# Secure IDS Offloading with Nested Virtualization and Deep VM Introspection

Shohei Miyama and Kenichi Kourai

Kyushu Institute of Technology, Iizuka, Fukuoka, Japan
{miyama,kourai}@ksl.ci.kyutech.ac.jp

**Abstract.** To securely execute intrusion detection systems (IDSes) for virtual machines (VMs), IDS offloading with VM introspection (VMI) is used. In semi-trusted clouds, however, IDS offloading inside an untrusted virtualized system does not guarantee that offloaded IDSes run correctly. Assuming a trusted hypervisor, secure IDS offloading has been proposed, but there are several drawbacks because the hypervisor is tightly coupled with untrusted management components. In this paper, we propose a system called *V-Met*, which offloads IDSes outside the virtualized system using *nested virtualization*. Since V-Met runs an untrusted virtualized system in a VM, the trusted computing base (TCB) is separated more clearly and strictly. V-Met can prevent IDSes from being compromised by untrusted virtualized systems and allows untrusted administrators to manage even the hypervisor. Furthermore, V-Met provides *deep VMI* for offloaded IDSes to obtain the internal state of target VMs inside the VM for running a virtualized system. We have implemented V-Met in Xen and confirmed that the performance of offloaded legacy IDSes was comparable to that in traditional IDS offloading.

**Keywords:** VM introspection, Nested virtualization, Insider attacks, IDS, Clouds

## 1 Introduction

In Infrastructure-as-a-Service (IaaS) clouds, users run their services in virtual machines (VMs). They can set up their systems in provided VMs and use them as necessary. As in traditional systems, it is necessary to protect the systems inside VMs from external attackers. For example, intrusion detection systems (IDSes) are useful to monitor the system states, filesystems, and network packets. To prevent IDSes from being compromised by intruders into VMs, *IDS offloading* with *VM introspection (VMI)* has been proposed [6–8, 12]. This technique runs IDSes outside VMs and introspects the internal state of VMs, e.g., the memory, storage, and network. It is difficult that intruders attack IDSes outside VMs.

In semi-trusted clouds, however, it is not guaranteed that offloaded IDSes run correctly. By semi-trusted clouds, we mean that their providers are trusted but some of the system administrators may be untrusted. If untrusted administrators manage virtualized systems for running VMs and offloaded IDSes, offloaded

IDSes can suffer from insider attacks. Malicious administrators can easily disable offloaded IDSes before attacking VMs. If they do not harden virtualized systems sufficiently, even external attackers can interfere with offloaded IDSes using various system vulnerabilities.

In such semi-trusted clouds, secure IDS offloading has been achieved by assuming a trusted hypervisor inside the virtualized system. Even if insiders on the hypervisor attempt to disable offloaded IDSes, their access to the IDSes is prohibited [3,16]. One drawback of this approach is that the hypervisor can be compromised relatively easily by untrusted administrators because it provides rich interfaces to the other management components on top of the hypervisor. Such interfaces become a broad attack surface. Another drawback is that untrusted administrators cannot manage the hypervisor because the integrity of the hypervisor has to be maintained. Consequently, administrators who may be untrusted cannot manage the entire virtualized system. These problems arise from the fact that a trusted hypervisor and untrusted management components are tightly coupled.

In this paper, we propose *V-Met*, which enables offloading IDSes outside the entire virtualized system. V-Met uses *nested virtualization* [2] to run an untrusted virtualized system in a VM called the cloud VM. Since the interface between the cloud VM and its hypervisor is more restricted, V-Met can separate the trusted computing base (TCB) from untrusted parts more clearly and strictly. Thus, V-Met can prevent offloaded IDSes from being compromised by untrusted virtualized systems confined in the cloud VM. In addition, it allows any administrators to completely manage the entire virtualized system including the hypervisor because the hypervisor has to be no longer trusted.

*Deep VMI* is a key to offloaded IDSes to monitor the memory, storage, and network of user VMs inside the cloud VM. For *deep memory introspection*, V-Met first finds the memory of a user VM in that of the cloud VM and then obtains data in the user VM. For *deep network introspection*, it captures packets at both boundaries of a user VM and the virtualized system. For *deep storage introspection*, it accesses a virtual disk of a user VM through the analysis of a virtual disk of the cloud VM. Using Transcall [10] with deep VMI, V-Met can offload even legacy IDSes. We have implemented V-Met in Xen and offloaded several legacy IDSes. Then, we confirmed that the performance was comparable to that of traditional IDS offloading.

The organization of this paper is as follows. Section 2 describes issues of IDS offloading in semi-trusted clouds. Section 3 proposes IDS offloading using nested virtualization and deep VMI. Section 4 describes the implementation details of V-Met. Section 5 reports experimental results for examining the effectiveness of V-Met. Section 6 describes related work and Sect. 7 concludes this paper.

## 2 IDS Offloading in Semi-trusted Clouds

To execute IDSes securely, *IDS offloading* with *VMI* has been proposed [6–8,12]. This technique enables IDSes to run outside their target VMs and monitor the

systems inside the VMs from the outside. Even if attackers intrude into a user VM, they cannot disable offloaded IDSes. IDSes are often offloaded to a privileged VM for managing user VMs, e.g., the *management VM* in Xen. Offloaded IDSes can directly obtain detailed information such as the memory, storage, and networks inside user VMs, using VMI. IDSes in the management VM can map memory pages of target VMs and obtain the system state. They can access disk images of user VMs, which are located in the management VM. Also, they can capture packets from virtual network devices created in the management VM.

Although the management VM running offloaded IDSes is managed by system administrators in clouds, not all system administrators are trusted even if cloud providers are trusted. It is reported that 28% of cybercrimes are caused by insiders [27]. One example of insiders is malicious system administrators, who attack systems actively. For example, a site reliability engineer in Google violated user's privacy in 2010 [33]. Another example is curious but honest system administrators, who may eavesdrop on attractive information that they can easily obtain from user VMs. It is revealed that 35% of system administrators access sensitive information without authorization [5].

In such *semi-trusted clouds*, secure IDS offloading using a trusted hypervisor has been proposed. The hypervisor is a part of a virtualized system, which is managed by administrators. For example, BVMD [25] directly runs IDSes inside a trusted hypervisor and protects them from untrusted administrators. SSC [3] enables offloaded IDSes to run only in user's own administrative VMs and prevents system administrators from interfering with those VMs. RemoteTrans [16] offloads IDSes to trusted remote hosts and remotely performs VMI via the trusted hypervisor.

Unfortunately, such a trusted hypervisor is tightly coupled with the untrusted management VM running management components. Therefore, the approach of using a trusted hypervisor has three drawbacks. First, untrusted administrators in the management VM can compromise the hypervisor relatively easily. The hypervisor provides rich interfaces to the management VM to delegate many management tasks. Such interfaces can be abused using vulnerabilities in specification and implementation and become a broad attack surface. Once the hypervisor is penetrated, attackers can disable or compromise IDSes.

Second, it is not allowed that untrusted administrators manage the trusted hypervisor. If untrusted administrators are given such a privilege, they could even replace the hypervisor with a malicious one. This means that administrators who may be untrusted cannot manage the entire virtualized system. In general, the virtualized system is updated using packages like the other software. Since packages have dependency, the entire virtualized system including the hypervisor is usually updated at once. To enable only the hypervisor to be updated separately, it is necessary to largely change the current management method.

Third, the approach of using a trusted hypervisor is only applicable to specific virtualized systems. To trust only the hypervisor, it is necessary that the hypervisor and the other management components are clearly separated. Examples of such virtualized systems are Xen and Hyper-V. In contrast, the hypervisor can-

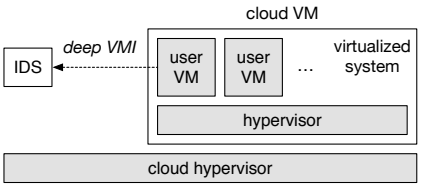



**Fig. 1.** The system architecture of V-Met.

not be separated in KVM because the hypervisor runs inside the host operating system. Although it is possible to trust the entire host operating system, the TCB becomes large because the operating system is much more complex than the hypervisor.

## 3 V-Met

We propose a system called V-Met for enabling secure IDS offloading outside the virtualized system. We assume that some of the system administrators who manage virtualized systems may be untrusted. The hypervisor and the management VM can be abused by untrusted administrators. On the other hand, we assume that cloud providers are trusted. This assumption is widely accepted [16, 17, 30, 31, 35] because a bad reputation is critical for their business. We also assume that the components outside the virtualized system, i.e., V-Met, offloaded IDSes, and hardware, are managed correctly by cloud providers. The integrity of V-Met is guaranteed by remote attestation with TPM at boot time and secure checking with the system management mode (SMM) [1, 29, 34] or Intel TXT and AMD SVM [21] at runtime.

### 3.1 Secure IDS Offloading with Nested Virtualization

V-Met runs IDSes outside a virtualized system using a technique called *nested virtualization* [2]. Traditional approaches rely on trusted hardware outside the virtualized system. For example, Copilot [26] detects tampering with the kernel memory on a PCI add-in card. HyperGuard [29] runs IDSes in the SMM of processors. However, it is difficult to run feature-rich legacy IDSes, which monitor high-level system state, filesystems, and network communication. Nested virtualization enables a virtualized system to run in a VM, which is called the *cloud VM*. Figure 1 illustrates the system architecture of V-Met. V-Met offloads IDSes outside the cloud VM and runs them and the cloud VM on top of the *cloud hypervisor*. The IDSes monitor user VMs inside the cloud VM using *deep VMI*, whose details are described in Sect. 3.2.

V-Met can resolve several issues on IDS offloading in semi-trusted clouds. First, V-Met makes insider attack against offloaded IDSes difficult by running a virtualized system inside the cloud VM. This is because the interface between the cloud VM and the cloud hypervisor is narrower than that between

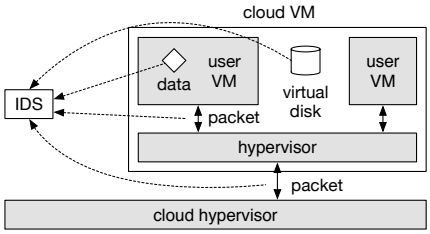



**Fig. 2.** Deep VMI.

the management VM and the hypervisor in the virtualized system. The former hardware-level interface is less vulnerable than the latter rich interface. Therefore, the cloud VM is isolated from the cloud hypervisor more strongly. Second, V-Met allows untrusted administrators to manage the entire virtualized system including the hypervisor. In other words, they can use the traditional management method. This advantage comes from the fact that the responsibility of administrators is more clearly separated at the boundary of virtualization, i.e., between the cloud VM and the cloud hypervisor. Third, V-Met enables clouds to use arbitrary virtualized systems because it does not need to trust specific hypervisors. To achieve this, V-Met provides deep VMI independent of virtualized systems.

In terms of performance, our approach of using nested virtualization is feasible because it is reported that the overhead is 6–8% [2] for common workloads. Special-purpose host hypervisors as used in CloudVisor [35] and TinyChecker [32] can improve the performance of nested virtualization more. Recently, hardware support for nested virtualization has been also added. For example, Intel VMCS Shadowing [11] can eliminate VM exits due to VMREAD and VMWRITE instructions for accessing VMCS. When it is not necessary to run offloaded IDSes, devirtualization [4, 13, 15, 18, 24] could largely reduce the overhead of nested virtualization. This is a technique for temporarily disabling virtualization provided by the hypervisor.

### 3.2 Deep VMI

Deep VMI enables offloaded IDSes to monitor user VMs inside the cloud VM, as depicted in Fig. 2. For *deep memory introspection*, V-Met finds data of a target user VM from the memory of the cloud VM and provides it to offloaded IDSes. Since the memory of the cloud VM contains the memory of multiple user VMs in general, V-Met has to identify the memory of the target user VM and then the target data inside it. To perform this, V-Met executes address translation *three times*. First, it translates a virtual address of target data into a physical address in a user VM using the page tables stored in the user VM. Second, it translates the address into a physical address in the cloud VM using the extended page tables (EPT) for the user VM, which are stored in the hypervisor inside the

cloud VM. Third, it translates the address into a physical address in the entire system using EPT for the cloud VM. In traditional VMI, address translation is only twice.

For *deep network introspection*, V-Met captures packets sent and received by a target user VM at both boundaries of the user VM and the virtualized system. These two methods are called *VM-level* and *system-level* packet capture, respectively. Using VM-level packet capture at the boundary of a user VM, V-Met can monitor sent packets that have not been processed yet by the virtualized system and received ones that have been already processed by that. This means that offloaded IDSes can introspect exact packets sent and received by a user VM. Also, they can introspect packets between user VMs inside the same cloud VM. Using system-level packet capture at the boundary of the virtualized system, on the other hand, V-Met can monitor sent packets that have been already processed by the virtualized system and received ones that have not been processed yet by that. This means that offloaded IDSes can inspect exact communication of a user VM with the outside. Comparing these communication logs of VM-level and system-level packet capture, offloaded IDSes can also detect attacks by insiders in the virtualized system.

For *deep storage introspection*, V-Met supports both local and remote virtual disks. When a virtual disk of a user VM is located in the virtualized system, V-Met first analyzes the virtual disk of the cloud VM and finds a virtual disk of a user VM inside it. The virtual disk of a user VM is stored in the form of a disk image file. Furthermore, V-Met analyzes the found virtual disk and finds data and metadata in it. In contrast, when a virtual disk of a user VM is located in network storage, e.g., for migration support, V-Met shares the virtual disk via the network. Then, it analyzes the virtual disk and finds data and metadata in it.

V-Met identifies a target user VM using a *VM tag* registered by the user VM itself. This is because V-Met cannot securely specify the ID of a user VM, which is managed inside the virtualized system. In V-Met, a user VM registers a unique VM tag to the cloud hypervisor using an *ultracall*. An ultracall is a new mechanism for directly invoking the cloud hypervisor outside the virtualized system. It enables a user VM to securely communicate with the cloud hypervisor without being interfered by the virtualized system. Using the registered VM tag, offloaded IDSes can monitor a target user VM inside the cloud VM uniquely.

### 3.3 Transcall with Deep VMI

Using deep VMI, legacy IDSes can be run outside the virtualized system in cooperation with Transcall [10]. Transcall provides an execution environment for legacy IDSes to introspect a user VM without any modifications. Transcall consists of the system call emulator, the shadow filesystem, and shadow network devices. The system call emulator traps the system calls issued by IDSes and, if necessary, returns information on the kernel from the memory of a user VM, using deep memory introspection. The shadow filesystem provides the same filesystem view as that in a user VM, using deep storage introspection. To achieve

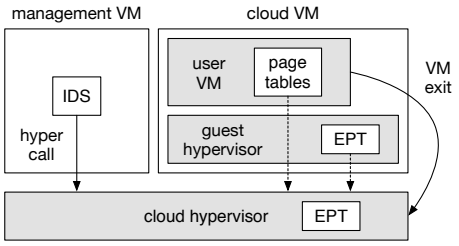



**Fig. 3.** Deep memory introspection.

this, it constructs the shadow proc filesystem, which is a counterpart of the proc filesystem inside a user VM. The proc filesystem provides information on the system such as processes and sockets. The shadow proc filesystem analyzes the memory of a user VM using deep memory introspection and provides files and directories containing system information. In addition, a shadow network device provides a network interface for capturing packets of a user VM, using deep network introspection.

## 4 Implementation

We have implemented V-Met in Xen 4.4.0. In V-Met, the cloud VM and the cloud management VM run on top of the cloud hypervisor. The *cloud management VM* is a VM that provides virtual devices to the cloud VM and has a privilege for introspecting the cloud VM. The cloud VM runs an existing virtualized system, which consists of the hypervisor, the management VM, and user VMs. To distinguish these components from those of V-Met, we call them the *guest hypervisor* and the *guest management VM*, respectively. V-Met assumes that both the cloud VM and user VMs run in full virtualization using Intel VT-x.

### 4.1 Deep Memory Introspection

As described in Sect. 3.2, V-Met executes address translation three times to access target data in a user VM inside the cloud VM from the cloud management VM. Figure 3 shows the flow of deep memory introspection. First, V-Met traverses the page tables inside a user VM to translate a virtual address into a physical address in the user VM (guest physical address). It identifies the page tables by the address of the page directory. This address is stored in the CR3 register of a virtual CPU for the user VM. Although that register is maintained by the guest hypervisor, V-Met cannot trust the state of virtual CPUs stored in the untrusted guest hypervisor.

Therefore, V-Met obtains the value of the CR3 register without relying on the guest hypervisor. It configures the cloud hypervisor so that a VM exit occurs when a user VM attempts to modify the CR3 register. Since a VM exit does not occur by default at this time, we configured VM-execution control fields in the

VMCS of virtual CPUs for a user VM. The VMCS for a user VM is also managed by the untrusted guest hypervisor, but the cloud hypervisor can securely manage it after the VMCS is loaded to a virtual CPU for the cloud VM. When the cloud hypervisor traps access to the control registers including CR3, it first checks whether the access is a write to the CR3 register. If so, it obtains the value that the user VM attempts to write to the register and saves it. Offloaded IDSes in the cloud management VM can obtain the latest value of the CR3 register by issuing a new hypercall to the cloud hypervisor, traverse the page tables, and execute the first address translation.

Second, V-Met traverses EPT inside the guest hypervisor to translate the guest physical address into a physical address in the cloud VM (host physical address). The address of EPT is stored in the VMCS of virtual CPUs for the user VM. When the cloud hypervisor traps access to the CR3 register, it also saves the address of EPT in the VMCS loaded to the virtual CPU. When IDSes in the cloud management VM issue a new hypercall, the cloud hypervisor executes address translation using the saved EPT.

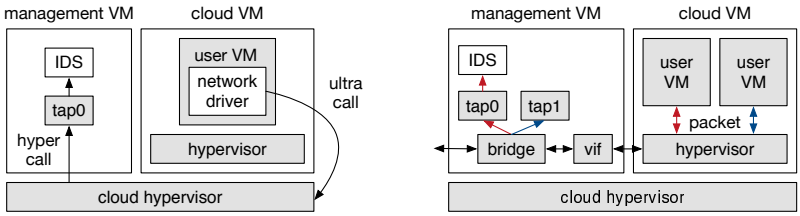
Finally, V-Met translates the host physical into a physical address in the entire system (machine address) using EPT inside the cloud hypervisor. This translation is automatically done by the cloud hypervisor when IDSes issue a hypercall for mapping the memory of the cloud VM.

V-Met assumes that the page tables inside a user VM are protected by the memory isolation technique of CloudVisor [35]. Since CloudVisor restricts access to the memory of a user VM from the virtualized system, insiders cannot tamper with the page tables in the memory of a user VM. Similarly, V-Met protects EPT inside the guest hypervisor by the memory owner tracking technique of CloudVisor. CloudVisor allows only the memory of a user VM to be registered to EPT. Therefore, it is difficult for insiders to modify EPT as they intended.

### 4.2 Deep Network Introspection

For VM-level packet capture, the network driver in a user VM directly passes packets to the cloud hypervisor, as illustrated in Fig. 4(a). Another possible location of this packet capture is a virtual NIC for a user VM. However, V-Met may not be able to correctly capture packets because that virtual NIC runs in the untrusted guest management VM. The other possible method is to trap all the I/O access of a virtual NIC. This method is more secure, but it is more heavyweight and much more difficult to implement. In addition, it is probably impossible for para-virtualized network devices because such devices strongly depend on mechanisms provided by the guest hypervisor.

The network driver in a user VM uses an ultracall to the cloud hypervisor to prevent the virtualized system in the cloud VM from interfering with packet capture. Since it passes the guest physical address of packet data using the ultracall, the cloud hypervisor first translates that address into a host physical address using EPT for the user VM. Then, it copies the packet data to the memory of the cloud hypervisor. V-Met periodically issues a new hypercall for obtaining the saved packets and writes them to a TAP device created for each




(a) VM-level (b) System-level

**Fig. 4.** Deep network introspection.

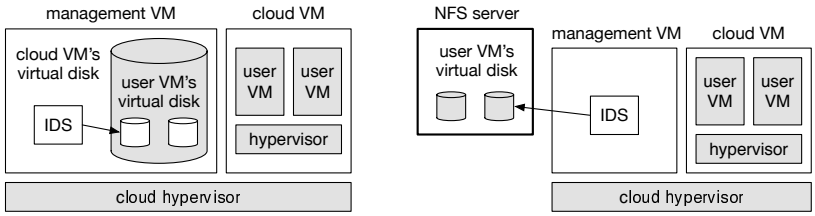
user VM. Offloaded IDSes can capture packets of a user VM from the TAP device.

For system-level packet capture, on the other hand, V-Met can obtain all the packets from the virtual NIC (vif) for the cloud VM, as illustrated in Fig. 4(b). From this virtual NIC, however, the packets sent and received by all the user VMs in the cloud VM are captured. To enable obtaining packets for each user VM separately, V-Met uses the ulog mechanism of ebtables in Linux. ulog is used to pass packets received by the Ethernet bridge to a userland daemon using a netlink socket. After V-Met obtains packets using ulog, it classifies and writes them to a TAP device created for each virtual NIC of user VMs. Offloaded IDSes can capture packets for a user VM from one or some of the TAP devices.

To classify packets without any knowledge of MAC addresses of user VMs, V-Met uses information on sender and receiver devices obtained using ulog. If the sender device of a packet is the virtual NIC of the cloud VM, V-Met automatically creates a TAP device corresponding to the sender's MAC address and writes the packet to the device. In contrast, if the receiver device is the virtual NIC, V-Met writes the packet to a TAP device for the receiver's MAC address. In addition, if such a MAC address is a broadcast address or multicast addresses, V-Met writes a packet to all the TAP devices.

### 4.3 Deep Storage Introspection

When a virtual disk of a user VM is located in a virtual disk of the cloud VM, as in Fig. 5(a), V-Met first mounts the disk image of the cloud VM in the management VM. Then it mounts the disk image of the user VM in the virtual disk of the cloud VM. This seems to be easy, but the reality is not so simple. These mounts need to be done in a read-only manner because the filesystem in a virtual disk is corrupted if its metadata is simultaneously modified by multiple VMs. However, when the filesystem has to be recovered for various reasons, the virtual disk is temporarily mounted in a writable manner to modify its metadata. Since the virtual disk of the cloud VM is mounted in a read-only manner, the disk image of a user VM inside it is not writable. Therefore, it is impossible to modify the virtual disk of a user VM for recovery.




(a) Local (b) Remote

**Fig. 5.** Deep storage introspection.

To solve this dilemma, V-Met creates a snapshot of the disk image of the cloud VM using dm-thin, which is a mechanism for thin provisioning using the device mapper in Linux. When data is read from the snapshot, dm-thin directly returns the corresponding blocks of the disk image. When data is written to the snapshot, dm-thin allocates new blocks in another disk image and writes the data to them. V-Met mounts the snapshot in a writable manner and then mounts the virtual disk of the user VM inside it. Thus, the virtual disk of the user VM can be recovered by temporarily mounting in a writable manner.

On the other hand, when a disk image of a user VM is located in network storage, as in Fig. 5(b), V-Met first mounts a directory where the disk image is stored in an NFS server and then mounts the disk image in it. The directory is also mounted in the guest management VM to run a user VM. Since NFS is designed for sharing files in mind, the directory in the NFS server can be mounted in a writable manner. Therefore, the disk image can be mounted temporarily in a writable manner if its filesystem has to be recovered.

### 4.4 Ultracall

An ultracall directly invokes the cloud hypervisor by executing the vmcall instruction. This instruction is originally used for a hypercall to the hypervisor. When a user VM executes the vmcall instruction in the nested virtualization, the cloud hypervisor first traps the instruction and usually redirects it to the guest hypervisor. Then, the guest hypervisor executes the corresponding hypercall for the user VM. In contrast, the cloud hypervisor in V-Met does not redirect the instruction if a user VM sets a special value to the EAX register. Instead, the cloud hypervisor executes an ultracall for the user VM.

### 4.5 Management of User VMs

In V-Met, the cloud hypervisor manages user VMs inside the cloud VM using VM tags registered by user VMs themselves, the addresses of EPT, and the addresses of the page directories. It binds a VM tag to the address of EPT when the tag is registered. EPT is created at the boot time of a user VM and its address is usually not changed during the execution of the user VM. If EPT

is re-created, the cloud hypervisor detects that at a VM exit caused by CR3 access and changes the binding. Also, the cloud hypervisor binds the address of the current page directory to the VM tag. This binding is changed whenever context switches between processes occur in the user VM. These two bindings have to be removed when the user VM is destroyed, but the detection of VM destruction is our future work.

## 5 Experiments

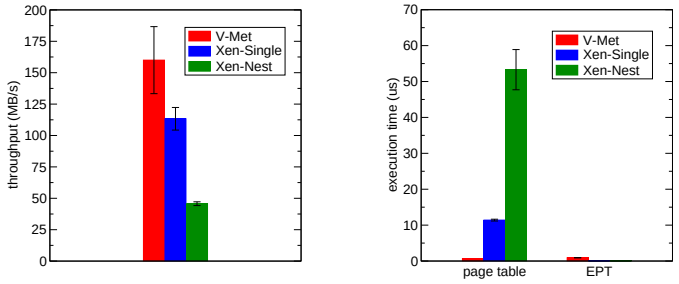
We conducted experiments to examine the effectiveness of V-Met with deep VMI. We used a PC with an Intel Xeon E3-1270v3 processor, 16 GB of DDR3 SDRAM, 2 TB of SATA III HDD, and Gigabit Ethernet. In this PC, we ran Xen 4.4.0 implemented V-Met. For the cloud VM, we assigned two virtual CPUs, 3 GB of memory, and a virtual disk of 40 GB. We ran vanilla Xen 4.4.0 in this cloud VM. For a user VM inside the cloud VM, we assigned one virtual CPU, 1 GB of memory, and a virtual disk of 8 GB. We ran Linux 3.13 in the cloud and guest management VMs and the user VM. For an NFS server, we used a PC with an Intel Xeon X5675 processor, 32 GB of memory, a RAID 5 disk of 3.75 TB, and Gigabit Ethernet. These PCs were connected using a Gigabit Ethernet switch.

For comparison, we used two systems: the traditional, single-level virtualized system without nested virtualization (Xen-Single) and the virtualized system with nested virtualization (Xen-Nest). For Xen-Single, we ran vanilla Xen with the same resource assignment as that inside the cloud VM. Offloaded IDSes were run in the management VM using traditional VMI. For Xen-Nest, we ran vanilla Xen both on top of hardware and in the cloud VM. Offloaded IDSes were run in the guest management VM inside the cloud VM using traditional VMI.

### 5.1 Performance of Deep VMI

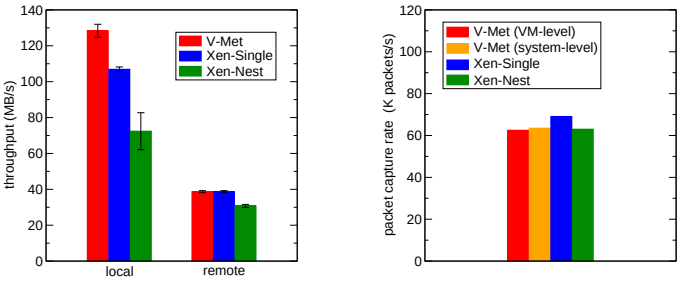
We measured the performance of VMI in the three systems. First, we examined the performance of memory introspection by reading data in the memory of the user VM. The benchmark tool repeated translating virtual addresses of the user VM, mapping its memory pages, and copying the page contents to measure the throughput. Surprisingly, the throughput in V-Met was 41% higher than that in Xen-Single, as shown in Fig. 6(a).

To clarify why the throughput in V-Met was the highest, we measured the execution time of the hypercalls used for address translation. Figure 6(b) shows the time needed for two new hypercalls. In V-Met, it took 0.64 $\mu s$ to execute the hypercall for obtaining the address of the page directory of the user VM. In Xen-Single, it took 11 $\mu s$ because a general-purpose hypercall for obtaining all the state of a virtual CPU was used. The execution time in Xen-Nest was much longer due to the overhead of nested virtualization. Although only V-Met needs the other hypercall for address translation using EPT, which took 0.92 $\mu s$, the hypercall for the page directory was a dominant factor.




(a) Throughput (b) Hypercalls

**Fig. 6.** The performance of memory introspection.

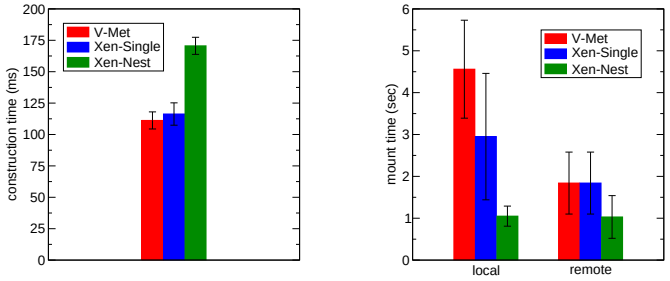



(a) Storage introspection (b) Network introspection

**Fig. 7.** The performance of storage and network introspection.

Next, we examined the performance of storage introspection using the IOzone 3.465 benchmark [23]. In this experiment, we created a file of 1 GB in the user VM and measured the throughput of sequentially reading the file using VMI. We flushed the page cache in the cloud management VM, the guest management VM, and the NFS server every run. Figure 7(a) shows the results when we used virtual disks located in the cloud VM and the NFS server. When using a local virtual disk, the throughput in V-Met was 20% higher than that in Xen-Single although V-Met has to access two virtual disks of the cloud VM and the user VM in turn. According to our analysis, this is because read-ahead for the two virtual disks was more effective than that for only one virtual disk used in Xen-Single. The performance in Xen-Nest largely degraded due to increasing the overhead of storage virtualization for the cloud VM. When using a remote virtual disk, V-Met and Xen-Single accessed the virtual disk in exactly the same manner and consequently the throughput was the same. However, the performance was much lower than using a local virtual disk.

Finally, we examined the performance of network introspection. We transferred 1470-byte UDP datagrams in 500 Mbps to the user VM using iperf 2.0.5 [9] and captured these packets with tcpdump. The maximum rate of packet capture was shown in Fig. 7(b). The rate in Xen-Single was the highest, but the performance degradation in V-Met was only 10% and 8% in VM-level and system-level




(a) Shadow proc filesystem (b) Mount

**Fig. 8.** The initialization time of Transcall.

packet capture, respectively. The overhead of VM-level packet capture comes from writing packets to a TAP device via the cloud hypervisor, while that of system-level packet capture is caused by ebtables. The performance in Xen-Nest was almost the same as that in V-Met.

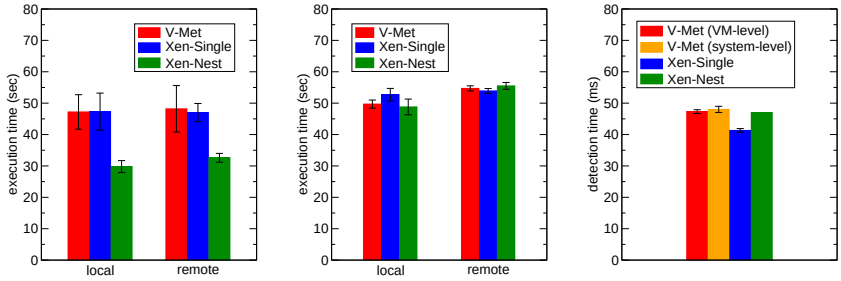
### 5.2 Performance of Offloaded IDSes

We examined the performance of legacy IDSes offloaded with Transcall. Since we need to run Transcall before executing IDSes, we first measured the initialization time of Transcall. Transcall constructs the shadow proc filesystem for the user VM and mounts its virtual disk. To construct the filesystem, Transcall gathers information on the operating system using memory introspection. As shown in Fig. 8(a), the construction time in V-Met was only 5 ms shorter than that in Xen-Single although the performance of memory introspection was largely different, as shown in Fig. 6(a). This is because Transcall caches the results of address translation and the portion of memory introspection was relatively small. Figure 8(b) shows the time needed for mounting the virtual disk of the user VM. For a local virtual disk, V-Met took a long time because it had to mount two virtual disks of the cloud VM and the user VM. For a remote virtual disk, the mount time was much shorter.

After Transcall was initialized, we measured the execution time of chkrootkit 0.50 [22], which is an IDS for detecting installed rootkits by inspecting processes, files, and sockets. Figure 9(a) shows the results for local and remote virtual disks. In both storage configurations, the execution time in V-Met was almost the same as that in Xen-Single.

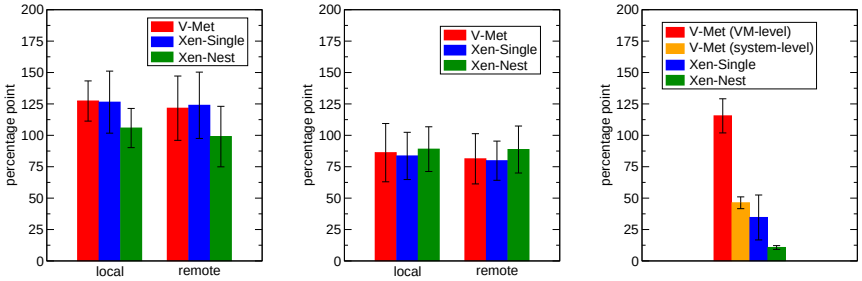
Next, we measured the execution time of Tripwire 2.4 [14], which is an IDS for checking the integrity of filesystems. Figure 9(b) shows the results. For a local virtual disk, the execution time in V-Met was 6% shorter than that in Xen-Single, which came from higher performance of storage introspection. For a remote virtual disk, the execution time in three systems was almost the same.

Finally, we performed a TCP port scan against the user VM using nmap 6.40 [20] and measured the time for detecting this attack. The detection time is




(a) chkrootkit (b) Tripwire (c) Snort

**Fig. 9.** The performance of offloaded IDSes.




(a) chkrootkit (b) Tripwire (c) Snort

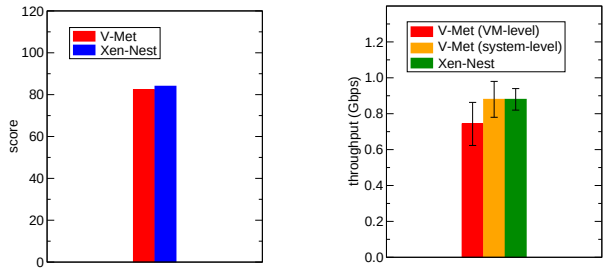
**Fig. 10.** The increases in CPU utilization during running offloaded IDSes.

from when we started a port scan until Snort 2.9.8.3 [28] detected it. Figure 9(c) shows the results. In V-Met, the detection time increased only by 6 or 7 ms, compared with Xen-Single. When using VM-level packet capture, this delay was caused by passing packets from the user VM to the cloud management VM via the cloud hypervisor. When using system-level packet capture, the overhead of the packet classifier led to the detection delay.

### 5.3 Overhead

We measured the increase in CPU utilization of the entire system during the execution of offloaded IDSes. While we ran chkrootkit and Tripwire, CPU utilization increased only in the (cloud) management VM, where IDSes were offloaded, for V-Met and Xen-Single. The increase was almost the same, as shown in Fig. 10(a) and Fig. 10(b). In Xen-Nest, CPU utilization increased in both the cloud and guest management VMs. The reason why the increase was smaller is that Xen-Nest could not utilize CPUs effectively due to nested virtualization.

For Snort, we compared CPU utilization when we transferred 1470-byte UDP datagrams in 100 Mbps with and without Snort. For VM-level packet capture in V-Met, the CPU utilization of the cloud management VM increased largely. Although the user VM issued ultracalls many times, its CPU utilization did not




(a) UnixBench (b) iperf

**Fig. 11.** The overhead of a user VM.

increase. For system-level packet capture, in contrast, CPU utilization increased only by 12% point, compared with Xen-Single.

Next, we examined the overhead of causing VM exits for enabling deep memory introspection. While these VM exits are not caused by default, V-Met needs to trap access to the CR3 register. For V-Met and Xen-Nest, we ran UnixBench 5.1.3 [19] in the user VM. Figure 11(a) shows the UnixBench scores and the performance degradation due to VM exits were only 2%.

Finally, we examined the impact of deep network introspection on network performance of the user VM. We measured the TCP throughput of the user VM using iperf. As shown in Fig. 11(b), VM-level packet capture degraded the throughput by 16%. In contrast, there was no performance degradation in system-level packet capture.

## 6 Related Work

Several systems enable secure IDS offloading by using trusted hypervisors. A self-service cloud (SSC) computing platform [3] provides users with privileged VMs called service domains (SDs) to monitor their own VMs. SDs can monitor the memory of target VMs, disk blocks accessed by VMs, and system calls issued by them. Even cloud administrators cannot disable IDSes in SDs. However, a VM called DomB has to be trusted in addition to the hypervisor.

RemoteTrans [16] enables IDSes to be offloaded to trusted remote hosts outside a semi-trusted cloud and to securely monitor user VMs in the cloud via the network. IDSes offloaded to remote hosts communicate with trusted hypervisors using encryption to obtain memory contents, network packets, and disk blocks of user VMs. However, instead of cloud providers, users themselves have to manage offloaded IDSes and remote hosts running offloaded IDSes are part of the TCB.

Using hardware support has been proposed for secure IDS offloading. These systems allow untrusted cloud administrators to manage the entire virtualized system except for hardware. Copilot [26] can monitor the integrity of the kernel memory by using a PCI add-in card inserted in a target host. The Copilot monitor on the card obtains the kernel text and jump tables from memory by

DMA and calculates its hash. It sends the results of integrity checking to a remote host via a dedicated network. Due to hardware limitation, it is difficult to run legacy IDSes.

HyperGuard [29] and HyperCheck [34] enable IDSes to securely monitor the integrity of the hypervisor using SMM. In SMM, the CPU can securely execute code in System Management RAM (SMRAM), which cannot be accessed in the normal mode. However, all the regular tasks are suspended while an IDS is running in SMM. Another drawback is that SMM is much slower than the normal mode. Running the whole IDS in SMM suffers from larger overhead. In addition, it is not easy to execute various IDSes in SMM because developers need to modify BIOS.

HyperSentry [1] allows a measurement agent inside the hypervisor to be securely executed using SMM even if the hypervisor has been compromised. The handler running in SMM is invoked via Intelligent Platform Management Interface (IPMI), which is an out-of-band communication channel with a remote host. Then the handler verifies the agent, disables interrupts, and runs the agent for collecting the detailed information on the hypervisor. The measurement output is attested by the remote host. One drawback is that the agent cannot run simultaneously with the other tasks.

Flicker [21] is an infrastructure for executing security-sensitive code using the hardware support such as Intel TXT and AMD SVM. When such code needs to be executed, Flicker suspends the current execution environment, securely executes the code using late launch, and resumes the previous execution environment. Late launch enables code execution without interferences by the attackers. However, it also stops all CPU cores other than the one used by the executed code. While the security-sensitive code is running, the other applications cannot be running.

## 7 Conclusion

In this paper, we proposed a system called V-Met, which enables IDS offloading outside the virtualized system using nested virtualization. V-Met runs an untrusted virtualized system in a VM and allows offloaded IDSes to securely monitor user VMs inside it using deep VMI. Such clear separation of the TCB can prevent IDSes from being attacked by untrusted virtualized systems. Also, it allows untrusted administrators to manage the entire virtualized system including the hypervisor as traditionally done. We have ported Transcall for offloading the legacy IDSes to V-Met and confirmed that the overhead is comparable to the traditional IDS offloading.

One of our future work is to automatically and securely identify the MAC addresses and the virtual disk used by a user VM. In the current implementation, we assume that these are known in advance. We are also planning to monitor components other than user VMs, e.g., the hypervisor and the management VM in the virtualized system. Another direction is to run other virtualized systems such as KVM. We believe that this is not difficult thanks to the design of V-Met.

## Acknowledgment

This work was partially supported by JSPS KAKENHI Grant Number JP16K00101.

## References

1. Azab, A., Ning, P., Wang, Z., Jiang, X., Zhang, X., Skalsky, N.: HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity. In: Proc. ACM Conf. Computer and Communications Security. pp. 38–49 (2010)
2. Ben-Yehuda, M., Day, M.D., Dubitzky, Z., Factor, M., Har'El, N., Gordon, A., Liguori, A., Wasserman, O., Yassour, B.A.: The Turtles Project: Design and Implementation of Nested Virtualization. In: Proc. USENIX Symp. Operating Systems Design and Implementation. pp. 423–436 (2010)
3. Butt, S., Lagar-Cavilla, H.A., Srivastava, A., Ganapathy, V.: Self-service Cloud Computing. In: Proc. ACM Conf. Computer and Communications Security. pp. 253–264 (2012)
4. Chen, H., Chen, R., Zhang, F., Zang, B., Yew, P.C.: Mercury: Combining Performance with Dependability Using Self-virtualization. In: Proc. IEEE Int. Conf. Parallel Processing (2007)
5. CyberArk Software: Global IT Security Service (2009)
6. Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J., Lee, W.: Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In: Proc. IEEE Symp. Security and Privacy. pp. 297–312 (2011)
7. Fu, Y., Lin, Z.: Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. In: Proc. IEEE Symp. Security and Privacy. pp. 586–600 (2012)
8. Garfinkel, T., Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection. In: Proc. Network and Distributed Systems Security Symp. pp. 191–206 (2003)
9. Gates, M., Warshavsky, A.: iperf2. `https://sourceforge.net/projects/iperf2/`
10. Iida, T., Kourai, K.: Transcall. `http://www.ksl.ci.kyutech.ac.jp/oss/transcall/`
11. Intel Corp.: 4th Generation Intel Core vPro Processors with Intel VMCS Shadowing (2013)
12. Jiang, X., Wang, X., Xu, D.: Stealthy Malware Detection through VMM-based "Out-of-the-box" Semantic View Reconstruction. In: Proc. ACM Conf. Computer and Communications Security. pp. 128–138 (2007)
13. Keller, E., Szefer, J., Rexford, J., Lee, R.B.: NoHype: Virtualized Cloud Infrastructure Without the Virtualization. In: Proc. ACM/IEEE Int. Symp. Computer Architecture. pp. 350–361 (2010)
14. Kim, G., Spafford, E.: The Design and Implementation of Tripwire: A File System Integrity Checker. In: Proc. ACM Conf. Computer and Communications Security. pp. 18–29 (1994)
15. Kooburat, T., Swift, M.: The Best of Both Worlds with On-demand Virtualization. In: Proc. USENIX Workshop on Hot Topics in Operating Systems (2011)
16. Kourai, K., Juda, K.: Secure Offloading of Legacy IDSes Using Remote VM Introspection in Semi-trusted Clouds. In: Proc. IEEE Int. Conf. Cloud Computing. pp. 43–50 (2016)

17. Li, C., Raghunathan, A., Jha, N.K.: Secure Virtual Machine Execution under an Untrusted Management OS. In: Proc. IEEE Int. Conf. Cloud Computing. pp. 172–179 (2010)
18. Lowell, D.E., Saito, Y., Samberg, E.J.: Devirtualizable Virtual Machines Enabling General, Single-node, Online Maintenance. In: Proc. ACM Int. Conf. Architectural Support for Programming Languages and Operating Systems. pp. 211–223 (2004)
19. Lucas, K.: UnixBench. https://github.com/kdlucas/byte-unixbench
20. Lyon, G.: Nmap – Free Security Scanner for Network Exploration & Security Audits. http://nmap.org/
21. McCune, J., Parno, B., Perrig, A., Reiter, M., Isozaki, H.: Flicker: An Execution Infrastructure for TCB Minimization. In: Proc. European Conf. Computer Systems. pp. 315–328 (2008)
22. Murilo, N., Steding-Jessen, K.: chkrootkit – Locally Checks for Signs of a Rootkit. http://www.chkrootkit.org/
23. Norcott, W.D.: IOzone Filesystem Benchmark. http://www.iozone.org/
24. Omote, Y., Shinagawa, T., Kato, K.: Improving Agility and Elasticity in Bare-metal Clouds. In: Proc. ACM Int. Conf. Architectural Support for Programming Languages and Operating Systems. pp. 145–159 (2015)
25. Oyama, Y., Giang, T., Chubachi, Y., Shinagawa, T., Kato, K.: Detecting Malware Signatures in a Thin Hypervisor. In: Proc. ACM Symp. Applied Computing. pp. 1807–1814 (2012)
26. Petroni, Jr., N., Fraser, T., Molina, J., Arbaugh, W.: Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor. In: Proc. USENIX Security Symp. (2004)
27. PwC: US Cybercrime: Rising Risks, Reduced Readiness (2014)
28. Roesch, M.: Snort – Lightweight Intrusion Detection for Networks. In: Proc. USENIX System Administration Conf. (1999)
29. Rutkowska, J., Wojtczuk, R.: Preventing and Detecting Xen Hypervisor Subversions. Black Hat USA (2008)
30. Santos, N., Gummadi, K.P., Rodrigues, R.: Towards Trusted Cloud Computing. In: Proc. Workshop on Hot Topics in Cloud Computing (2009)
31. Tadokoro, H., Kourai, K., Chiba, S.: Preventing Information Leakage from Virtual Machines' Memory in IaaS Clouds. IPSJ Online Trans. 5, 156–166 (2012)
32. Tan, C., Xia, Y., Chen, H., Zang, B.: TinyChecker: Transparent Protection of VMs against Hypervisor Failures with Nested Virtualization. In: Proc. IEEE/IFIP Int. Workshop on Dependability of Clouds, Data Centers and Virtual Machine Technology (2012)
33. TechSpot News: Google Fired Employees for Breaching User Privacy. http://www.techspot.com/news/40280-google-fired-employees-for-breaching-user-privacy.html (2010)
34. Wang, J., Stavrou, A., Ghosh, A.: HyperCheck: A Hardware-assisted Integrity Monitor. In: Proc. Int. Symp. Recent Advances in Intrusion Detection. pp. 158–177 (2010)
35. Zhang, F., Chen, J., Chen, H., Zang, B.: CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In: Proc. ACM Symp. Operating Systems Principles. pp. 203–216 (2011)