Dancing with Wolves: Towards Practical Event-driven VMM Monitoring

Liang Deng
State Key Laboratory for Novel Software Technology, Nanjing University, China
Sangfor Technologies Inc.

Peng Liu Jun Xu Ping Chen
College of Information Sciences and Technology, Pennsylvania State University

Qingkai Zeng
State Key Laboratory for Novel Software Technology, Nanjing University, China
zqk@nju.edu.cn

Abstract

This paper presents a novel framework that enables practical event-driven monitoring for untrusted virtual machine monitors (VMMs) in cloud computing. Unlike previous approaches for VMM monitoring, our framework neither relies on a higher privilege level nor requires any special hardware support. Instead, we place the trusted monitor *at the same privilege level* and *in the same address space* with the untrusted VMM to achieve superior efficiency, while proposing a unique *mutual-protection* mechanism to ensure the integrity of the monitor. Our security analysis demonstrates that our framework can provide high-assurance for event-driven VMM monitoring, even if the highest-privilege VMM is fully compromised. The experimental results show that our framework only incurs trivial performance overhead for enforcing event-driven monitoring policies, exhibiting tremendous performance improvement on previous approaches.

1. Introduction

During the past decade, virtual machine (VM) technology has generated great impact and is playing a critical role in cloud computing. For example, Amazon's Elastic Compute Cloud (EC2) platform [1] provides resizable computing resources in the form of millions of VMs managed by Xen-based VMMs. However, driven by various economic incentives, the sizes of commodity VMMs have reached hundreds of thousands of lines of code (i.e., Xen currently includes around 300,000 lines of code). The large code bases involve many vulnerabilities that could be exploited by adversaries, and some of them (e.g., CVE-2015-7835 [3]) have been shown to be high-risk vulnerabilities that can lead to fully compromised VMMs in Amazon's EC2 and other cloud computing platforms.

Since VMMs are granted with the highest privilege, attackers who compromise VMMs could jeopardize the whole cloud infrastructure and endanger any data and computation in the cloud. To mitigate the threats rooted from compromised VMMs, the first line of efforts should be VMM monitoring.

VMM monitoring introduces an external monitor, which runs in a secure environment as the root of trust, to assess VMM's integrity status at real time based on predefined monitoring policies. Since polling the VMM's integrity status based on a schedule cannot stop the attackers from launching transient attacks which first cause damages and then hide their traces during the polling interval, fully-trusted VMM monitors must do event-driven monitoring (or active monitoring) instead of polling (or passive monitoring). Event-driven monitoring triggers the monitoring round on the occurrences of designated VMM events. For example, in VMM data integrity monitoring, an event can signify the occurrence of each access of the monitored data in VMM. Therefore, whenever the VMM tries to access the monitored data for computation, the monitor can be timely invoked for integrity assessment. With this event-driven monitoring, there is no time dependency that the attackers can exploit to launch transient attacks.

Previous approaches, including HyperCheck [29] and HyperSentry [6], realize polling-based VMM monitoring based on system management mode (SMM). However, the nature of SMM (e.g., SMM cannot provide interception on MMU configuration) makes these approaches unable to realize event-driven monitoring.

MGuard [23], Vigilare [26] and KI-Mon [19] achieve trusted event-driven monitoring relying on customization of the underlying hardware, in particular, modifications to the processor and memory controller. However, hardware cus-

VEE '17, April 08-09, 2017, Xi'an, China

DOI: http://dx.doi.org/10.1145/3050748.3050750

tomization, which is far more expensive than software-level enhancement, may be unattractive to many cloud providers. To our best knowledge, none of the hardware customization approaches proposed in [19, 23, 26] have been widely deployed in real-world cloud computing.

Without customizing commodity hardware, nested virtualization [9] is the candidate approach that can be deployed for event-driven VMM monitoring in the literature. It introduces a higher privilege software layer (host hypervisor) to securely run the monitor and relies on nested virtualization functionalities to capture critical VMM events. However, the overhead of inter-privilege transition between the VMM and the monitor for each monitor invocation makes this approach unsuitable for event-driven monitoring. As discussed earlier, in order to effectively monitor VMM's data integrity, the monitor needs to be timely and frequently invoked in an event-driven manner (e.g., whenever the monitored data is accessed). This introduces a large amount of privilege switches at runtime and thus makes the system performance unacceptable. As shown in our experiment in Section 7, to monitor only a subset of VMM's critical data objects with nested virtualization, the measured performance overhead is over 300%.

To address the high overhead of inter-privilege transition, some recent efforts focus on how to achieve "same-privilege" isolation and protection without relying on a higher privilege level. For example, Virtual Ghost [12] and KCoFi [11] introduce a software TCB at the same privilege level as the untrusted kernel and rely on compiler techniques to secure the TCB. They in principle could be applied to monitoring a VMM by placing the TCB (monitor) at the same privilege level as the VMM. However, these approaches require special compiler support and considerable modifications to VMM's source code, which are impractical to cloud providers. Additionally, since these approaches are originally designed for OS kernels, they cannot address the security threats rooted from a compromised VMM. For example, a VMM generally enables hardware virtualization which introduces some new ways to compromise the VMM (more details are discussed Section 3.1.5).

Nested Kernel [13] is another same-privilege protection approach that leverages x86's write protection bit in the cr0 register to secure the same-privilege TCB, and could be also applied to VMM monitoring. However, even if the monitor and the VMM share the same privilege level in this approach, the transition between them still requires to execute the expensive mov-to-cr0 instruction for the isolation purpose. As shown in our experiment in Section 7, the overhead of mov-to-cr0 is still in the same order of magnitude with that of the inter-privilege transition (the switching between VMM and host hypervisor). Therefore, this approach is still unsuitable for event-driven VMM monitoring which by nature requires frequent monitor invocation.

To make event-driven VMM monitoring sufficiently efficient, we need not only to place the monitor at the same privilege level as the untrusted VMM, but also to avoid any expensive transition boundary (e.g., mov-to-cr0 execution, address space switching) between them. However, this obviously brings huge threats to the monitoring scheme. To make the monitor reliably work (*carefully dance*) with the untrusted VMM (*the wolf*), one must ensure the capture of designated VMM events and the complete integrity of the monitor. As we will shortly explain later in Section 2, performing this dance with such a strong wolf is un-trivially challenging.

In this paper, we present a novel event-driven VMM monitoring framework on mainstream commodity x86 hardware platforms. Our *key insight* is that through specific mutual-protection, "Dancing with wolves" is actually possible! In our framework, the event-driven monitor (called ED-monitor) is mounted at the same privilege level and in the same address space as the VMM, and there is no software that runs at a higher privilege level than the VMM. Our framework achieves superior efficiency by not requiring any expensive boundary crossing when switching to ED-monitor for a captured VMM event. To secure ED-monitor and its event-driven mechanism from the VMM, we propose a novel solution which relies on the mutual-protection of a unique pair of the techniques: Instrumentation-based Privilege Restriction (IPR) and Address Space Randomization (ASR). At the high level, IPR intercepts the most privileged operations in the VMM and transfers these operations to ED-monitor, while ASR hides ED-monitor in the address space from the VMM. ASR prevents IPR from being evaded and in turn, IPR protects ASR from being disabled. More details of the mutual-protection are presented in Section 2 and Section 3.

Our main contributions are as follows:

- We propose the first practical event-driven VMM monitoring system without any hardware customization.
- We propose a novel mutual-protection of IPR and ASR to secure ED-monitor that runs at the same privilege level and in the same address space with the VMM.
- We have implemented a prototype of our framework on the top of KVM architecture. The implementation does not require any modification to VMM's source code. The experimental results show that our framework only introduces trivial overhead for event-driven monitoring, exhibiting tremendous performance improvement on previous approaches.

2. Overview

2.1 Threat Model

We deal with compromised VMMs. Although cloud providers do not intend to be malicious, the threats come from the bugs and vulnerabilities within VMMs. Since the VMM is the

highest privilege software, attackers who compromise the VMM could manipulate the underlying hardware state (e.g., MMU configuration, interrupt handling, system registers), execute arbitrary code and access arbitrary data in memory or on disk. We assume that there exists no software that runs at a higher privilege level than the VMM.

Physical attacks are not considered. We assume that the CPU, memory controller, I/O peripheral devices and firmware are not malicious. That is to say, they operate exactly following their specifications and do not perform unintended operations. In addition, we assume the existence of a trusted booting mechanism (e.g., Intel TXT [16]).

2.2 Event-driven VMM Monitoring

To tackle the threats from compromised VMMs, event-driven VMM monitoring invokes the integrity assessment routines in the monitor upon the occurrences of designated VMM events. A common approach to capture designated events is to place hooks into the VMM [27]. These hooks can be code hooks (jump instructions) inserted at arbitrary locations in VMM code, data hooks within call tables, or any other technique that can transfer the control flow. When the VMM's execution reaches a hook, it will transfer the control flow to the monitor. By implanting hooks at the code positions where VMM events occur, the monitor can capture the occurrences of designated events to assess the VMM's integrity status.

2.3 Placing the Monitor in the Same World

Despite of the high importance of event-driven VMM monitoring for secure cloud computing, there practically lacks approaches for monitoring a VMM in an event-driven manner. In this work, we propose the first practical event-driven VMM monitoring framework.

Figure 1 shows an overview of our framework based on both KVM and Xen architectures. The VMM is executed at the highest privilege level (ring 0, root mode) to manage guest VMs (in non-root mode) as traditional. Instead of introducing a higher privilege level, we place our event-driven monitor (ED-monitor) at the same privilege level and in the same address space as the untrusted VMM. In this way, the VMM and ED-monitor run in the same world; The switching between the VMM and ED-monitor is very lightweight as no expensive boundary crossing (privilege level change or address space switch) is required.

To enable the event-driven feature, the hooks are implanted into the VMM to redirect the control flow to ED-monitor for designated events as usual. Then ED-monitor can directly access all of VMM's code and data for the integrity assessment purpose.

2.4 Requirements and Challenges

Within our threat model, whether one can trust our monitoring framework relies on two security requirements:

1) Integrity of ED-monitor. The integrity of ED-monitor should never be corrupted by the VMM. This requirement guarantees that the integrity assessment work performed by ED-monitor is trusted. It can be further divided into three integrity requirements:

- **Memory integrity.** ED-monitor's memory containing its code and data should never be corrupted by the VMM.
- **Control flow integrity.** ED-monitor's control flows should never be maliciously altered by the VMM.
- **Designated entry point.** Execution should switch to ED-monitor only through the designated entry point.

2) Protection of hook placement. The hooks implanted inside the untrusted VMM should not be removed or modified by the VMM. In addition, when the VMM's execution reaches a hook, the control flow must be non-bypassably transferred to ED-monitor.

It is challenging to achieve both security requirements in our framework. The highest-privilege VMM, which is executed in the same world, can easily compromise the integrity of unprotected ED-monitor and the hook placement.

2.5 Our Approach: Mutual-Protection

In the literature, address space randomization (ASR) is a lightweight approach to protect code and data from the adversary (e.g., a compromised application) residing in the same address space. It randomizes the code and data to ensure that their locations are unpredictable in virtual address space, thus preventing the adversary who has no knowledge of the memory locations from subverting them. This ASR-based protection has been well explored to hide code and data against application-level attacks. For example, O-CFI [25] uses this approach to hide its BLT page. However, we found that the existing ASR techniques are wholly insufficient to hide ED-monitor against compromises of the highest-privilege VMM. The difference between an application and a VMM is that the VMM can additionally perform the most privileged operations (e.g., privileged instruction execution, MMU configuration, interrupt handling). With these privileged operations, the VMM can easily bypass ASR or find out the randomized location, e.g., the VMM could manipulate MMU configuration to remap ED-monitor's memory to a known address and hence bypasses ASR.

In order to make the ASR-based protection effective against compromised VMM, we should deprive VMM's ability of directly performing the most privileged operations. To this end, an existing method is to leverage code instrumentation to eliminate all privileged instructions in VMM code, so that the VMM must request ED-monitor to execute privileged instructions on its behalf (similar to TZ-RPK [5]). In this way, ED-monitor can intercept and validate the most privileged operations in the VMM (privileged instruction execution, MMU configuration, interrupt handling, etc.). How-

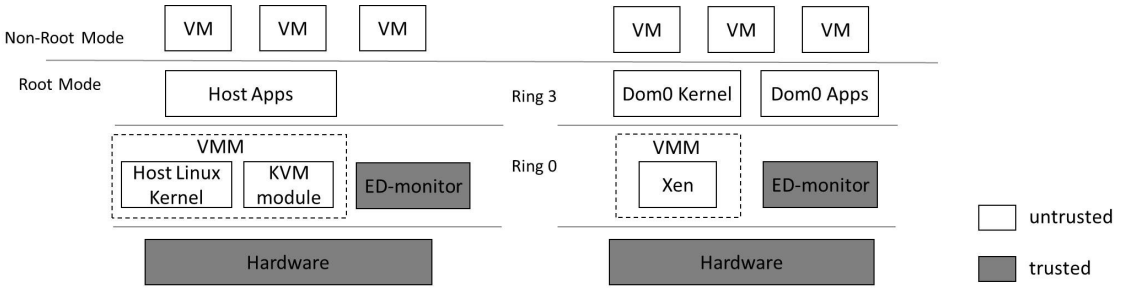

Figure 1. An overview of our framework based on both KVM (left) and Xen (right) architectures

ever, implementing this instrumentation-based privilege restriction (IPR) is challenging, because the highest-privilege VMM can easily introduce new privileged instructions into code to bypass the intercept or subvert ED-monitor's integrity to compromise the validation.

In this paper, we combine ASR and IPR in an innovative way which provides a unique form of mutual-protection. Although neither ASR nor IPR can resist the threats from the highest-privilege VMM as discussed above, it is clearly under-investigated whether combining ASR and IPR can solve this problem. Our unique mutual-protection (between ASR and IPR) generates a surprising result: IPR can leverage ASR to guarantee the inevitability of interception and validation on VMM's most privileged operations, and in turn, ASR can rely on IPR to mitigate attempts on destructing the ASR-based protection. In this way, we simultaneously protect ASR and IPR even in face of a completely compromised VMM. In the following sections, we will detail the design of our mutual-protection mechanism and the trusted event-driven monitoring framework built on it.

3. Design of Mutual-Protection

In this section, we will detail the technique IPR in Section 3.1, and ASR in Section 3.2. In the meanwhile, we will describe how the mutual-protection relationship between them is established.

3.1 Instrumentation-based Privilege Restriction

The technique IPR allows ED-monitor to intercept and validate VMM's most privileged operations, including privileged instruction execution, MMU configuration, interrupt and exception handling, IOMMU configuration and hardware virtualization configuration. In the following subsections, we will discuss the interception and validation of these operations one by one. They are the foundation of our mutual-protection mechanism.

3.1.1 Interception of Privileged Instruction Execution

We leverage offline code instrumentation to eliminate privileged instructions in VMM code. In this way, the VMM is forced to route the execution of privileged instructions to ED-monitor, and thus ED-monitor can intercept and validate the execution of each privileged instruction before it changes system states. However, considering the VMM is executed at the highest privilege level, providing a non-bypassable assurance for this method is not straightforward.

Specifically, the code bytes forming a privileged instruction include both intended bytes which are originally executed as a privileged instruction, and unintended bytes located at an un-aligned instruction boundary. To eliminate intended bytes, we simply replace them with a hook that redirects the control flow to ED-monitor. For unintended bytes, we eliminate them from VMM code using instruction substitution. We substitute the instructions forming the unintended bytes with other function-equivalent instructions that have no unintended bytes in them.

After privileged instruction elimination, we ensure that the VMM cannot introduce any new privileged instruction for execution with the following two mechanisms. *1)* We enforce a strict memory mapping policy that maps VMM's code and data pages as either writable or executable (W$\oplus$X), which disallows any attempt that either modifies existing code or brings in new code for execution. Furthermore, the VMM is not permitted to maliciously modify page tables to violate the W$\oplus$X mapping policy, because ED-monitor also intercepts MMU configuration and makes necessary validations (see Section 3.1.2). *2)* We use an x86 hardware feature named SMEP [17] to prevent the VMM from executing other code residing at lower privilege level, e.g., host apps' code in ring 3. The VMM cannot disable the SMEP protection because the privileged instruction (mov-to-cr4) which is used to disable this hardware feature has been eliminated in VMM code.

3.1.2 Interception of MMU Configuration

In x86, MMU configuration includes two most privileged operations: executing the privileged instruction mov-to-cr3 to modify the cr3 register and updating page tables. As discussed earlier, intercepting the execution of the mov-to-cr3 instruction is realized through privileged instruction elimination. To intercept page table update, we leverage the technique named shadow page table which is originally used in virtualization. Specifically, the VMM continues to maintain its own page tables; however the hardware MMU actually

uses the page tables maintained by ED-monitor. We refer to the first page tables as VMM page tables (VMMPTs), and the second as shadow page tables (SPTs). ED-monitor allocates the whole SPTs within its internal memory and guarantees that the cr3 register must point to the root of SPTs based on mov-to-c3 intercept. Since ED-monitor's memory integrity is effectively protected by ASR based on the mutual-protection (detailed in Section 3.2), the VMM cannot update SPTs directly and must rely on ED-monitor to synchronize the page table updates to SPTs. As a result, ED-monitor can intercept and validate each SPT update before it is applied to MMU.

3.1.3 Interception of Interrupt and Exception Handling

In x86, the handlers of interrupts and exceptions are specified by an architectural in-memory table, the Interrupt Descriptor Table (IDT). The base address of IDT is specified by a system register, the Interrupt Descriptor Table Register (IDTR). To intercept interrupt and exception handling, we allow the VMM to continue to create and manage its own IDT whereas the CPU actually uses a different IDT (shadow IDT) prepared by ED-monitor. When an interrupt (or an exception) arrives, it will be first delivered to shadow IDT. In this way, ED-monitor can first intercept and handle the interrupt, and then deliver it to VMM's IDT for handling.

We ensure the security of shadow IDT using the following mechanisms. *1)* ED-monitor intercepts the execution of the privileged instruction lidt, and does not permit the VMM to maliciously modify the IDTR. *2)* Shadow IDT and its handlers' code are mapped as read-only to prevent being maliciously modified. ED-monitor does not permit the VMM to change the read-only settings based on its interception of MMU configuration. *3)* Since IDTR and shadow IDT entries all hold virtual addresses, their corresponding address mappings in SPTs are also protected by ED-monitor. *4)* Shadow IDT uses interrupt gates [17] to specify its handlers, which ensures that the execution of the handler is atomic with interrupt disabled. With these four mechanisms, whenever an interrupt (or an exception) arrives, it will be non-bypassably delivered to a handler of shadow IDT. Then, the handler will atomically invoke ED-monitor to handle the interrupt without interference of the VMM.

3.1.4 Interception of IOMMU Configuration

Modern x86 processors provide the IOMMU hardware that translates addresses used in DMA transactions to protect physical memory from illegal accesses of DMA. In VT-d, the control register set of IOMMU is placed at a fixed 4KB-aligned physical memory location. It specifies I/O page tables to complete the address translations in DMA transactions. In our framework, ED-monitor intercepts and emulates VMM's accesses to the control register set using SPTs, and further leverages shadow I/O page tables to control the address translations in DMA transactions.

3.1.5 Interception of Hardware Virtualization Configuration

In x86, hardware virtualization introduces root mode and non-root mode execution. Certain operations are restricted in non-root mode. If software running in non-root mode (e.g., a VM) tries to conduct these restricted operations, the processor triggers a VM exit which transits execution to root mode. Root mode software (e.g., a VMM) can use the architectural in-memory structure named virtual-machine control structure (VMCS) to manage the mode transitions as well as specify the restricted operations causing the VM exit. The VMCS can be also used to specify extended page tables (EPTs) that control the memory access in non-root mode. In our framework, ED-monitor intercepts and validates two kinds of hardware virtualization configuration: *1)* VMCS manipulations and *2)* EPT updates.

VMCS manipulations. Hardware virtualization includes a set of privileged instructions (vmptrld, vmptrst, vmclear, vmread, vmwrite) to load, store, clear, read and write the current VMCS. ED-monitor intercepts the execution of these privileged instructions by eliminating them from VMM's code (recall Section 3.1.1), so that any manipulation on the VMCS can be validated by ED-monitor. In addition, since the content of the VMCS would be synchronized to its associated in-memory region (named VMCS region in x86), the VMM could bypass the interception via directly accessing the VMCS region. To prevent this, ED-monitor additionally maps the VMCS region as inaccessible in the address space using SPTs.

Hardware virtualization introduces another way to modify system registers, such as the cr3 register and the IDTR. When the CPU switches from non-root mode to root mode, it loads some of the system registers from the corresponding field of the VMCS. For instance, the cr3 register is loaded from the HOST_CR3 field of the VMCS. With this feature, the compromised VMM would maliciously manipulate the VMCS to modify these system registers and thus bypass ED-monitor's interception. To avoid this, ED-monitor intercepts and validates each VMCS manipulation to prevent any interception-violating value of system registers from being written to the VMCS.

EPT updates. To intercept EPT updates, we still keep the VMM managing its own EPTs, however the root of EPTs (the EPTP field of the VMCS) actually points to the EPTs (shadow EPTs) maintained by ED-monitor. The VMM cannot manipulate the EPTP field or shadow EPTs that are stored in ED-monitor's memory. When the VMM attempts to update its own EPT, ED-monitor traps the update operation and synchronizes the update to the corresponding shadow EPT. In this way, ED-monitor can make necessary validations on every EPT update and control the memory accesses in non-root mode.

3.2 Address Space Randomization

As discussed in Section 3.1, the effectiveness of IPR relies on the integrity of ED-monitor. Firstly, IPR depends on ED-monitor's integrity (memory integrity) to guarantee that SPTs, shadow I/O page tables and shadow EPTs allocated within ED-monitor's internal memory cannot be maliciously modified. Additionally, IPR depends on ED-monitor's integrity to ensure that the validation performed by ED-monitor cannot be corrupted. Therefore, we make use of ASR to protect the integrity of ED-monitor.

The idea of our ASR-based protection is conceptually simple: we map all of ED-monitor's memory to a random location in virtual address space during the initialization phase at system startup, and rely on the huge size of virtual address space (e.g., 2^{48} Bytes in x86_64) to achieve probabilistic protection. We assume that there exists a trusted booting mechanism and thus, the initialization phase which starts immediately after a clean boot can be considered to be trusted. This assumption is shared with previous monitoring frameworks [27, 28].

However, the challenge is how to prevent the VMM from subverting the ASR-based protection when the VMM enters a running state where it is assumed to be subject to malicious VMs and other attack attempts. In the following, we first identify possible attacks that could be performed by the untrusted VMM to subvert our ASR-based protection. Then, we leverage mutual-protection to let the first technique (IPR) prevent each of these possible attacks and thus achieve high-assurance probabilistic protection

3.2.1 Possible Attacks

Within our threat model, the ASR-based protection is vulnerable to three kinds of attacks: ASR-bypass attacks, information-leak attacks and brute-force attacks.

ASR-bypass attacks. Since ASR is deployed in virtual address space, the VMM could disable hardware paging or initiate malicious DMA transactions to access ED-monitor's memory directly in physical address space. The VMM could also manipulate MMU configuration to re-map ED-monitor's memory to other known virtual addresses, and hence bypass ASR. In addition, hardware virtualization brings in additional CPU mode (non-root mode), which could be maliciously used to bypass ASR. Specifically, the VMM could launch a non-root mode execution (e.g., a manipulated VM) and access ED-monitor's memory in non-root mode.

Information-leak attacks. ASR-based protection is vulnerable to information-leak attacks in which the attackers are able to acquire the information about the address space layout and find out the location of ED-monitor's memory. At software level, any pointer to ED-monitor's virtual address could cause information leaks. At hardware level, due to the highest privilege, the VMM could also access the underlying hardware configuration or state to acquire ED-monitor's virtual address information. In x86, the potential hardware-level leakage sources include: *1)* MMU configuration. ED-monitor inevitably relies on MMU configuration to translate each of its virtual addresses to the physical address. The translation information could be a leakage source. *2)* System register. In x86, the cr2 register contains the page-fault virtual address when a page fault occurs. Thus, when ED-monitor encounters a page fault during its execution, the cr2 register will be a leakage source. *3)* Debugging hardware. For example, Last Branch Recording (LBR) is a hardware feature supported in recent Intel processors [17]. When LBR is enabled, the processor records the last N branches (source and target virtual addresses of the branches) in a set of MSRs or a specified memory buffer. Therefore, these branch records would result in information leaks if LBR is enabled during ED-monitor's execution.

Brute-force attacks. The VMM could also perform a brute-force scan of the whole virtual address space to find out ED-monitor's location. In previous application-level ASR-based protection [7, 25], attackers cannot safely scan the whole address space, because defenders can place as many as unmapped pages in the address space. Any reference to these unmapped pages will trigger the page fault and will be detected. However, in our threat model, the VMM with the highest privilege could maliciously manipulate the MMU configuration or page-fault handler to safely probe the whole virtual address space and launch brute-force attacks without being detected.

In the following, we will discuss how to depend on IPR to prevent each of these attacks.

3.2.2 Preventing ASR-bypass Attacks

ED-monitor does not permit the VMM to disable hardware paging by intercepting the execution of the privileged instruction mov-to-cr0. It also uses shadow I/O page tables to prevent DMA transactions that directly access ED-monitor's memory in physical address space. In this way, the VMM can only access ED-monitor's memory through virtual address space. In addition, ED-monitor intercepts and validates every MMU configuration update, and does not permit the VMM to re-map ED-monitor's memory to any other virtual address. ED-monitor also validates every EPT update to prevent its memory from being accessed in any non-root mode execution. As a result, ED-monitor's memory is only mapped to the random location in virtual address space in root mode. To access ED-monitor's memory, the VMM must first know the random location.

3.2.3 Preventing Software-level Information Leaks

The next step is to prevent address leaks of ED-monitor. At software level, there are no ED-monitor's virtual addresses maintained in the VMM. This keeps ED-monitor hidden from the VMM. However, the VMM must invoke the execution of ED-monitor, e.g., to request ED-monitor to execute privileged instructions or update SPTs. This process would

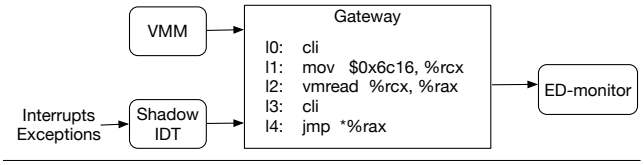

Figure 2. Gateway

be maliciously used by the VMM to cause address leaks. In the following, we break the process into three phases (ED-monitor invocation, ED-monitor execution and ED-monitor return), and detail how to prevent address leaks for each phase.

ED-monitor invocation. The invocation of ED-monitor requires the VMM to know ED-monitor's entry point. However, the entry point, which is a code address of ED-monitor, is the cause of address leaks. To address this, we propose a novel approach for the VMM to enter ED-monitor securely while preventing address leaks caused by the entry point. Specifically, since the VMM needs to use the VMCS to control the non-root mode environments of guest VMs, we store the entry point of ED-monitor in the HOST_RIP field of the VMCS. To prevent address leaks, ED-monitor does not permit the VMM to read this field relying on its interception of VMCS manipulations (recall Section 3.1.5).

Based on the entry point hidden in the VMCS, we provide a secure gateway for the VMM to enter ED-monitor. The gateway is a read-only and executable page mapped at a predefined virtual address known to the VMM.

Figure 2 shows the instruction sequence in the gateway. The sequence first disables interrupts (at I0) and loads ED-monitor's entry point from the VMCS's HOST_RIP field to the rax register (at I1 and I2). Then, the cli instruction (at I3) is executed again (in case that the first one would be maliciously bypassed) and the control flow is finally transferred to the entry point of ED-monitor (at I4). Though the gateway contains a vmread instruction, the VMM can never use it to acquire the information hidden in the VMCS. This is because once the vmread instruction is executed, interrupt is immediately disabled and the control flow will be atomically transferred to ED-monitor. Although there exists a small attack window that the VMM could generate a malicious interrupt or exception (e.g., a single-step exception) between I2 and I3 and regain the control to steal ED-monitor's entry point stored in the rax register, ED-monitor prevents this attack by detecting such interrupt or exception using shadow IDT. ED-monitor then runs atomically with interrupt disabled during its execution. It also temporarily blocks any non-maskable interrupt (NMI) by shadow IDT. The VMM cannot regain the control until ED-monitor finally switches back to the VMM. With this design, the VMM can never acquire any virtual address of ED-monitor during the invocation phase.

Since the VMM may use multiple VMCSes to launch different guest VMs, ED-monitor guarantees that the entry point is always stored in the current VMCS, so that the entry point can be always acquired through the vmread instruction in the gateway. Specifically, when the VMM tries to execute the vmptrld instruction to load the current VMCS, ED-monitor intercepts the execution of this privileged instruction and then stores the entry point in this current VMCS. If the VMM tries to execute the vmclear instruction to make the current VMCS invalid, ED-monitor intercepts this execution and additionally specifies an empty VMCS as the current VMCS to store the entry point. In addition, ED-monitor intercepts and emulates the execution of the vmxon instruction and the vmxoff instruction in case that the VMM disables hardware virtualization. Since hardware virtualization is also controlled by the VMXE bit of the cr4 register and the IA32_FEATURE_CONTROL MSR, ED-monitor also intercepts the mov-to-cr4, the mov-from-cr4, the rdmsr and the wrmsr instruction to provide a virtualized view of these system registers to the VMM.

Another benefit of storing ED-monitor's entry point in the HOST_RIP field of the VMCS is that ED-monitor is able to non-bypassably capture each VM exit event for event-driven monitoring. Specifically, whenever a VM exit occurs, the control flow will be first transferred to ED-monitor's entry point specified by the HOST_RIP field. ED-monitor handles the VM exit event (if needed) and then transfers the control flow to the VMM.

In addition, Figure 2 also shows the procedure of entering ED-monitor when interrupts or exceptions occur. When an interrupt or exception arrives, the handler of shadow IDT will invoke the gateway and then the control flow will be transferred to ED-monitor. Since no ED-monitor's address is stored in shadow IDT or its handlers, there is no need to hide shadow IDT from the VMM. Therefore, ED-monitor does not intercept the execution of the sidt instruction. The VMM is free to directly execute the sidt instruction to acquire the address of shadow IDT, but cannot modify the read-only shadow IDT and its handlers.

ED-monitor execution. To avoid address leaks in the execution phase, ED-monitor is designed to be self-contained. It performs the simple validation and VMM integrity assessment tasks based on its separate executable, stack and dynamic memory, and does not rely on any VMM service.

ED-monitor return. After ED-monitor finishes its validation and VMM integrity assessment tasks, it clears all the information in CPU registers, switches the stack, and resumes the execution of the VMM.

3.2.4 Preventing Hardware-level Information Leaks

We leverage mutual-protection that lets IPR deprivilege the VMM, so that the VMM is not permitted to directly access the underlying hardware configuration or state that could cause information leaks. Specifically, for each potential hardware-level leakage sources presented in Section 3.2.1, the countermeasures are taken as follows. *1)* MMU configuration. The VMM is allowed to access its own VMMPTs

as usual, but there is not any address mapping information of ED-monitor in VMMPTs. The VMM cannot read the address mappings in SPTs that are stored in ED-monitor's memory. *2)* System register. ED-monitor intercepts the execution of the privileged instruction mov-from-cr2 and does not permit the VMM to directly read the cr2 register. The cr3 register will not cause information leaks, because even if the attacker acquires the physical addresses of SPTs via cr3, she still cannot access SPTs in virtual address space. *3)* Debugging hardware. To prevent leaks from debugging hardware, in our current implementation, ED-monitor just forbids the VMM to directly access the debug registers and debugging control MSRs (e.g., IA32_DEBUGCTL MSR). This is achieved by intercepting the execution of the mov-to-dr, mov-from-dr, wrmsr and rdmsr instructions.

3.2.5 Preventing Brute-force Attacks

ED-monitor is randomly located inside a huge virtual memory area and the locations in this area other than where occupied by ED-monitor are mapped as access-deniable based on SPTs. The VMM is not permitted to maliciously manipulate the MMU configuration to modify the access-deniable settings. In this way, the brute-force scan, which requires to probe the whole area, will trigger the deniable accesses with a high possibility and will raise alerts in ED-monitor with page-fault exceptions.

3.2.6 Probabilistic Protection

Since the VMM cannot bypass ASR or make any precise assumption on the location of ED-monitor, it must guess ED-monitor's location which is randomized at system boot. The odds of correctly guessing the location are low enough to provide probabilistic protection. In our current implementation, the whole size of the virtual memory of ED-monitor is less than 1 GB. We randomize it in a region of the virtual address space that is not used by 64-bit Xen or KVM architecture (the whole size is 2^{44} Bytes). The chances of guessing ED-monitor's location are:

$$\frac{1GB}{2^{44}B} = \frac{1}{16384} \quad (1)$$

Since a benign VMM will never access the region where ED-monitor is randomized, any incorrect guess in this region can be considered to be malicious. These incorrect guesses will trigger the deniable accesses and will raise alert.

4. Trusted Event-driven VMM Monitoring

On top of the mutual-protection mechanism, we successfully achieve both security requirements (presented in Section 2.4) for trusted event-driven VMM monitoring.

4.1 Protection of Hook Placement

The mutual-protection enables ED-monitor to have full mediation power over the MMU configuration. Therefore, ED-monitor enforces the W⊕X mapping policy to protect code hooks and uses SPTs to protect data hooks similar to the approach proposed in Lares [27]. The implanted hooks redirect the control flow to a self-contained trampoline which immediately disables interrupt with a cli instruction and stores VMM's current execution context. Then the trampoline atomically jumps to the gateway of ED-monitor. Since the trampoline is self-contained and atomically executed, the control flow will be inevitably transferred to ED-monitor whenever the execution reaches a protected hook.

4.2 Assurance of ED-monitor's Integrity

With the mutual-protection, we achieve all of the three integrity requirements for the integrity of ED-monitor:

Memory integrity. All of ED-monitor's memory is protected by ASR and all possible attacks are prevented as discussed from Section 3.2.2 to Section 3.2.5.

Control flow integrity. ED-monitor runs with interrupt disabled and blocks NMIs during its execution. The VMM cannot subvert ED-monitor's control flows.

Designated entry point. Since VMM has no knowledge of the location of ED-monitor, it can only invoke ED-monitor through the gateway that specifies the designated entry point. Other entry points are not allowed.

4.3 ED-monitor API

Our framework allows system administrators to add their event-driven monitoring policies to ED-monitor through the ED-monitor API. In our current implementation, the ED-monitor API includes programming interface to specify the places of hooks that trigger the monitoring process, monitored memory regions, integrity rules enforced on the monitored memory regions (e.g., whitelisting, semantic verification), and the corresponding actions. The ED-monitor API enables convenient and rapid development of the event-driven monitoring policies based on our framework.

5. Prototype Implementation

We have implemented a prototype of our VMM monitoring framework based on KVM architecture on x86_64 platforms. System administrators who want to use our framework only need to *1)* perform offline code instrumentation on VMM's binary executable, *2)* add their monitoring policies to ED-monitor through the ED-monitor API, and *3)* load ED-monitor into the system as a kernel module of host Linux at system startup. *We do not require any modification to VMM's source code.* ED-monitor currently includes around 5.5K source lines of code.

5.1 Offline Code Instrumentation

Table 1 lists all of the privileged instructions that need to be eliminated in VMM code, in order to achieve mutual-protection. The reason why we need to eliminate them has been discussed in previous sections. We make use of offline code instrumentation on the VMM's binary executable with

Table 1. Privileged instructions to be eliminated.

Instructions	Opcode	Instructions	Opcode
mov-to-cr3	0x 0f 22/3	mov-from-dr	0x 0f 21/r
mov-from-cr3	0x 0f 20/3	vmxon	0x f3 0f c7/6
mov-to-cr0	0x 0f 22/1	vmxoff	0x 0f 01 c4
mov-from-cr0	0x 0f 20/0	vmptrld	0x 0f c7/6
mov-to-cr4	0x 0f 22/4	vmptrst	0x 0f c7/7
mov-from-cr4	0x 0f 20/4	vmclear	0x 66 0f c7/6
mov-from-cr2	0x 0f 20/2	vmlaunch	0x 0f 01 c2
lidt	0x 0f 01/3	vmlresume	0x 0f 01 c3
wrmsr	0x 0f 30	vmread	0x 0f 78
rdmsr	0x 0f 32	vmwrite	0x 0f 79
mov-to-dr	0x 0f 23/r		

debugging information, to eliminate both intended and unintended bytes of the privileged instructions in VMM code. Debugging information is required to disassemble VMM's code and identify the locations of the privileged instructions.

Eliminating intended bytes. For intended bytes of a privileged instruction, we simply replace them with a jmp instruction that will redirect the control flow to the gateway of ED-monitor. Since the jmp instruction would overwrite the subsequent instructions in VMM code, we additionally execute the overwritten instructions in a code stub when ED-monitor resumes the execution of the VMM.

Eliminating unintended bytes. As shown in Table 1, the privileged instructions to be eliminated all have at least 2-byte opcode. Thus their unintended bytes, which must fully match this rigorous formation, are rare in VMM's code. For example, we only found tens of unintended privileged instructions in the host Linux kernel image (version 3.8.0).

To eliminate the unintended bytes introduced by other instructions, we simply substitute these instructions with others that perform the same computation but do not contain any unintended bytes in them. This substitution based method, which is originally used to remove unintended return instructions in kernel code, has been well described in previous work [20, Section 3.4]. Actually, there is no fundamental difference between removing unintended return instructions and unintended privileged instructions. The elimination of the unintended bytes is detailed in Appendix A.

5.2 Initialization Phase

The initialization phase of our framework is initiated by loading ED-monitor as a host Linux kernel module into the system at system startup. We assume the existence of a trusted booting mechanism (e.g., Intel TXT [16]). Therefore, the initialization phase which starts immediately after a trusted boot can be considered to be trusted.

During the initialization phase, ED-monitor first performs a byte-by-byte scanning on VMM's code to verify that all of the privileged instructions (intended and unintended) are eliminated, so that the whole offline code instrumentation process can be removed from TCB. Since host Linux kernel will modify its code and bring in new privileged instructions at system startup, ED-monitor eliminates these newly added privileged instructions at runtime before the verification. Then, ED-monitor initializes the IPR and ASR and enables the mutual-protection. Finally, ED-monitor performs the initialization work for the monitoring policies, such as implanting hooks in the VMM.

5.3 Paravirtualization Interface

We take full advantage of host Linux kernel's paravirtualization interface to facilitate the implementation of the interception of MMU configuration. The paravirtualization interface includes natural hooks which are originally for the Linux kernel (which is the VMM now) to make requests to a higher privilege software layer, when the Linux kernel needs to perform MMU configuration. ED-monitor modifies these hooks during the initialization phase so that all VMM's MMU configuration requests will be redirected to the gateway and will be finally handled by ED-monitor. Then, ED-monitor synchronizes the updated address mapping from VMMPTs to SPTs and makes necessary validations. The use of paravirtualization interface brings the benefits of *1)* decreasing the TCB size of ED-monitor, *2)* improving the performance of the interception and *3)* requiring no modifications to VMM's source code.

5.4 Loadable Kernel Modules

Host Linux kernels often need to load kernel modules at runtime. However, attackers may use privileged instructions in malicious kernel modules to bypass IPR. To prevent this, our current implementation requires system administrators to first perform offline code instrumentation to remove all privileged instructions in the kernel module binary. When the host Linux kernel loads the kernel module and tries to map the kernel module code to be executable in the address space, ED-monitor will intercept this procedure and make validations on the code. If any privileged instruction exists in code, ED-monitor will deny the mapping process.

6. Security Analysis

In this section, we systematically analyze possible threats against the mutual-protection mechanism which serves as the cornerstone of our trusted event-driven VMM monitoring framework. The analysis results demonstrate that the mutual-protection scheme can effectively protect both IPR and ASR from being subverted.

To compromise the mutual-protection, an adversary can subvert either IPR or ASR. We examine both attacks in the following.

6.1 Subverting IPR

IPR enables ED-monitor to intercept and validate VMM's most privileged operations. To subvert IPR, the adversary can either bypass the interception or subvert the validation.

6.1.1 Bypassing the Interception

As discussed in Section 3.1, to bypass the interception, the adversary must either introduce new privileged instructions for execution or directly manipulate SPTs, shadow I/O page tables or shadow EPTs. To achieve the first goal, the adversary must disable the SMEP or tamper the W⊕X restriction. Disabling the SMEP requires the mov-to-cr4 instruction, which however has been eliminated from VMM code. Changing the W⊕X restriction must modify the SPTs. However, the SPTs, shadow I/O page tables and shadow EPTs are all allocated in ED-monitor's internal memory and all managed by ED-monitor. Thus, to bypass the interception, the adversary must compromise ED-monitor's integrity. However, as discussed in Section 3.2, we make use of ASR to protect ED-monitor's integrity while relying on IPR to thwart all attempts on destructing the ASR-based protection (as detailed from Section 3.2.2-3.2.5). That is to say, before successfully subverting IPR, the advisory would not be able to compromise ED-monitor's integrity or bypass the interception. In this case, using the way of bypassing the interception to subvert IPR is not viable.

6.1.2 Subverting the Validation

Once VMM's most privileged operations are intercepted, ED-monitor performs validations on them based on its separate code and data without relying on any VMM service. To subvert the validation, the adversary must compromise ED-monitor's integrity. However, as analyzed above, without successfully subverting IPR, the adversary would be unable to compromise ED-monitor's integrity. Therefore, using the way of subverting the validation to subvert IPR is also unachievable.

6.2 Subverting ASR

To subvert ASR, the adversary can perform three kinds of possible attacks including ASR-bypass attacks, information-leak attacks and brute-force attacks. However, as detailed from Section 3.2.2 to 3.2.5, we rely on IPR to prevent all of these possible attacks. Therefore, to subvert ASR, the adversary must first subvert IPR. However, subverting IPR is impossible as discussed above.

7. Performance Evaluation

We now report the performance of our monitoring framework. We first report the overhead introduced by the mutual-protection. Then, we evaluate the overhead of performing event-driven VMM monitoring based on the mutual-protection mechanism. To demonstrate the performance advantages, we compare the overhead with that of nested virtualization [9] and Nested Kernel [13].

We made experiments on a Dell Optiplex 9010 host configured with an Intel i7-3770 (4 cores) processor, an 8 GB RAM, a 500 GB SATA disk and an Intel Ethernet 10Gbps NIC. The host Linux kernel and KVM module came from a 64-bit Ubuntu 12.10 Linux distribution with the unmodified Linux kernel 3.8.0. The guest VM came from an unmodified 64-bit Ubuntu 12.04 Linux distribution. We configured the guest VM with one virtual CPU, 512MB RAM and a virtual NIC. In addition, we used another similar remote computer (connected directly by 10Gbps fiber) as a target for network I/O measurement.

Table 2. Lmbench results in host Linux kernel.

Microbenchmarks	Native (us)	Our framework (us)	Overhead
page fault	0.1765	0.1887	1.07x
mmap	4314	4569	1.06x
protection fault	0.1918	0.2046	1.07x
signal installation	0.0891	0.0936	1.05x
signal delivery	0.5727	0.5765	1.01x
fork+exit	161	189	1.17x
fork+exec	522	582	1.11x
fork+/bin/sh	1304	1550	1.19x
ctxsw 2p/0k	4.46	4.59	1.03x
ctxsw 8p/4k	5.53	5.64	1.02x
ctxsw 16p/8k	5.59	5.67	1.01x

7.1 Mutual-protection Overhead

We leveraged a wide range of benchmarks to measure the overhead of mutual-protection experienced by both VMM and guest VM. At this time, ED-monitor did not capture any VMM event or perform any monitoring task, and thus the benchmark results only exhibit the mutual-protection overhead in our framework.

Overhead experienced by VMM. We used Lmbench [24] to measure the latencies of different individual kernel operations in host Linux kernel. We restricted the measurement to a subset of the microbenchmarks, including memory operations, signal dispatch, process creation and context switch, since these are areas that may be affected by mutual-protection. Table 2 shows the average result based on three identical runs. The overhead for each microbenchmark is small, indicating that the mutual-protection is quite lightweight.

Overhead experienced by Guest VM. We ran two I/O intensive benchmarks (Netperf [4] and ApacheBench [2]) in a guest VM to measure the mutual-protection overhead experienced by guest VM. Netperf and ApacheBench, which trigger many VM exits into the VMM, can exhibit the worst case of the overhead. Specifically, we used Netperf's TCP_STREAM test to measure the throughput of transferring 16KB message to the remote machine with a single TCP connection. We used Netperf's TCP_RR test to measure the TCP request and response latency. We used ApacheBench to measure the throughput of requesting 256-byte files. We configured ApacheBench to use 100 concurrent threads and to perform 10,000 requests. Figure 3 shows the average results of both Netperf and ApacheBench based on ten identical runs with the standard deviations being up to 0.6% of the average. We used the native execution as the baseline

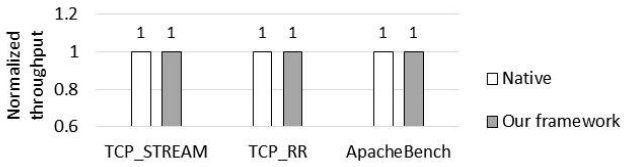

Figure 3. Netperf and ApacheBench results for mutual-protection overhead

and calculated the normalized throughput. As we see, the performance degradation of mutual-protection experienced by guest VM can be ignored.

7.2 Monitoring Overhead

7.2.1 ED-monitor Invocation Overhead

We used microbenchmarks to measure the invocation overhead of ED-monitor in our framework. We measured the time required to switch to ED-monitor through the gateway when the execution reaches an implanted hook and immediately switch back. ED-monitor in this measurement did not perform any monitoring task. To compare the result with nested virtualization, we also implanted a hook (a vmcall instruction) into a VMM that runs in a nested virtualization environment and measured the time required to switch to the host hypervisor when the execution reaches the hook and immediately switch back.

We repeatedly executed the round trip 100000 times and used the rdtsc instruction that reads CPU's Time Stamp Counter (TSC) to measure the average cycles. Then, we converted the measured cycles to microseconds based on the TSC speed. Table 3 shows the results based on ten identical runs. The monitor invocation in our framework (VMM/ED-monitor switch) is 67 times faster than that (VMM/Host hypervisor switch) in nested virtualization.

For comparison, we also list the overhead of switching to the same-privilege TCB in Nested Kernel [13]. As we see, this overhead (Kernel/Same-privilege TCB switch in the table) is still in the same order of magnitude with that of VMM/Host hypervisor switch, due to the execution of the expensive mov-to-cr0 instruction in Nested Kernel. It is still 18 times slower than the switch in our framework.

We also used the syscall instruction to invoke an empty system call added to the kernel, and measured the switch time between an application and the kernel. The result is also shown in Table 3. The VMM/ED-monitor switch is even 11.6 times faster than the application/kernel switch that uses the fast system call.

We achieve this substantial performance improvement because the switching between the VMM and ED-monitor does not require any expensive boundary-crossing. An event-driven monitoring policy which requires the monitor to be frequently invoked can be significantly benefitted from the performance advantages of our framework.

Table 3. Overhead of different switches.

Switches	Average time (us)	Std. dev.
VMM/ED-monitor	0.0076	0.00016
VMM/Hosted hypervisor	0.515	0.008
Kernel/Same-privilege TCB [13]	0.139	-
Application/Kernel	0.0882	0.002

The mutual-protection also experiences the overhead of switching between the VMM and ED-monitor, since ED-monitor needs to intercept VMM's most privileged operations to achieve mutual-protection. Due to the negligible switch overhead measured above, it is unsurprising that the mutual-protection overhead (measured in Section 7.1) is trivial.

7.2.2 Overhead of VMM Data Integrity Monitoring

To perform a more elaborate and practical evaluation of the performance advantages compared with nested virtualization, we developed a monitoring policy that performs event-driven VMM data integrity monitoring based on our framework.

Event-driven data integrity monitoring for KVM module. Although data integrity is essential to the security of VMM, no existing research provides a complete solution to this problem. In this monitoring policy, we solve a subset of this problem. Since the KVM module is the key component in the VMM and suffers from large attack surface, we use ED-monitor to monitor the data integrity of critical areas in the KVM module in an event-driven manner. In our policy, critical areas are defined as critical function pointers and critical object fields.

To monitor a function pointer, we statically identify all of its possible values as the whitelist and implant code hooks at every location where the function pointer is called in code. In this way, whenever the function pointer is used to divert the control flow in the KVM module, ED-monitor will be first invoked to check the integrity of the function pointer according to the whitelist.

To monitor an object field, we maintain a shadow copy for it within ED-monitor's memory. We also implant code hooks at all regular write access points of the object field in KVM module code, so that ED-monitor will be invoked to update the shadow copy whenever the object field is updated. Thus, the values of the object field and its shadow copy always keep the same during benign execution. This equivalence is used as the invariant for our integrity monitoring. Malicious operations that try to modify the object field (e.g., a heap overflow) is hard to keep such invariant because they cannot directly manipulate the shadow copy that is securely stored in ED-monitor's internal memory. Consequently, we use ED-monitor to monitor the invariant in an event-driven manner. We implant code hooks at all regular read access points of the object field, so that ED-monitor will be invoked to compare the value of the object field with

the corresponding shadow copy. This event-driven monitoring policy guarantees that the integrity of the object field is timely checked whenever it is read for computation, so that the attack window can be significantly reduced.

In our current implementation, we selected 110 function pointers in KVM module and 16 object fields in six objects (kvm, kvm_arch, kvm_vcpu, kvm_run, kvm_vcpu_arch and kvm_mmu) as critical areas. These 16 object fields are chosen because in our observations they are the key fields for managing guest VMs. Then we made use of manual source code analysis to identify all call sites for the function pointers (overall 111 call sites) and all regular read and write access points for the object fields (overall 125 access points). Note that the point-to analysis or the dynamic analysis proposed in HookSafe [30] can be also used to automate this process. Finally, we implanted the code hooks at each call site and each access point to enable the data integrity monitoring.

Performance comparison. We applied the above monitoring policy to our framework. Since the policy only monitors the data integrity of the KVM module, we chose I/O intensive benchmarks (Netperf and ApacheBench) that run in a guest VM to measure the monitoring overhead. These benchmarks generate many VM exits into the KVM module and thus can show the worst case of the overhead in terms of KVM module involvement. Note that the benchmark results show the combined overhead of mutual-protection and enforcing the monitoring policy. We configured Netperf and ApacheBench with the same configuration in Section 7.1.

For a fair comparison, we developed a simplified nested virtualization framework. Similar to CloudVisor [33], the unnecessary nested virtualization functionalities are decoupled and removed from the host hypervisor in this simplified framework, in case that they introduce unwanted performance overhead. The host hypervisor is only used as a higher privilege software layer to run the monitor and relies on EPTs to realize event-driven monitoring. Tagged TLB is also used to decrease the overhead of VMM/host hypervisor switches. Finally, we applied the same monitoring policy to this nested virtualization framework and measured the monitoring overhead using Netperf and ApacheBench of the same configuration.

Figure 4 shows the results of Netperf and ApacheBench, which are based on ten identical runs with the standard deviations being up to 1.2% of the average. We used native execution as the baseline and calculated the normalized throughput for both our framework and nested virtualization (*Nested* in the figure). As we see, our approach incurs trivial performance overhead (less than 2%) for each benchmark. In contrast, the throughput with nested virtualization is poor due to the extremely expensive monitor invocation overhead. It is 84% worse than native execution in TCP_STREAM test. It is 63% worse than native execution in TCB_RR test. It is 74% worse than native execution in ApacheBench test.

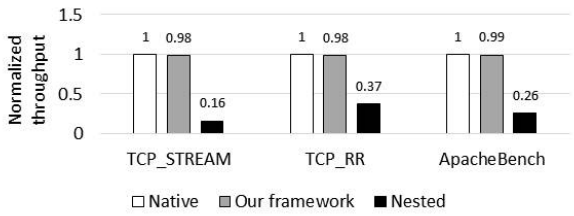

Figure 4. Results for the data integrity monitoring

8. Other Related Work

8.1 Isolation

CloudVisor [33] uses nested virtualization to isolate guest VMs from the untrusted VMM, thus protecting the privacy and integrity of guest VMs' resources. Haven [8] leverages Intel's SGX to isolate critical applications from the cloud platform it runs (the VMM, the VMs and firmware). Our system focuses on VMM monitoring, and neither relies on nested virtualization nor requires special hardware support.

8.2 TCB Reduction and Privilege Restriction.

HyperLock [31] removes the KVM module from the TCB using the combination of address space isolation and software-based fault isolation (SFI). DeHype [32] further demotes the KVM module to user mode and applies the least privilege principle to it. Mycloud [21] and MycloudSEP [22], on the other hand, deprivilege the execution of the host VM in non-root mode and remove it from TCB. Self-service cloud computing [10] splits the host VM into a system-wide domain and per-client administrative domains, so that each domain can be assigned with less privilege.

8.3 VM monitoring

The idea of placing the monitor at the same privilege level was first proposed by SIM [28] for VM monitoring. SIM places the monitor and the untrusted OS into different address space at the same privilege level, and relies on virtualization to guarantee the address space isolation. Regarding whether we can integrate SIM with nested virtualization for VMM monitoring, we have two key observations. *1)* Even if this integration could be realized, SIM still requires expensive address space switching, which would probably become a performance bottleneck when the monitor needs to be frequently invoked for an event-driven monitoring policy. *2)* The integration relies on nested virtualization, which however is rarely deployed in real-world cloud computing: on commodity x86 hardware, nested virtualization imposes unwanted performance penalties on guest VMs [18].

8.4 Side-channel attacks

Based on the literature, ASR could be threatened by side-channel attacks. These channels include memory caches, TLB caches [15], and BTB (or branch predictor) [14]. But ED-monitor is not vulnerable to these attacks. To be specific,

memory caches can be applied to infer physical address space layout. It is still insufficient to infer virtual address of ED-monitor, because the address mapping for ED-monitor is protected. Attacks based on TLB caches require to probe the virtual address space. Such probing will be detected and prevented in our design. Regarding BTB-based attack, it is hard to be applied against ED-monitor, as we prevent BTB collisions by randomizing the addresses at high order bits (bits 43:30).

9. Conclusion

In this paper, we present the first practical event-driven VMM monitoring framework without hardware customization. We place ED-monitor at the same privilege level and in the same address space with the VMM, which achieves extremely lightweight context switching. We propose a unique mutual-protection of IPR and ASR to protect ED-monitor's integrity and the event-driven mechanism. The experimental results demonstrate that our framework only incurs negligible performance overhead, which shows significant performance improvement on previous approaches.

Acknowledgement

This work has been partly supported by National NSF of China under Grant No. 61572248, 61431008, 61321491. Peng Liu was supported by ARO W911NF-13-1-0421(MURI), NSF CNS-1422594, and NSF CNS-1505664.

References

[1] Amazon ec2. `http://aws.amazon.com/ec2`, 2016.

[2] Apachebench. `http://httpd.apache.org`, 2016.

[3] Cve. `http://cve.mitre.org/`, 2016.

[4] Netperf. `http://www.netperf.org/netperf/`, 2016.

[5] A. M. Azab, P. Ning, J. Shah, Q. Chen, and R. Bhutkar. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the ACM conference on Computer and communications security*, 2014.

[6] A.M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N.C. Skalsky. Hypersentry: Enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the ACM conference on Computer and communications security*, pages 38–49, 2010.

[7] M. Backes and S. Nurnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *Proceedings of the conference on USENIX Security Symposium*, 2014.

[8] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, pages 267–283, 2014.

[9] M. Ben-Yehuda, M.D. Day, Z. Dubitzky, M. Factor, N. Har'El, and A. Gordon. The turtles project: Design and implementation of nested virtualization. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–6, 2010.

[10] S. Butt, H.A. Lagar-Cavilla, A. Srivastava, and V. Ganapathy. Self-service cloud computing. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 292–307, 2012.

[11] J. Criswell, N. Dautenhahn, and V. Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 253–264, 2014.

[12] J. Criswell, N. Dautenhahn, and V. Adve. Virtual ghost: protecting applications from hostile operating systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 81–96, 2014.

[13] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 191–206, 2015.

[14] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *IEEE/ACM International Symposium on Microarchitecture*, pages 1–13, 2016.

[15] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space aslr. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 191–205, 2013.

[16] Intel. *Intel Trusted Execution Technology*. February 2011.

[17] Intel. *Intel 64 and IA-32 Architectures Software Developer s Manual: System Programming Guide.* March 2013.

[18] B. Kauer, P. Verissimo, and A. Bessani. Recursive virtual machines for advanced security mechanisms. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*, pages 117–122, 2011.

[19] H. Lee, H. Moon, D. Jang, K. Kim, J. Lee, Y. Paek, and et al. Ki-mon: a hardware-assisted event-triggered monitoring platform for mutable kernel object. In *Proceedings of Proceedings of Usenix Security Symposium*, pages 511–526, 2013.

[20] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with return-less kernels. In *Proceedings of the 5th European conference on Computer systems*, pages 195–208, 2010.

[21] M. Li, W. Zang, K. Bai, M. Yu, and P. Liu. Mycloud: supporting user-configured privacy protection in cloud computing. In *Proceedings of Annual Computer Security Applications Conference*, pages 59–68, 2013.

[22] M. Li, Z. Zha, W. Zang, M. Yu, P. Liu, and K. Bai. Detangling resource management functions from the tcb in privacy-preserving virtualization. In *Proceedings of European Symposium on Research in Computer Security*, pages 310–325, 2014.

[23] Z. Liu, J. Lee, J. Zeng, Y. Wen, Z. Lin, and W. Shi. Cpu transparent protection of os kernel and hypervisor integrity with programmable dram. In *Proceedings of the Annual*

International Symposium on Computer Architecture, pages 392–403, 2013.

[24] L. Mcvoy and C. Staelin. Lmbench: Portable tools for performance analysis. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 279–294, 2000.

[25] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz. Opaque control-flow integrity. In *Proceedings of the Annual Computer Security Applications Conference*, pages 339–348, 2015.

[26] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B.B. Kang. Vigilare: toward snoop-based kernel integrity monitor. In *Proceedings of the ACM conference on Computer and communications security*, pages 28–37, 2012.

[27] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 233–247, 2008.

[28] M. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-vm monitoring using hardware virtualization. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 477–487, 2009.

[29] J. Wang, A. Stavrou, and A.K. Ghosh. Hypercheck: A hardware-assisted integrity monitor. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, pages 158–177, 2010.

[30] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the ACM conference on Computer and communications security*, pages 545–554, 2009.

[31] Z. Wang, C. Wu, M. Grace, and X. Jiang. Isolating commodity hosted hypervisors with hyperlock. In *Proceedings of the ACM european conference on Computer Systems*, pages 127–140, 2012.

[32] C. Wu, Z. Wang, and X. Jiang. Taming hosted hypervisors with (mostly) deprivileged execution. In *Proceedings of Annual Network and Distributed System Security Symposium*, pages 1–16, 2013.

[33] F. Zhang, J. Chen, H. Chen, and B. Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Annual Network and Distributed System Security Symposium*, 2011.

A. Elimination of the Unintended Bytes

An instruction in x86 is composed of several fields, including *Instruction Prefixes*, *Opcode*, *ModR/M*, *SIB*, *Displacement* and *Immediate*. One or several of these fields may casually match a privileged instruction. We will detail the elimination of the unintended bytes for each situation.

Our analysis indicates that the *Instruction Prefixes* and *Opcode* field cannot separately form the unintended bytes of any privileged instruction.

If the *SIB* or the *ModR/M* field of an instruction participates in forming the unintended bytes, we can substitute this instruction using register replacement. An example of the unintended bytes looks like this:

```
48 8b 44 0f 30      mov 0x30(%rdi,%rcx,1),%rax
```

In the example, the *SIB* field (0f) of the mov instruction participates in forming the wrmsr instruction (0f 30). The new substitute for this instruction is:

```
52                  push %rdx
48 89 ca            mov  %rcx,%rdx
48 8b 44 17 30      mov  0x30(%rdi,%rdx,1),%rax
5a                  pop  %rdx
```

In the new substitute (the third instruction), we replace the rcx register used in the original mov instruction to the rdx register. Therefore, we change the *SIB* field and eliminate the unintended bytes of the wrmsr instruction. Since the rdx register would be originally used, we should first store its original value on the stack and restore its value afterwards.

If the *Immediate* field or the *Displacement* field participates in forming a privileged instruction, we can change this field using instruction substitution. The details of instruction substitution for this kind of unintended bytes are well discussed in prior work [20, Section 3.4], and will not be repeated due to the limited paper space.

To reserve space for the new substitute which is usually longer than the original instruction, we overwrite the original instruction with a jmp instruction that detours the control flow to a code stub. In the code stub, the new substitute is executed and then the control flow is transferred back to the subsequent instructions.

In addition, since the length of a privileged instruction is longer than one byte, the unintended bytes would also be formed across two instructions. An example in host Linux kernel code (in the function pci_add_new_bus()) looks like this:

```
75 0f       jne  0xf(%rip)
30 c0       xor  %al,%al
```

In the example, two instructions form the unintended bytes of wrmsr (0f 30). In the new substitute, we only need to insert a nop instruction (this instruction performs no operation in x86) between these two instructions and thus break the unintended bytes. The new substitute for this example is:

```
75 0f       jne  0xf(%rip)
90          nop
30 c0       xor  %al,%al
```

Although the privileged instruction elimination method described above is intended to be comprehensive, we still tested it for over 20 different versions of host Linux kernel equipped with KVM module and 10 versions of Xen. Our experimental results show that only tens of the unintended bytes are found for each version, and all of them can be eliminated using this method.