HyperMI: A Privilege-level VM Protection Approach against Compromised Hypervisor

Abstract—**Remote or local adversaries can easily access other customers' sensitive data in the memory or subvert other guest virtual machines (VMs), once the hypervisor is compromised and controlled by them. Therefore, it is essential to protect VMs, especially the context and the memory of VMs, against the compromised hypervisor. Some previous efforts rely on extra customized hardware and higher privileged system components. These approaches increase both the maintenance effort and the code base size of privileged system components, which consequently increases the risk of having security vulnerabilities.**

This paper proposes HyperMI, a privilege-level VM protection approach against the compromised hypervisor. HyperMI is designed to be placed at the same privilege-level with the compromised hypervisor, it is smoothly embedded into the current commercial cloud environment, which presents no requirements for the upper VMs and the underlying existing hardware. HyperMI can be applied to multi-platforms too. The key of HyperMI is that it decouples the functions of interaction between VMs and the hypervisor, and the functions of physical memory management from the compromised hypervisor. As a result, HyperMI isolates memory completely, controls memory mapping when a page is allocated, and resists malicious memory access to VMs from the compromised hypervisor. We have implemented a prototype for KVM hypervisor on x86 platform with multiple VMs running Linux. KVM with HyperMI can be applied to current commercial cloud computing industry with portability. The security analysis shows that this approach can provide VMs with effective isolation and protection from the compromised hypervisor, and the performance evaluation confirms the efficiency of HyperMI.

Index Terms—**Virtualization, VM Protection, VM Security**

I. Introduction

As more and more functionalities are added into the hypervisor, the code bases of commodity hypervisors (KVM or Xen) increase to be large lines. On the one hand, commodity hypervisor has more vulnerabilities because of the larger code bases. On the other hand, because hypervisor possesses the highest privilege in the cloud environment, an attacker who compromises hypervisor could harm the whole cloud infrastructure and endanger data in the cloud. Being aware of such serious situation, current researchers try to alleviate those vulnerabilities by customizing hardware or software on a higher level than hypervisor.

Customized Hardware Some efforts (SecureME [5], Bastion [4] and Iso-x [8]) 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.

Reconstructed Hypervisor Some efforts (NoHype [10] and TrustOSV [17]) pre-allocate fixed cores or memory resource to isolate VMs via reconstructing hypervisor. In the meantime, these efforts deprive some virtualization capabilities and introduce lots of modification to the hypervisor. Moreover, NoHype removes the virtualization layer while retaining the key features enabled by virtualization. TrustOSV protects cloud environment by removing interactions between exposed executing environment and hypervisor.

Software at a Higher Privilege Level In order to mitigate the hazard caused by the hypervisor, plenty of software solutions propose and introduce a higher privilege-level than the original hypervisor. Nested virtualization is one of the representative approaches, which provides a higher-privileged and isolated execution environment to run the monitor securely. The turtles project [3] and CloudVisor [19] are examples of systems that propose nested virtualization idea to achieve isolation for protected resources. Especially, CloudVisor uses nested virtualization to decouple resource management into the nested hypervisor to protect VMs.

In practice, the independence on platforms and minimum changes to existing systems are the most prized features for cloud providers. For this purpose, some recent efforts introduce software-based approaches that achieve the same privilege-level isolation and protection instead of relying on a higher privilege level. Intel SGX, a hardware approach, is designed to provide protection for applications instead of the whole VM. AMD SEV tries to protect the memory of VM, however this feature is only available in the newly released hardware.

Inspired by the idea of ”same privilege-level” isolation, we propose HyperMI, a ”same privilege-level” and software-based VM protection approach for guest VMs against the compromised hypervisor. With HyperMI, the runtime data and VM memory isolation module are resided in a secure execution environment, referred to as HyperMI World. More details based on x86 are described as follows.

HyperMI protects interactions between hypervisor and VMs and achieves memory isoltaion among VMs. There are some especially critical structures data that records state information of VMs HyperMI should protect. These data includes Extended Page Tables (EPT) and Virtual Machine Control Structure (VMCS). EPT contains the mapping relationship from Guest Physical Address (GPA) to Host Physical Address (HPA). VMCS is used in VMX operation to manage the behavior of VMs as well as transitions between the VM and the hypervisor. Thus, the modified content of VMCS, especially the Guest-state area and VM-execution control fields, subverted by the compromised hypervisor may cause unpredictable consequences. Given the great importance of the data structures mentioned above, HyperMI isolates them beyond access from compromised hypervisor.

VM memory isolation can resist malicious VM memory accessing from compromised hypervisor, especially, 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 address translation function of the hypervisor to ensure that the page is marked with the owner when it is mapped. Finally, in order to avoid double mapping attack, the owner of the page is verified when EPT updates. In order to resist remapping attack, HyperMI clears the content of the page when it is released.

Our prototype introduces 4K SLOC (Source Lines of Code) to VM protection and 300 SLOC modifications to the hypervisor. The experimental results show trivial performance overhead and independence on multi-platforms for runtime VM protection.

Our contributions are as follows:

- An novel VM approach against compromised hypervisor with no requirement to extra hardware or higher privilege.
- A non-bypassable protection approach for VMCS and EPT which can ensure the security of interactions between hypervisor and VMs.
- A prototype based on KVM and x86 architecture with trivial performance overhead, high security, platform independence and fine applicability for cloud providers.

The rest of this paper is organized as follows. Section II discusses our threat model and assumption. Section III will elaborate on the design and implementation of HyperMI on x86 platform. Section IV gives the evaluation of security and performance. Section V compares HyperMI to previous work. At last, Section VI is the conclusion.

II. Threat Model and Assumption

A. *Threat Model*

We assume that the hypervisor has been compromised and controlled by the powerful adversary. The adversary can implement attacks based on the two attack paths. First, the adversary can subvert the critical interaction data in the context switching process between VMs and the hypervisor. Second, the adversary can tamper values in EPT entry. This can result in remapping attack and double mapping attack.

a) Modifying the Interaction Data: For the modification of the critical interaction data to the context switching, the attacker can obtain the address and modify VMCS, such as HOST_RIP, GUEST_CR0, EPTP, et al. For example, modifying the value of privilege register, CR0, closes DEP mechanism, and modifying CR4 can closes SMEP mechanism.

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

Scene 1. For double mapping attack, the attacker first controls and compromises a VM, then obtains the privilege of hypervisor through VM 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. Also, for a guest virtual address in VM2, named ’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 double mapping.

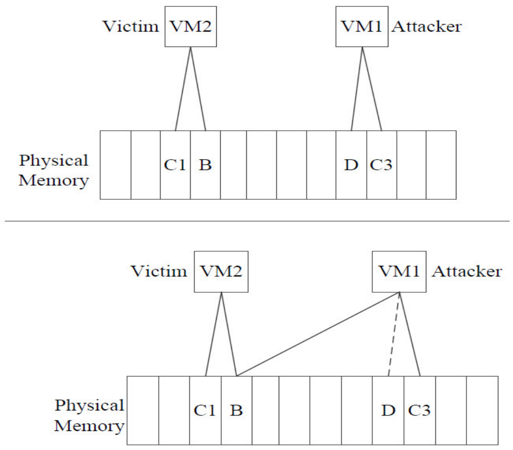

Fig. 1. The execution process of double mapping.

Scene 2. For the remapping attack, there are VM1 (attacker) and VM2 (victim). A physical page (named ’A’) used by VM2 is released after being used. After ’A’ is released, VM1 remaps to ’A’. So that the guest virtual address of VM1 points to the physical page ’A’. By this way, VM1 can access the information on ’A’ used by VM2, causing information leakage. Through the analysis of these two kinds of attack models, it is necessary to achieve attack prevention.

B. *Assumption*

We propose some assumptions. First, we assume hardware resources are trusted including processor, buses, BIOS, UEFI and so on, the trusted boot based on hardware can ensure the security and integrity of bootloaders. 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, such as cold-boot attacks and RowHammer.

III. Design and Implementation

A. *Architecture*

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

Figure 2 depicts the architecture of HyperMI. HyperMI creats two different address space based on two sets of page

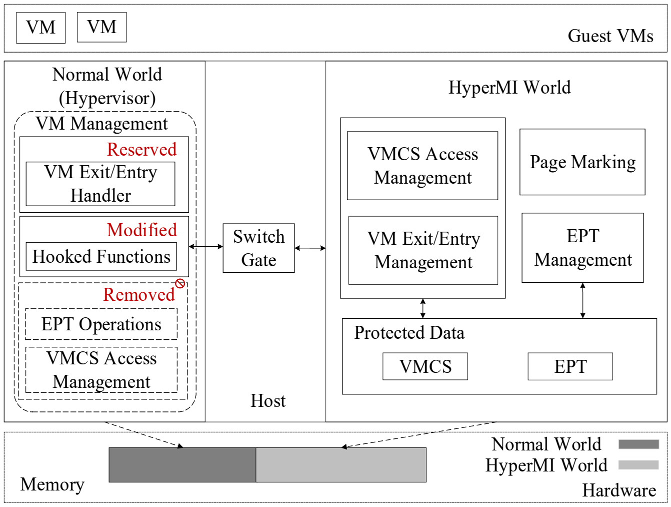

Fig. 2. The architecture of HyperMI.

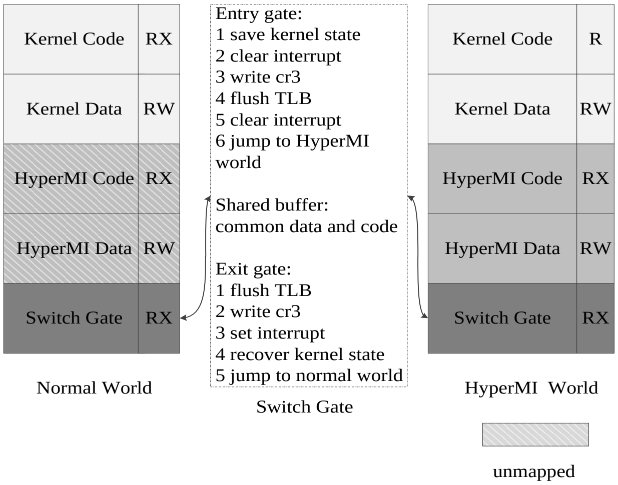

Fig. 3. An overview of address space layout.

tables. It is composed of three parts, modified hypervisor, HyperMI World and Switch Gate. The first component is the modified hypervisor, some functions such as EPT operations, VMCS access management, are removed from the original hypervisor. These functions which are closely related to protected data structures are hooked. Once they are called, the execution flow is transferred to HyperMI World through Switch Gate. The second component is HyperMI World, which is a secure and isolated execution environment based on the same privilege level with the hypervisor. VM Exit/Entry management and page marking, rely on this HyperMI World to ensure their security of runtime data and memory isolation. The last component is Switch Gate. This component is an atomic operation to ensure the secure switch between the normal world and HyperMI World. Because VMCS and EPT are hidden in HyperMI World, all access control flow for them must be traped to HyperMI World from modified hypervisor.

While the hypervisor together with guest VMs runs in the normal world, the hypervisor is forced to request HyperMI to perform four operations: 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 HyperMI, the whole system is ready to create an isolated executing environment. With these designs, HyperMI enforces the isolation and protection of memory used by each VM. Furthermore, HyperMI guarantees the security of interaction, memory isolation between the hypervisor and VMs.

B. HyperMI World

The creation of HyperMI World has two purposes: 1) Creating a space which can provide protection for key data and memory of VMs. 2) Creating a software system that does not depend on customized hardware devices and adapts to multi-system platforms. The key point of its design is that it creates another address space at the same privilege-level with hypervisor. Unlike other same privilege-level software, HyperMI World depends on another new page table and two different address space.

Creating HyperMI World We use two isolated address space based on two sets of page table to achieve isolation of HyperMI World. Figure 3 describes the address space layout of two worlds through two sets of page table, the normal page table and HyperMI page table. On the left of Figure 3, the normal page table contains code and data of the normal world except for that of HyperMI World in case of compromised hypervisor breaking the integrity of HyperMI World. So that program running in normal world cannot access data in HyperMI World. On the right of Figure 3, all address are mapped in HyperMI page table. HyperMI code remains executable and HyperMI data remains writable. What's the most important, kernel code is forbid to execute with reference to protection for running HyperMI World, so that non-executable kernel code cannot attack HyperMI World.

Creating Switch Gate In the middle of Figure 3, 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 for returning to the normal world. The 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 the HyperMI World and return address to the 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 a malicious attacker accessing HyperMI World causally after trusted boot. This is introduced in section III-D.

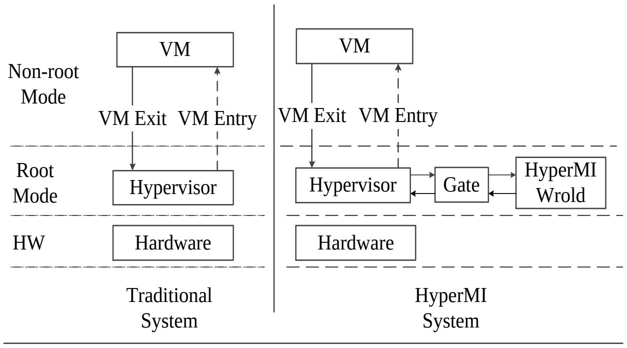

Fig. 4. Interaction comparision.

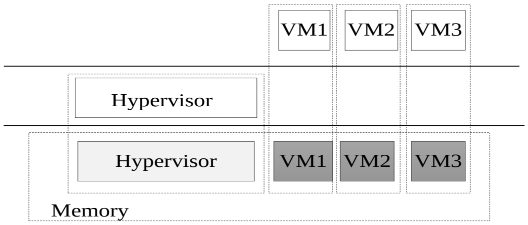

Fig. 5. Memory isolation for VMs.

C. VM Protection Approach

Interaction Monitoring VMCS and EPT are the most two important data structures that a hypervisor can utilize to interact with the VM. And these two data structures can only be accessed by hypervisor in traditional virtualization environment without HyperMI. If the hypervisor is compromised by an attacker, during the interval between VM Exit and VM Entry. Some attacks can be conducted to sebvert the VM.

1) The compromised hypervisor can illegally get the address of VMCS and modifies the content of VMCS directly. It can falsify the value of HOST_RIP and causes control flow hijack attack. 2)It can also supply the VM with a dedicated illegal EPT by tampering the value of EPT_POINTER (EPTP) of VMCS. 3) The compromised hypervisor can illegally modifies the content of EPT entries. Because the EPT is responsible for managing all physical memory access of VM. The compromised hypervisor can easily conduct remapping or double mapping attack to the VM. 4) The attacker can load EPT of any VM and access the VM's normal memory illegally.

Hence, we straightforwardly provide the protection for these data structures by using HyperMI World. These two data are hidden in HyperMI World in case of malicious access from hypevisor. In specific, at each time when VM exits to compromised Hypervisor, HyperMI catches these events and transfers VM_Exit to HyperMI World. All functions that modify VMCS and EPT entry are hooked into HyperMI World.

We hide VMCS in HyperMI World to avoid access from hypervisor. In order to ensure that some functions (vmcs_writel, vmcs_readl et al.) can access VMCS properly, HyperMI hooks these functions into HyperMI World. So hypervisor requests HyperMI World to handle operations about VMCS and return the corresponding result for the legal request.

Since VMCS is hidden in HyperMI World, all context management (accessing VMCS operations) must be trapped to hypervisor. During VM Exit, hypervisor needs to access VM Exit reason data of VMCS, and then deal with the exit event. Because hypervisor can not access VMCS, VM exit redirection is designed. Control flow jumps to HyperMI World to access VM exit reason data of VMCS structure, then switches to hypervisor and executes VM exit event handler function. VM Entry also accesses VMCS in HyperMI World. The control flow is shown as Figure 4.

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

EPT Protection Some functions, EPT creating, loading, walking and destroying, need access address of EPT. It can cause system suspend if they can not access the address of EPT. In order to ensure these functions can execute normally. HyperMI places hooks on these functions, then dispatches them to HyperMI World and handles 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.

If EPT isolation among VMs can not be ensure. A malicious VM can load other VM's EPT and access the memory data. It is important to ensure EPT isolation and one VM only access own corresponding EPT. To ensure one EPT for one VM, HyperMI creates the VM-Mark structure stored in HyperMI World as Table 1 described. It records VMID, EPTID, EPT_Address and binds them together. VMID is created when the VM is created. EPTID and EPT_Address is recorded as long as the EPT of current VM is created.

VM Memory Isolation Isolating memory is another aspect that should be considered. We need to ensure that VM and hypervisor can only access their own memory, as shown in Figure 5. Without memory isolation, a VM may suffer double mapping attack and remapping attack described in section II-A. We use Page-Mark structure described in Table 2 to record the owner and status of every page.

TABLE II
PAGE-MARK TABLE.

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

In order to go against the double mapping attack in the process of EPT updating, HyperMI should finish these two tasks: 1) It should verify the owner of pages when EPT update. 2) It should mark 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 catalogs: 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 clears the Page-Mark structure of the corresponding page.

D. Security Guarantee for HyperMI World

Nevertheless, without any protection measures, the page table to load to switch to HyperMI World is not secure for four reasons: 1)Attackers can control page table with the highest privilege after hypervisor is compromised. 2) Attackers can bypass the switch gate to break the security of HyperMI World. 3) Attackers with the highest privilege can free to execute privileged instructions to access the value of privilege registers, such as CR0, CR3 and so on. 4) Attackers can carry out DMA attack to access HyperMI World casually. We detail the protection measures for these four types of attacks below.

Protecting Page Table There are three reasons for controlling the two sets of kernel page tables: 1) To access casually or bypass HyperMI World, the attacker can tamper normal page table to map address of HyperMI World or load malicious page table to CR3. 2) The attacker can cover the hooked functions, redirect the functions to their own malicious code and bypass secure monitoring of HyperMI. 3) To break HyperMI World, malicious kernel code with execution permission can be executed to subvert HyperMI World.

For the first attack, we remove all entries that map to HyperMI World from the page table in normal world. Deprive the ability to access CR3 of the kernel.For the second attack, we intercept the accessing operation to CR0 and maintain the WP bit as 1. We stick to W$\oplus$X and maintain the code segment of hooked functions unwritable. For the third attack, we set the kernel code segment as NX (non-executable) when HyperMI World is running. For more security, we modify the kernel to configure these two sets of page table as read-only by setting the memory regions of the page tables unwriteable. This is necessary to prevent the page tables from being modified by attackers. Any write permission modification to two sets of page table must cause the kernel to page fault, then we dispatch page fault to HyperMI World to verify the correctness of address mapping.

Worlds Switching Securely HyperMI creates a switch gate between the normal world and HyperMI World by loading a page table of the next space into CR3. In order to avoid ensure switch security, we design the switch process as follows.

The switching process described in Figure 3 is as follows: 1) Save the kernel state to the stack including general registers and interrupt enable/disable status. 2) Clear the interrupt with the CLI instruction. 3) Load the page table to the register CR3. 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.

Accessing Privilege Registers Securely The hypervisor is pivileged and it can free to execute privilege instructions, so that it can write any value to the related privileged registers. 1) Malicious attackers can close DEP mechanism by writing CR0, close SMEP mechanism by writing CR4. 2) Kernel code can load a crafted page table to bypass HyperMI World by converting a meticulously constructed address of one page table to CR3. To prevent the attack, HyperMI deprives sensitive privileges instructions executed by the hypervisor, and dispatches captured events to the HyperMI World. HyperMI World can choose how to handle this event, such as issuing alerts, terminating the process, or doing nothing.

Resisting 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 crucial focus of intercepting direct access to physical pages belonging to HyperMI World 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) We 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) HyperMI intercepts the address mapping functions about I/O, verifies whether the address is an address space of HyperMI World, then chooses to map or unmap.

Through the above security measures, HyperMI can be protected from being bypassed and being breaking, thus providing a secure execution environment for VM protection.

IV. EVALUATION

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

A. Security Analysis

In section III, we discuss in detail how HyperMI achieves memory isolation, and secure interactions between VMs and the hypervisor. Just as the threat model described in the section II-A, an attacker could subvert the upper guest virtual machines by implementing attacks such as cross-domain attack with a malicious virtual machine. In this section, we will elaborate the security evaluation on how HyperMI achieves memory isolation among VMs through the monitoring to interaction-data. In addition, we analyzed the security of HyperMI itself. Table III shows the real attack instances in line with the above attack model.

Modifying Interaction-Data Attack We clarify interaction data including VMCS in Section III-C. To prevent interaction-data leakage, we protect VMCS from the attacker. Firstly, VMCS is hidden in HyperMI World and can not be accessed by hypervisor. Secondly, functions that can access VMCS location are hooked into HyperMI World, therefore, no funtion outside HyperMI World can access to VMCS. The attacker can not get location of VMCS, nor to access it, this prevents attackers from tampering interactive data attacks. We examine the security of protection to VMCS by conducting several attack cases which are widely adopted in real world. Table III lists all attack cases we used. The attack, named Interaction-data attack, tries to tamper the Guest_CR3 field in VMCS. The result of this attack failed because of the unaccessable to VMCS. According to the above analysis, the attacker can not access VMCS, and can not conduct further attack. Therefore, the VM runtime states and interaction-data can be prectected by HyperMI.

Subverting Memory Across VMs Attack The main attacks that attackers can execute on subverting memory are double mapping attack and remapping attack. Firstly, double mapping attack succeeds by assigning memory pages that have already been owned by a hostile VM to a victim VM. Page tracking and write-protection of EPT prevent this kind of attack. 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 Page-Mark 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 a page is released, its content must be zeroed out before creating a new mapping.

We implement a real attack, CVE-2017-8106 in kernel version 3.12. A privileged KVM guest OS user accesses EPT, conducts attacks via a single-context INVEPT instruction with a NULL EPT Pointer. Attackers can not implement successfully and incur EPT access fault because HyperMI hides the address of EPT in HyperMI World and hijacks the loading of EPT. Therefore, HyperMI can avoid subverting memory across VMs including double mapping attack, remapping attack as well as malicious EPT access.

Destroying HyperMI World HyperMI is created by relying on page tables. We analyze the protection of HyperMI World from four aspects, page table modifying attack, hooks redirection attack, reg attack and DMA attack.

1) Page Table Modifying Attack: Page table protection has been introduced in section III-D. The entry address mapping of the new page table is deleted from the old page table mapping to prevent the kernel from accessing HyperMI World directly through the page table mapping. When HyperMI World is active, the kernel code does not have any executable permissions in case of attacking running processes in HyperMI World. An attacker may attack in two ways. First, the attacker may try to directly access the new page table address on the kernel page table by virtual address mapping, but when he accesses it, there is page fault due to the absence of address mapping. Moreover, the attacker may run kernel code while the HyperMI World is active to attack programs running in the HyperMI World. This can be prevented because of the absence of executable privilege of kernel.

TABLE III
HYPERVISOR ATTACKS AGAINST HYPERMI.

Attack	*Description*
Interaction-data Attack	Load a crafted GUEST_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

2) Hooks Redirection Attack: Due to the code of hooked functions including VMCS operations, EPT operations and control register access operations is writable-protection, CR0 register access operation used to set W$\oplus$X is controlled and page table updating used to change code execution privilege is limited, the attacker can not redirect hooked functions and bypass being monitoring.

3) Reg Attack: Access to some registers access operations including CR0, CR3, CR4, are controlled and hooked to HyperMI World. CR0 register can control the W$\oplus$X privilege of code, CR3 can control the loading of the page table and CR4 can decide SMEP mechanism. Protection for page table, hooked functions and regs play a role mutually in protection for HyperMI.

4) DMA Attack: DMA attack is described in detail in section III-D. Attackers can use this feature to read or corrupt arbitrary memory regions. DMA attack is not a threat to HyperMI, because HyperMI is inherently secure against DMA using IOMMU. 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. DMA attack that aims at modifying the VM memory or the page tables will also be defeated.

B. Performance Evaluation

In HyperMI, the kernel is modified so that HyperMI World is initialized during the bootup sequence. This includes creating a new memory page table for HyperMI, allocating memory pages, as well as creating Page-Mark table and VM-Mark table. This process introduces security verification for pages according to Page-Mark table, and security accessing for VMCS in HyperMI World during VM Exit/Entry sequence.

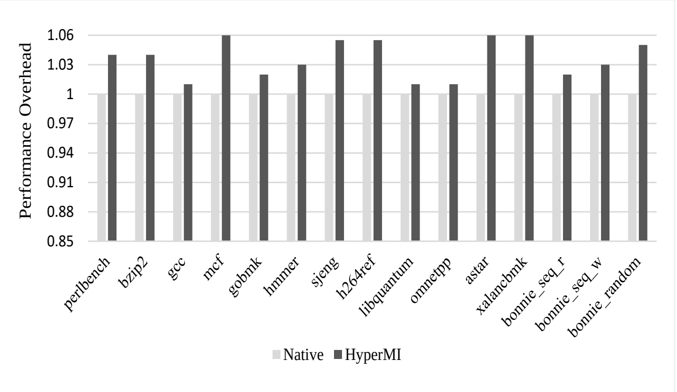

Fig. 6. Performance overhead.

The kernel is modified to place hooks upon some functions. introducing worlds switching overhead using switch gate.

In order to assess the effectiveness of all aspects of HyperMI, we conduct a set of experiments to evaluate the performance impact imposed by HyperMI against an original KVM system (the baseline). We run three groups of experiments and compare the performance overhead, benchmarks performance overhead, VM load time as well as VM Exit/Entry overhead. For simplicity, we only present the performance evaluation on a server with 64 cores and 32 GB memory, running at 2.0 GHz and guest VM with 2 virtual cores. The version of the hypervisor and guest VM is 3.10.1. Different experiments are based on different numbers of guest VMs with different memory size. Both the original and HyperMI systems have the same configuration except the protection supported by HyperMI. The deviation of these experiments is insignificant. All the experiments are replicated fifty times and the average results are reported here.

Benchmarks Performance In order to obtain the impact of HyperMI on the whole system, we measure HyperMI with microbenchmarks and application benchmarks. We use one guest VM with 2 GB memory size.

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. The experiments result described in Figure 6(the last three groups) shows a 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 and verify the legality of page mapping when EPT updates. This can incur worlds switching which involves controlling register access and incur VM exit which involves page fault of EPT. For Bonnie++, we choose a 1000 MB file to perform the sequential read, write and random access. The performance loss of sequential read, write and random access is 2%, 3% and 5%, not high, the main reason is that HyperMI has no extra memory operations for I/O data. The performance result shows that HyperMI introduces trivial switch overhead of two worlds and trivial overhead of memory isolation of VMs.

TABLE IV
EXECUTION TIME OF VM OPERATION(S).

Test Case	***VM Create***	***VM Destroy***
No_HyperMI	11.79 s	1.75 s
With_HyperMI	12.97 s	1.89 s
efficiency	1.1	1.08

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

V. Related work

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

A. Customized 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 [9], [12] 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. Vigilare [13] and KI-Mon [11] 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.

B. Reconstructed Hypervisor

Except for approaches based on hardware, some works [14], [15], [18] 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 [15] 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 [18] prepares backup KVM for every

VM by copying KVM code, and ensures every VM run in own isolated space. 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, SKEE [1] and SecPod [16], [7], adopt the same privilege-level idea to avoid performance overhead of inter-level translation. SKEE provides a lightweight secure kernel-level execution environment, this environment is placed at the same privilege level and the same address space with kernel. SKEE is exclusively designed for commodity ARM platforms using system characteristics of ARM. The difference between HyperMI and SKEE is that HyperMI uses two sets of page tables to create the execution environment, and SKEE uses one set of page tables. The design of the switch gate for HyperMI and SKEE are also different. SKEE is more focused on using the characteristic of the ARM platform while HyperMI has no dependence on the platforms. SecPod, an extensible approach for virtualization-based security systems that can provide both strong isolation and the compatibility with modern hardware. The biggest difference between SecPod and HyperMI is that SecPod creates the secure isolation environment for every VM. SecPod solves the problem of virtual machine address mapping with the assistance of shadow page table (SPT) technology.

We neither adopt software at a higher level than the hypervisor, nor use customized hardware. Inspired by the same privilege-level, we propose HyperMI World placed at the same privilege-level with hypervisor.

VI. 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 VMs, 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, portability, and independence on multi-platforms. 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] 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)

[8] 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)

[9] 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)

[10] 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)

[11] 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)

[12] 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)

[13] 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)

[14] 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)

[15] 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)

[16] 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)

[17] 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)

[18] 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)

[19] 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)