation approach for VM to guarantee memory isolation against cross-domain attack. 3) Non-bypassable secure runtime monitoring for interaction between hypervisor and VM when process exits from non-root mode to root mode. HyperMI also monitors access to VMCS data structure to finish context switching.

As Figure 1 shows, the origin hypervisor is divided into two parts: HyperMI world which security tools can run in and normal world which hypervisor runs in. Firstly, 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. HyperMI world is created to place security tools including VM monitoring and VM isolation. For VM isolation, page marking module is used to mark physical memory pages when EPT updates and isolate memory among VMs. Also, protecting EPT and monitoring the related operations about EPT are provided by EPT monitoring module. For VM monitoring, we intercept the related functions about VMCS structure by setting hooks in normal world, then dispatch these functions to HyperMI world to handle, and monitor the interaction between hypervisor and every VM. Secondly, as long as the monitored event is triggered, these two worlds switch through the only secure channel, named Switch Gate. Thirdly, memory of the whole system is divided into two parts, memory of normal region and HyperMI region. In the meantime, normal world can't access memory of HyperMI Region casually, which is isolated securely from normal region.

B. 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 the meantime, efforts must be adopted for the security of this environment. Firstly, the only and secure switch gate is required to provide switch channel for two different worlds. Secondly, some secure approaches must be adopted to protect HyperMI world against bypassing HyperMI world and breaking HyperMI world.

Creation of HyperMI World Figure 2 describes the address space layout of two worlds through two sets of page table. On the left of Figure 2, the normal page table contains address of the normal world expect for address of HyperMI world in case of compromised hypervisor breaking the integrity of code and data in HyperMI world. On the right of Figure 2, security tools run in HyperMI. 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 2, 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 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$\oplus$X mapping and code segment belonging to hooked functions still maintains non-writable. 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 2 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.

C. Security Guarantee

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 IV-B.

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 makes no doubt of focusing on DMA attack. DMA operation is used by hardware devices to access physical address directly. DMA, this feature can be used by malicious attackers to read or write arbitrary memory regions. 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 framework 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, VM-Mark structure 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.

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 acess 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, so it is important to control page mapping when EPT updates. To achieve the goal of VM isolation completely, we adopt the approach that mark 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. 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 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 perticularly protected. The first place is the EPT creatation function where the funtion 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 funtions, in addition to the EPT creatation 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 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.

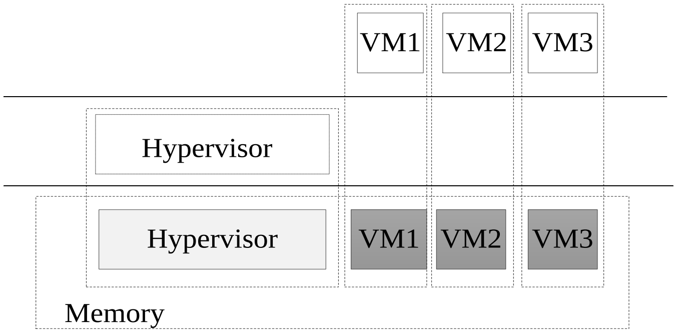

Fig. 4. Memory isolation for VM.

TABLE II
PAGE-MARK TABLE.

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

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 shoule be considered to achieve the goal of VM isolation completely, which is depicted in Figure 4. 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. Monitor EPT updating and create Page-Mark structure described in Table 2 to record the owner of every physical memory page.

In order to counter 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 marks the Page-Mark structure of corresponding page unused. This solution is the same as the treatment method of pointer for the purpose of security protection.

Besides, becuase of the page sharing mechanism in the system, which may hinder the implementation of the above measures, HyperMI deprives the shared page setting function and sets the corresponding bit in Page-Mark structure. The approach is detailed as follow: 1) Hypervisor requests HyperMI to return the hash map array (page frame number and hash value based on content). 2) HyperMI collects the hash array with considering that the same content results in the same hash value. 3) Hypervisor gets the hash array, finds the pages, then updates stable tree and unstable tree. 4) Hypervisor requests HyperMI to merge the page and set the SharedBit flag of the corresponding physical memory page. HA-VMSI [27] has a similar way with this idea.

E. 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 address on demand or trap the functions to the HyperMI world, and avoid many related functions 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 4 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 checkes. 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 4. 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.

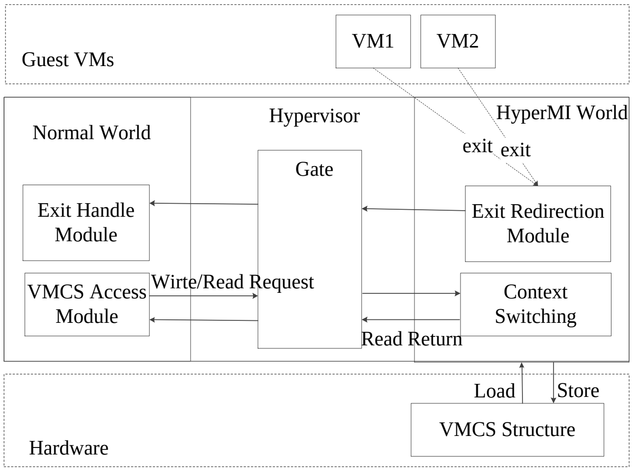

Fig. 5. An overview of VM monitoring.

V. Evaluation

A. Security Analysis

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

Attack against the Secure Isolated Execution Environment HyperMI uses page mechanism and related registers. So the protection for kernel page table and privilege registers can't be ignored when considering the attack of modifying page table item and registers. Attacker can achieve control flow hijack attack, leaking address of HyperMI world or critical data, loading malicious page table or covering hooked functions of event-driven monitoring to bypass secure monitoring, and even DMA attack. To avoid the above attacks, some measures are adopted. HyperMI intercepts modifying operation to CR4 and CR0 to ensure SMEP and DEP opened, prevents control flow hijack attack and code injection attack. And it adopts secure switching approach to prevent the entrance address of HyperMI world from being leaked, intercepts related operations about CR3 registers and sets kernel code non-executable in normal world.

Attack against VM Monitoring During the interaction between hypervisor and VM, attacker can attack the system

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

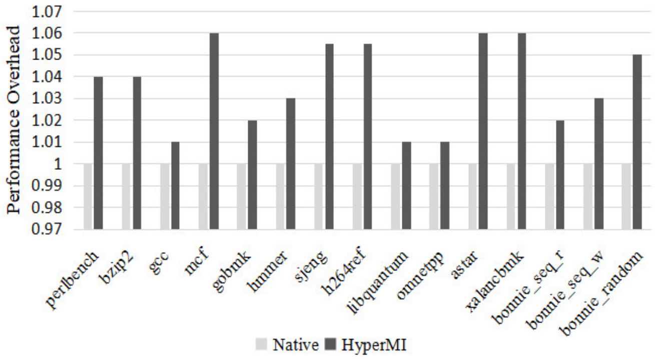

Fig. 6. Performance overhead.

by loading malicious EPT, double mapping or remapping to physical memory page of victimized VM when EPT updating, modifying VMCS structure, even DMA attack. To avoid these attacks, as described in section IV-D, HyperMI adopts VM-Mark table to ensure that load consistent EPT for every VM, and validate the owner of physical memory page when EPT updates in order to avoid double mapping or remapping attack. VMCS structure is hidden in HyperMI world and invisible in normal world, also, the related operation about VMCS structure is intercepted to decrease the possibilities of being attacked. Also, IOMMU is used to prevent DMA attack.

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 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 IV-C.

B. Performance Analysis

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.

Macro Benchmarks 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 5(the last 3 groups) shows relatively low cost. Most of the SPEC CPU2006 benchmarks (the first 12 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.

VI. RELATED WORK

We describe the related work from these 3 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 framework 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 framework for virtualization-based security systems that can provide both strong isolation and the compatibility with modern hardware. SecPod has two key techniques: paging delegation delegates and audits the kernel's paging operations to a secure space; execution trapping intercepts the (compromised) kernel's attempts to subvert SecPod by misusing privileged instructions.

VII. 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=C` (2018)

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

[27] Zhu, M., Tu, B., Wei, W., Meng, D.: Ha-vmsi: A lightweight virtual machine isolation approach with commodity hardware for arm. In: ACM Sigplan/sigops International Conference on Virtual Execution Environments. pp. 242–256 (2017)