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

***Abstract*—Once compromising the hypervisor, remote or local adversaries can easily access other customers' sensitive data in the memory and context of guest virtual machines (VMs). Therefore, it is essential to protect VMs, especially the context and the memory of VMs from the compromised hypervisor. However, previous efforts employ extra customized hardware which is not convenient for cloud providers to adopt widely. Or they employ a new software relying on higher privilege level than hypervisor.**

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 VM 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 the hypervisor possesses the highest privilege in the cloud environment, attackers who compromise hypervisor can 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 the hypervisor.

Hardware Different hardware platforms provide different hardware mechanisms,such as Intel's Software Guard Extensions (SGX) and AMD's Secure Memory Encryption (SME) [16]. SGX provides protection for pieces of application logic inside encrypted enclave memory against malicious OS. However, it is limited to protect a relatively small portion of memory, and the developers have to mostly reconstruct the protected software or build it from scratch. Therefore it is nontrivial for SGX to protect large-scale software like the entire VM. SME can encrypt memory in page level granularity by simply setting the C-bit in the page table entry. More importantly, AMD enables the Secure Encrypted Virtualization (SEV) feature, so that each virtual machine can use its own key to selectively encrypt memory. However, we observed that there are numerous issues when leveraging SEV for VM protection against an untrusted hypervisor. First, a malicious hypervisor can bypass the protection by manipulating some critical resources. For example, the hypervisor is in the position of managing the guest memory mapping and key sharing mechanisms, where replay attacks can be conducted to infer VM's encrypted memory, and the shared keys may be abused by the hypervisor with other collusive VM.

Reconstructed Hypervisor Some efforts (NoHype [7] and TrustOSV [14]) 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 of 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 [17] 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.

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, interaction data protection and VM memory isolation modules are resided in a secure execution environment, referred to as HyperMI World. We implement this prototype on x86 as an example, more details on HyperMI are described as follows.

HyperMI protects interactions between hypervisor and VM and achieves memory isolation among VMs. There are some especially critical interaction structure data that records state information of VMs which 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 VM and the hypervisor. Thus, the modification of field of VMCS, especially the Guest-state area and VM-execution control fields, may cause unpredictable consequences. Given the great importance of the data structures mentioned above, HyperMI isolates them in HyperMI World and 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, since any update to guest page table can be synced to EPT, HyperMI verifies whether any protected virtual addresses is double mapped or remapped. 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 of the hypervisor. The experimental results show trivial performance overhead and independence on multi-platforms for runtime VM protection.

Our contributions are as follows:

- A novel VM protection approach against compromised hypervisor with no requirement for extra hardware or a higher privilege-level software.
- A non-bypassable protection approach for VMCS and EPT which can ensure the security of interactions between VM and hypervisor.
- An approach which can isolate memory among VMs and hypervisor securely by using page marking technique.
- 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 elaborates on the design and implementation of HyperMI on x86 platform. Section IV gives the evaluation of security and performance. Section V compares HyperMI with previous work. At last, Section VI gives 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 two attack paths. First, the adversary can subvert the critical interaction data during the context switching process between VM and the hypervisor. Second, the adversary can tamper values of EPT entry. This can result in remapping attack and double mapping attack.

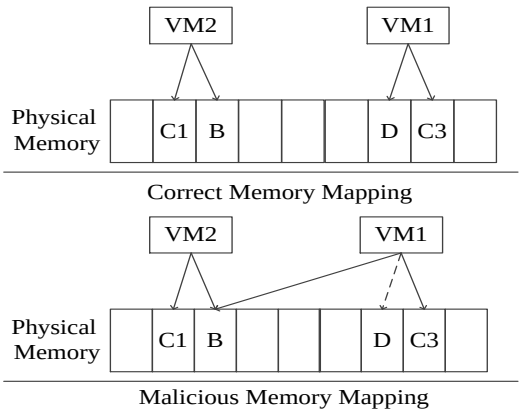

Fig. 1. The execution process of double mapping.

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

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

Scene 1. For double mapping attack, attackers control and compromise a VM, then obtain the privilege of hypervisor through VM escape attack, and maliciously access the VMCS structure to obtain the value of EPT_Pointer(EPTP). The attack process is as shown in Figure 1. There are two guest VMs, VM1 (attacker) and VM2 (victim). In this way, EPTP of VM1 and EPTP of VM2 are respectively obtained by attackers. Also, for a guest virtual address (GVA) in VM2, named 'A', is mapped to the corresponding real physical address, named '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. At last, 'A' and 'C' are mapped to the real physical address 'B', no address is mapped to 'D'. Then VM1 can access the data of VM2 successfully through accessing 'B', this attack process is called double mapping attack.

Scene 2. For the remapping attack, there are VM1 (attacker) and VM2 (victim). A physical page (named 'A') used by VM2 is released after it is used and it is released with VM2's content. After 'A' is released, VM1 remaps a GVA (named 'E') to 'A'. By this way, VM1 can access the content on 'A' used by VM2 through 'E', causing information leakage. Through the analysis of these two kinds of attack models, it is necessary to achieve attack prevention.

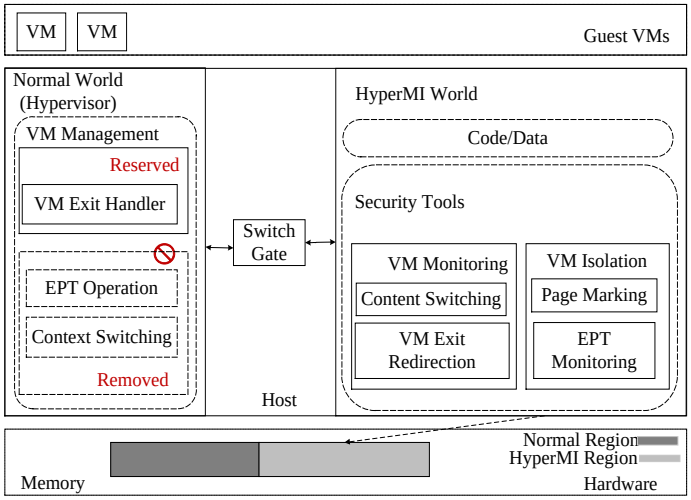

Fig. 2. The architecture of HyperMI.

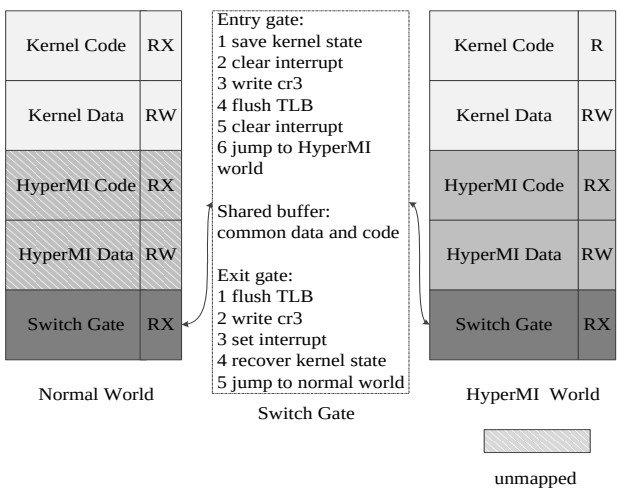

Fig. 3. An overview of address space layout.

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 attack and RowHammer. Third, we assume that there exists no software that runs at a higher privilege level than the hypervisor.

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 the hypervisor or extra customized hardware.

Figure 2 depicts the architecture of HyperMI. HyperMI creates another different address space based on a set of page table. It is composed of three parts, Modified Hypervisor, HyperMI World and Switch Gate. The first component is Modified Hypervisor. Some components such as EPT operations, VMCS access management, are removed from the original hypervisor. HyperMI replaces these functions which are closely related to interaction data with hooks so as to verify and execute them in HyperMI World. 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 placed at the same privilege level with the hypervisor. VM Exit/Entry management, EPT management and Page Marking, these components rely on HyperMI World to ensure their interaction data security and memory isolation. The last component is Switch Gate. It 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 to them must be trapped to HyperMI World from Modified Hypervisor through Switch Gate.

While the hypervisor together with guest VMs runs in the normal world, the hypervisor is forced to request HyperMI to perform four operations: 1) monitoring interaction-data (context switching) between the hypervisor and VM, 2) updating EPT of VMs, 3) verifying the pages when executing EPT update operations to resist double mapping attack, 4) verifying the pages when executing releasing operations to resist remapping attack. After starting HyperMI, the whole system is ready to create an isolated executing environment. With these designs, HyperMI guarantees the security of interaction between the hypervisor and VM as well as memory isolation among VMs.

B. HyperMI World

The creation of HyperMI World has two purposes: 1) creating a address space which can provide protection for interaction 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.

Creating HyperMI World We use two isolated address spaces based on two sets of page tables 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. Figure 3 describes normal world on the left and HyperMI World on the right. By choosing the different valid address space at runtime, HyperMI blocks or allows access to the memory region of HyperMI World depending on which of the two worlds is currently in control. When the normal world seizes execution control, HyperMI leaves memory region of HyperMI World unmapped, thereby preventing HyperMI World from being

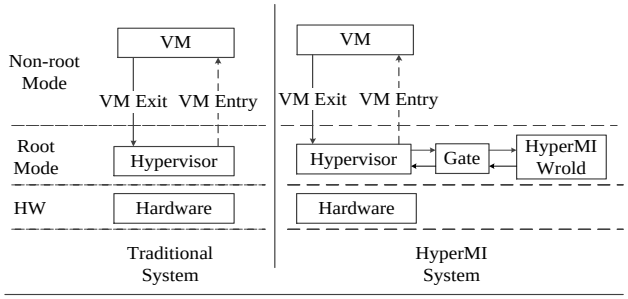

Fig. 4. Interaction comparision.

exposed and possibly exploited. On the other hand, while HyperMI World holds execution control, the valid virtual address range covers both HyperMI World and normal world, implying that HyperMI World owns a full access coverage reaching the entire memory space. Another difference between two worlds is that kernel code in different worlds has different privilege. What's the most important, kernel code is forbid to execute when HyperMI World is active, so that it can not attack HyperMI World.

Creating Switch Gate HyperMI designs and implements Switch Gate to perform worlds switching by executing a series of ordinary instructions. Switch Gate performs the control switching operation between the inner and outer domains, acting as a wrapper function for a handler which processes incoming requests in HyperMI World. It provides the normal world with a unique way to enter HyperMI World. In addition, Switch Gate is implanted across the normal world and is invoked with specific parameters in order to handle sensitive resources including control register, VMCS and EPT, by sending relevant requests to HyperMI World.

Figure 3 describes the details of Switch Gate and it is divided into the entry/exit gates and shared buffer. Entry gate provides the only entrance to HyperMI World and the exit gate provides the address for returning to the normal world. The shared buffer contains common data and code in the normal world and HyperMI World. Common code is switch code and common data includes the entry address of two sets of page tables. Of course, the entrance address must be protected after switching to HyperMI World in case that attackers access HyperMI World causally after trusted boot. This is introduced in section III-D.

C. VM Protection Approach

Interaction Monitoring VMCS and EPT are the most two important data structures that the hypervisor can utilize to interact with VM. VMCS, the architectural in-memory structure, is used to manage the mode transitions as well as specify the restricted operations causing the VM Exit/Entry. EPT contains the mapping relationship from GPA to HPA. If the hypervisor is compromised by attackers, during the interval between VM Exit/Entry, some attacks can be conducted to subvert VM. 1) The compromised hypervisor can illegally get the address of VMCS and modify the content of VMCS directly. It can falsify the HOST_RIP field causing control flow hijack attack or tampering the EPTP filed with a dedicated illegal EPT. 2) The compromised hypervisor can illegally modify the content of EPT entries. Because the EPT is responsible for managing all physical memory mapping of VM. The compromised hypervisor can easily conduct remapping or double mapping attack to the VM. 3) Attackers can load EPT of any VM and access the VM's normal memory illegally.

Obviously, the direct purpose of the attacker is to get the addresses of these two data structures. 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 hypervisor. In order to ensure that some functions can access VMCS and EPT properly, HyperMI hooks these functions into HyperMI World. So hypervisor requests HyperMI to handle operations and return the corresponding result for the legal request.

Hardware virtualization includes a set of privileged instructions (vmptrld, vmptrst, vmclear, vmread, vmwrite) to load, store, clear, read and write the current VMCS. HyperMI intercepts the execution of these privileged instructions by eliminating them from hypervisor's code, so that any manipulation on the VMCS can be validated by HyperMI. This is what VMCS access management in Figure 2 does.

During VM Exit, hypervisor needs to access VM Exit reason data of VMCS, and then deals with the VM exit event. Because hypervisor absents the ability to access VMCS directly, VM exit redirection is designed and finishes VM Exit process. After accessing VM exit reason data of VMCS in HyperMI World, HyperMI designs the control flow to switch to hypervisor and execute VM exit event handler functions. 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 to access address of EPT. In order to ensure these functions can execute normally and intercept EPT updates, HyperMI places hooks on these functions, then dispatches them to HyperMI World and handles appropriately. In the meantime, HyperMI avoids double mapping attack to ensure that there is only one virtual address mapping to one physical memory page during EPT updating, and handles remapping problems to ensure the content of page cleaned after page is swapped out. This will be described in detail later.

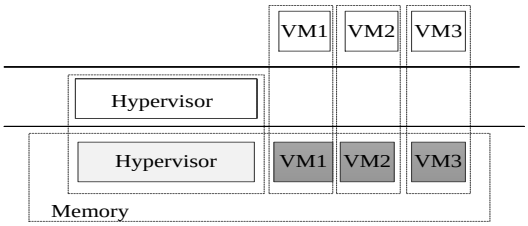

Fig. 5. Memory isolation for VMs.

TABLE II
PAGE-MARK TABLE.

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

If EPT isolation among VMs can not be guaranteed, 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 I 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 the hypervisor can only access their own corresponding 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 II to record the owner and status of every page.

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 updates. 2) It should mark the OwnerID field of Page-Mark structure for unused pages or thwart the mapping operation for used pages in case of malicious double mapping behavior. So page marking technique 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 content of the page when the page is swapped out, so attackers can't get the content of the page by the way of remapping. HyperMI clears the Page-Mark structure of the corresponding page.

D. Security Guarantee for HyperMI World

The security of HyperMI World guarantees the security of HyperMI, because HyperMI relies on HyperMI World to provide a secure execution environment. Nevertheless, without any protection measures, the normal world kernel page table is not secure for four reasons: 1)Attackers can control page table with the highest privilege after compromised hypervisor is . 2) Attackers can bypass 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. 4) Attackers can carry out DMA attack to access HyperMI World casually. We detail the protection measures below for these four types of attacks.

Protecting Page Table To maintain the validity of HyperMI World, HyperMI must ensure that the two sets of page tables are prevented from the following attacks: 1) To access casually or bypass HyperMI World, attackers can tamper normal page table to map address of HyperMI World or load malicious page table to CR3. 2) Attackers can cover the hooked functions, redirect the functions to malicious code and bypass interaction monitoring of HyperMI. 3) To break HyperMI World, malicious kernel code with execution permission can be executed to subvert HyperMI.

For the first attack, we remove all entries that map to memory region of HyperMI World from the page table in normal world. Deprive the hypervisor the ability to access CR3. 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, HyperMI allows only HyperMI World to conduct page table modifications after verifying that each modification avoid the above attacks. It initially configures page tables as read-only to prevent the normal world from modifying them. In addition, it instruments the normal world code to route all page table modification operations to page fault handled by HyperMI World.

Worlds Switching Securely HyperMI creates Switch Gate between the normal world and HyperMI World by loading entry address of page table into CR3. Switch Gate disables interrupts at the entry gate to ensure the atomicity of the gates. It may harden the security of HyperMI World by preventing control from being intercepted by the normal world when the control is in the gates or in HyperMI World. In order to 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 World. For the exit process, the control flow returns to the normal world by performing the operations in the reverse order.

Accessing Privilege Registers Securely Even if the integrity of the page tables is preserved, HyperMI World can be incapacitated through exploiting system control registers. The hypervisor without HyperMI is privileged and it can free to execute privileged instructions by writing any value to the related privileged registers. 1) Malicious attackers can close DEP mechanism by writing CR0, close SMEP mechanism by writing CR4. 2) Attackers can load a crafted page table to bypass HyperMI World by converting a meticulously constructed

address of one page table to CR3. To prevent these attacks, HyperMI deprives sensitive privilege instructions executed by the hypervisor, and dispatches captured events to HyperMI World. HyperMI replaces operations that control such privilege registers in the normal world with hooks so as to verify, then execute or terminate the process them in HyperMI World.

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, Page-Mark structure, VM-Mark structure and so on. 2) HyperMI intercepts the address mapping functions about I/O, verifies whether the address belongs to 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 broken, 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 evaluate 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 VM and the hypervisor. Just as the threat model described in the section II-A, attackers can subvert the upper VMs by conducting attacks such as cross-domain attack with a malicious virtual machine. In this section, we elaborate the security evaluation on how HyperMI achieves memory isolation among VMs through the monitoring interaction data. In addition, we analyze 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 in Section III-C. We take VMCS as a example to describe security analysis on modifying interaction-data attack. To prevent interaction-data leakage, we protect VMCS from attackers. Firstly, VMCS is hidden in HyperMI World and can not be accessed by hypervisor. Secondly, functions that can access VMCS are hooked into HyperMI World, therefore, no functions outside HyperMI World can access VMCS. Attackers can not get location of VMCS and access it. This prevents attackers from tampering interaction data attacks. We examine protection for 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. This attack fails because it can not access VMCS. According to the above analysis, attackers can not access VMCS, and can not conduct further attacks. Therefore, the VM interaction data can be protected by HyperMI.

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

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 allocating memory pages that have already been owned by a hostile VM to a victim VM. Page marking 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 according to Used field of Page-Mark structure. 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 another 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. HyperMI verifies the value of EPT to avoid load NULL value. 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 modification attack, hooks redirection attack, reg modification attack and DMA attack.

1) Page Table Modification Attack: Page table protection has been introduced in section III-D. The entry address mapping of the HyperMI World page table is deleted from the old page table 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. Attackers may execute attacks in two ways. First, attackers may try to directly access the new page table address on the kernel page table by virtual address mapping. When he accesses it, there is page fault due to the absence of address mapping. Second, attackers may run kernel code while HyperMI World is active to attack programs

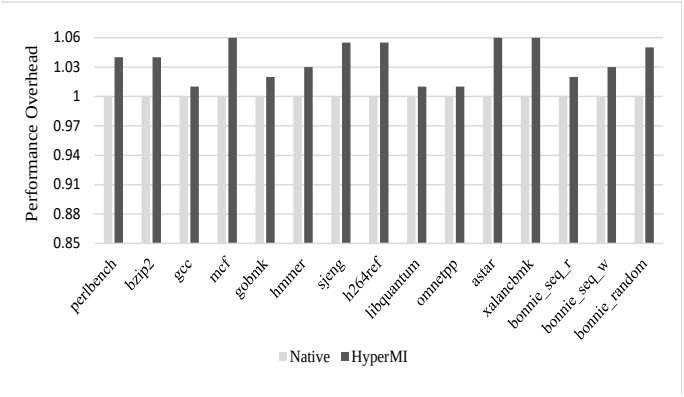

Fig. 6. Performance overhead.

running in HyperMI World. This can be prevented because of the absence of executable privilege of kernel code.

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

3) Reg Modification Attack: Some registers access operations including CR0, CR3, CR4, are controlled and hooked to HyperMI World. CR0 register can control the W⊕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, plays 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. HyperMI removes the corresponding mapping of the critical data from the page table which IOMMU uses and intercepts DMA mapping. These critical unmapped data includes the entrance address of HyperMI, data recording Page-Mark structure used in VM isolation, VM-Mark structure and so on. DMA attack that aims at modifying the VM memory or the page tables can also be defeated.

B. Performance Evaluation

In HyperMI, the hypervisor is modified so that HyperMI World is initialized during the boot up 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. 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 two groups of experiments, compare the performance overhead including benchmarks performance overhead and VM load time. 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 hypervisor 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.

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

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 1 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 experiment 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. HyperMI handles Page-Mark structure and verifies the legality of page mapping when EPT updates. This can incur worlds switching which involves controlling register access and incur VM exit which involves EPT updating. 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.

VM Load Time The load time of a VM is a critical aspect of performance to be considered because it influences user experience. We design experiments to evaluate the performance impact of HyperMI for VM loading. We measure the impact of completely booting and shutdowning 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 in HyperMI World. 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 [2], [8], [10], [4], [6], [9], at 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 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. It has requirement for version of CPU and is applied on a few platforms. Vigilare [10] and KI-Mon [8] 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 [11], [12], [15] pay attention to software isolation. Pre-allocating physical resource and completed isolated environment for every VM can avoid VM cross-domain attack, and data leakage attack. NOVA [12] 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 [15] 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 of hypervisor.

C. The Same Privilege Level Isolation

Some efforts, SKEE [1] and [13], [5], adopt the same privilege-level idea to avoid performance overhead of inter-level translation. SKEE provides a lightweight secure kernel-level execution environment, it is placed at the same privilege-level with kernel. SKEE is exclusively designed for commodity ARM platforms using system characteristics of ARM. SKEE can only be applied to limited levels of system software in comparison with HyperMI. First, targeting ARM's 32-bit architecture, SKEE capitalizes mainly on Translation Table Base Control Register (TTBCR) for dynamic page table activation. To be more specific, SKEE creates separate page tables for the secure world and activates it in a timely manner by modifying the N field of TTBCR. However, as this hardware feature is only defined in the kernel privilege level on AArch32, SKEE is not commonly applicable to different levels of system software, such as hypervisors. Unfortunately, SKEE faces a similar limitation on ARM's 64-bit architecture. SKEE is more focused on using the features of the ARM platform while HyperMI has no dependence on multi-platforms.

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. HyperMI is independent on multi-platforms and practical for cloud providers.

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 event-driven runtime monitoring for interaction between hypervisor and VM. 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 and HyperMI itself, 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] 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)

[5] 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)

[6] 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)

[7] 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)

[8] 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)

[9] 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)

[10] 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)

[11] 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)

[12] 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)
[13] 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)
[14] 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)
[15] 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)
[16] Wu, Y., Liu, Y., Liu, R., Chen, H., Zang, B., Guan, H.: Comprehensive VM protection against untrusted hypervisor through retrofitted AMD memory encryption. In: IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018. pp. 441–453 (2018). https://doi.org/10.1109/HPCA.2018.00045, https://doi.org/10.1109/HPCA.2018.00045
[17] 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)