Secure Offloading of Legacy IDSes Using Remote VM Introspection in Semi-trusted Clouds

Kenichi Kourai
Department of Creative Informatics
Kyushu Institute of Technology
Fukuoka, Japan
`kourai@ci.kyutech.ac.jp`

Kazuki Juda
Department of Creative Informatics
Kyushu Institute of Technology
Fukuoka, Japan
`kazuki@ksl.ci.kyutech.ac.jp`

***Abstract*—In Infrastructure-as-a-Service (IaaS) clouds, intrusion detection systems (IDSes) increase their importance. To securely detect attacks against virtual machines (VMs), IDS offloading with VM introspection (VMI) has been proposed. In semi-trusted clouds, however, it is difficult to securely offload IDSes because there may exist insiders such as malicious system administrators. First, secure VM execution cannot coexist with IDS offloading although it has to be enabled to prevent information leakage to insiders. Second, offloaded IDSes can be easily disabled by insiders. To solve these problems, this paper proposes *IDS remote offloading* with *remote VMI*. Since IDSes can run at trusted remote hosts outside semi-trusted clouds, they cannot be disabled by insiders in clouds. Remote VMI enables IDSes at remote hosts to introspect VMs via the trusted hypervisor inside semi-trusted clouds. Secure VM execution can be bypassed by performing VMI in the hypervisor. Remote VMI preserves the integrity and confidentiality of introspected data between the hypervisor and remote hosts. The integrity of the hypervisor can be guaranteed by various existing techniques. We have developed *RemoteTrans* for remotely offloading *legacy* IDSes and confirmed that RemoteTrans could achieve surprisingly efficient execution of legacy IDSes at remote hosts.**

***Keywords*-IDS, insider attacks, clouds, VMs, VMI**

I. Introduction

In Infrastructure-as-a-Service (IaaS) clouds, users run their services in virtual machines (VMs). They can set up the 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 [1]–[4]. This technique runs IDSes outside VMs and introspects the internals of VMs, e.g., the memory, storage, and network. It is difficult that intruders attack IDSes outside VMs.

On the other hand, VMs in *semi-trusted clouds* can suffer from insider attacks. In this paper, semi-trusted clouds mean that cloud providers are trusted but some of the system administrators may be untrusted. To reduce the risk of insider attacks, various mechanisms for *secure VM execution* in semi-trusted clouds have been proposed [5]–[8]. These mechanisms prevent information leakage to insiders, e.g., by encrypting the memory and storage of VMs or restricting access to VMs in the trusted hypervisor. However, it is difficult to securely offload IDSes inside such semi-trusted clouds. First, secure VM execution cannot coexist with IDS offloading because offloaded IDSes need to analyze the internals of VMs. Second, IDSes offloaded inside clouds can be easily disabled by insiders.

In this paper, we propose *IDS remote offloading* using *remote VMI*. This technique enables IDSes to run at trusted remote hosts outside semi-trusted clouds. Therefore IDSes are prevented from being disabled by insiders in clouds. Remote VMI is an enabling technology for IDS remote offloading. It is initiated by remote hosts and introspects VMs using a minimal *VMI engine* in the trusted hypervisor inside semi-trusted clouds. The VMI engine can bypass secure VM execution provided by the hypervisor. Remote VMI preserves the integrity and confidentiality of introspected data between the hypervisor and remote hosts. The integrity of the hypervisor can be guaranteed by various existing techniques [9]–[12].

We have developed a system for achieving remote offloading of legacy IDSes, called *RemoteTrans*. For remote memory introspection, the VMI engine introspects the memory of target VMs on demand. For remote network introspection, it analyzes the communication between network drivers and devices and captures the packets to/from target VMs. For remote storage introspection, remote hosts share protected storage with target VMs. Using remote VMI, legacy IDSes can be run at remote hosts in cooperation with Transcall [13]. According to our experiments, remotely offloaded legacy IDSes were surprisingly efficient when network delay was small.

The organization of this paper is as follows. Section II describes issues of IDS offloading in semi-trusted clouds. Section III proposes IDS remote offloading with remote VMI. Section IV explains the implementation details of RemoteTrans. Section V reports experiments for examining the effectiveness of IDS remote offloading. Section VI describes related work and Section VII concludes this paper.

II. IDS Offloading in Semi-trusted Clouds

To execute IDSes securely, *IDS offloading* with *VMI* has been proposed [1]–[4]. This technique enables IDSes to run outside their target VM and monitor the system inside the VM from the outside. Even if attackers intrude into a VM, they cannot disable offloaded IDSes. IDSes are often offloaded to a privileged VM, which is called the *management VM*. Offloaded IDSes can directly obtain detailed information inside VMs, e.g., the memory, storage, and networks, using VMI. IDSes in the management VM can map memory pages of target VMs and read the contents. They can access disk images of VMs, which are located in the management VM. Also, they can capture packets from virtual network devices created in the management VM because all the packets are sent via the devices. Using such information, even legacy IDSes can be offloaded [4], [13].

However, VMI can be abused by insiders in *semi-trusted clouds*. This results in leaking sensitive information inside VMs. It is reported that 28% of cyber crimes is caused by insiders [14]. 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 [15]. Another example is *curious but honest* system administrators, who may eavesdrop on attractive information that they can easily obtain from VMs. It is revealed that 35% of system administrators access sensitive information without authorization [16].

To reduce the risk of insider attacks, many researchers have proposed mechanisms for *secure VM execution*. The secure runtime environment [5] and VMCrypt [7] prevent information leakage from the memory of VMs. These systems encrypt the memory of VMs only when cloud administrators access it from the management VM, while they do not when the VMs access it. This encryption is done in the trusted hypervisor underlying all the VMs including the management VM. SSC [8] prevents the management VM from accessing users' VMs and allows only users' administrative VMs to access them using the trusted hypervisor.

In semi-trusted clouds, it is difficult to securely offload IDSes. First, secure VM execution cannot coexist with IDS offloading. Offloaded IDSes need to access the memory of VMs from the outside to introspect VMs. If the memory of VMs is encrypted or its access is restricted, offloaded IDSes cannot run. If secure VM execution is not enabled, insiders can access the internals of VMs as well as offloaded IDSes and information leakage cannot be prevented. Second, insiders can easily disable offloaded IDSes. They can stop IDSes offloaded to the management VM and tamper with their configurations. If they intrude into target VMs after that, IDSes could not detect it. Even if IDSes run in VMs protected by secure VM execution, insiders can attack such VMs in various ways. For example, it is possible to mount attacks exploiting vulnerabilities of the systems inside the VMs.

To avoid these problems, IDSes can be offloaded to more secure execution environments, not the management VM abused by insiders. BVMD [17] executes a malware detector in the hypervisor and compares the contents of data I/O with malware signatures. HyperGuard [11] runs an integrity checker in BIOS to check the integrity of the hypervisor in System Management Mode (SMM) of x86 processors. Flicker [10] runs a rootkit detector in an isolated environment using Intel TXT and AMD SVM. However, it is difficult to run legacy IDSes such as chkrootkit, Tripwire, and Snort in these systems because the capabilities of used execution environments are limited. Even if legacy IDSes can be run, they make the trusted computing base (TCB) inside clouds much larger and this results in increasing the attack surface. Also, it is problematic to securely update policy files such as signature files of Snort inside clouds.

III. IDS Remote Offloading

In this paper, we propose *IDS remote offloading*, which enables running legacy IDSes at remote hosts outside semi-trusted clouds.

A. Assumptions and Threat Model

We assume that cloud providers are trusted. This assumption is widely accepted [5]–[7], [18] because a bad reputation is critical for their business. Therefore we trust hardware used in clouds. The integrity of the hypervisor on top of trusted hardware is guaranteed by various techniques. At boot time, remote attestation with TPM enables trusted cloud providers, external trusted authorities, or users to check the integrity. At runtime, integrity checking can be securely done by using hardware such as PCI add-in cards [9], SMM [11], [12], [19], or Intel TXT and AMD SVM [10]. We assume that such infrastructure is securely maintained by cloud providers.

On the other hand, we assume that some of the system administrators may be untrusted. The management VM can be abused by malicious or honest but curious system administrators. Such insiders can take the root privilege in the management VM and even modify its operating system kernel. However, they have access rights only for the management VM and cannot disable the protection mechanisms of the hypervisor.

We assume that remote hosts are trusted because they are owned by users. Used hosts can be physical hosts prepared by users or VMs in private clouds of users' organization. In any case, they run in trusted execution environments outside semi-trusted public clouds.

B. Secure IDS Offloading Using Remote VMI

Fig. 1 illustrates the system architecture for IDS remote offloading. Since IDSes run at trusted remote hosts, they

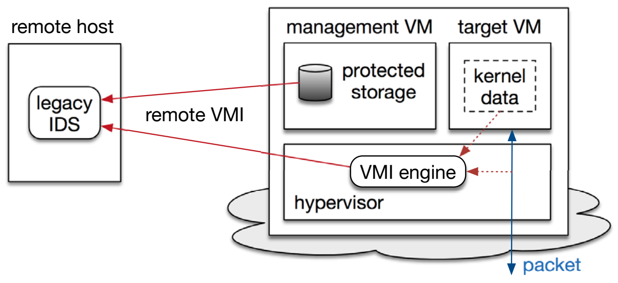


Figure 1. IDS remote offloading with remote VMI.

cannot be disabled by insiders in clouds. Even if such insiders mount DoS attacks by preventing the transfers of introspected data, remote hosts can notice that easily. In addition, it is easy and secure to update policy files for IDSes. An enabling technology for IDS remote offloading is *remote VMI*. Remote VMI is initiated by remote hosts and introspects VMs using a minimal *VMI engine* inside clouds. The VMI engine securely runs in the trusted hypervisor. Since it provides only basic mechanisms, it does not need to update policy files. Using the VMI engine, secure VM execution is bypassed because it is usually provided by the hypervisor [5], [7], [8]. As a result, IDS remote offloading can coexist with secure VM execution. Remote VMI also preserves the integrity and confidentiality of introspected data between the VMI engine and remote hosts.

For *remote memory introspection*, the VMI engine introspects the memory of target VMs and sends memory contents to remote hosts on demand. Although the memory of VMs can be encrypted or isolated by secure VM execution, the VMI engine inside the hypervisor can access the unencrypted memory. To access the memory of VMs from the hypervisor, the VMI engine translates virtual addresses used by VMs into physical addresses used by the hypervisor. It introspects the page tables inside VMs and looks up the nested page tables in the hypervisor. To preserve the integrity and confidentiality of requests and responses from/to remote hosts, the VMI engine encrypts introspected data and calculates its message authentication code (MAC). Remote hosts verify the MAC and decrypt received data. This mechanism can also prevent the VMI engine from being abused by users who do not have the encryption key.

For *remote network introspection*, the VMI engine captures packets to/from target VMs and sends them to remote hosts. It captures packets between target VMs and the virtual network devices created in the management VM. This is not so easy because the communication between target VMs and virtual devices is done directly. To capture packets, the VMI engine intercepts and analyzes the communication. It can capture packets sent from target VMs before the packets may be tampered with in the management VM. In contrast, it can capture packets received by target VMs after the packets may be tampered with. Thanks to the introspection in the hypervisor, the packets between VMs in the same host can be also captured. To preserve the integrity of transferred packets, the VMI engine calculates a MAC for them and remote hosts verify the MAC.

For *remote storage introspection*, unlike memory and network introspection, remote hosts share storage with target VMs directly, not via the hypervisor. This is because the hypervisor cannot read the entire disk images easily. The disk images of target VMs are located in the management VM and the hypervisor can capture only data read and written by target VMs. To protect storage from the management VM, storage is encrypted and every block is hashed by secure VM execution or the operating systems in target VMs. As such, the management VM securely provides the protected storage to remote hosts as network storage. Remote hosts decrypt it and check its integrity.

Using remote VMI, legacy IDSes can be run at remote hosts in cooperation with Transcall [13]. Transcall provides an execution environment for legacy IDSes to introspect a VM without any modifications. Transcall consists of the system call emulator and the shadow filesystem. The system call emulator traps the system calls issued by IDSes and obtains necessary information on the kernel from the memory of a VM. It also provides a network interface for network introspection. The shadow filesystem provides the same filesystem view as that in a VM. To achieve this, it generates the proc filesystem, which provides information on the processes and the networks in a VM. The shadow proc filesystem analyzes the memory of a VM and provides necessary information as pseudo files.

IV. Implementation

To achieve IDS remote offloading with remote VMI, we have developed a system called RemoteTrans using Xen 4.1.3. RemoteTrans consists of a runtime at a remote host, a server in the management VM, and a VMI engine in the hypervisor. The RemoteTrans runtime includes Transcall [13] for offloading legacy IDSes. The RemoteTrans server is used for relaying the communication between the runtime and the VMI engine because the Xen hypervisor does not have the capability of network communication. Note that we do not trust the RemoteTrans server.

A. Remote Memory Introspection

Whenever an IDS at a remote host attempts to read the memory data in a target VM, it sends a request to the RemoteTrans runtime, as illustrated in Fig. 2. The request consists of the virtual address and size of the data. The runtime forwards the request to the RemoteTrans server in the management VM via a network. Next, the server invokes the VMI engine in the hypervisor using a hypercall. The VMI engine traverses the page tables inside a target VM and translates the specified virtual address into a guest physical address. Then it translates the guest physical address into a

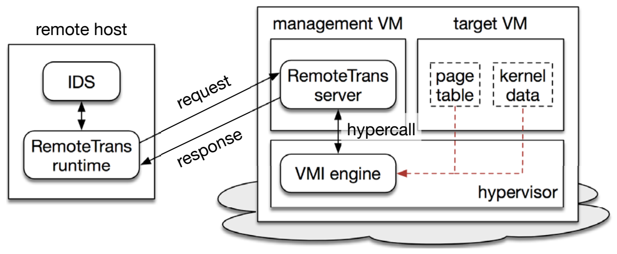


Figure 2. Memory introspection in RemoteTrans.

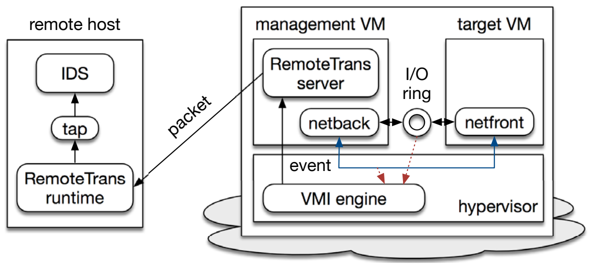


Figure 3. Network introspection in RemoteTrans.

host physical address. After that, it reads data of the specified size from the host physical address and returns it to the RemoteTrans server. Finally, the data is transferred to the RemoteTrans runtime as a response.

To prevent the requests and responses from being tampered with, the RemoteTrans runtime checks the integrity. First, the RemoteTrans runtime includes a random nonce in each request to prevent replay attacks. Before the VMI engine returns obtained data to the RemoteTrans server, it encrypts the data with AES-CBC to preserve confidentiality. Then it calculates a MAC from the virtual address, the size, and the nonce in a request, the obtained data, and a secret key in the VMI engine. When the RemoteTrans runtime receives the response, it first decrypts the data. Then it calculates a MAC from the saved virtual address, size, and nonce, the decrypted data, and the same secret key as the VMI engine. If the calculated MAC and the MAC included in the response do not match, that means that the request and/or response have been tampered with.

The RemoteTrans runtime locally caches obtained memory contents of a target VM. It does not send requests for the data that has been obtained once and returns cached data to an IDS. Although the size of data is often small, e.g., four bytes, the RemoteTrans runtime obtains memory pages including the required data at once for efficiency. Since most of IDSes run periodically, it is acceptable to monitor a little bit old data in the cache. To keep fresh data in the cache as much as possible, the RemoteTrans runtime flushes the cache periodically, e.g., after an IDS completes to check the system once. The freshness of the cache and the performance of IDSes are tradeoff.

B. Remote Network Introspection

To securely capture all the packets from/to a target VM, the VMI engine in the hypervisor monitors the communication between the management VM and a target VM, as shown in Fig. 3. Packets from a target VM are transferred to the management VM and are sent to the outside or the other VMs in the same host via the network bridge. In contrast, packets to a target VM are first received by the management VM and are then transferred to a target VM. The recent operating system often uses a paravirtual network driver named *netfront* for efficiency even if a VM is fully virtualized. The netfront driver communicates with the *netback* driver in the management VM using ring buffers called *I/O rings*. After these drivers write requests or responses to I/O rings, they send events to the other end via the hypervisor using *event channels*.

The VMI engine monitors events sent between the netfront and netback drivers and captures packets via I/O rings. When a target VM sends a packet, the netfront driver writes a request to the I/O ring for transmission and issues a hypercall for sending an event to the netback. The VMI engine analyzes the request in the I/O ring and extracts a grant table reference, which is assigned to a memory page shared between VMs. Then the VMI engine finds a page frame number corresponding to the grant table reference. Finally, it maps the corresponding page and saves an Ethernet frame stored in it. At this time, it assigns a sequence number to the frame to prevent replay attacks.

For packets that a target VM receives, the VMI engine analyzes both requests and responses in the I/O ring for reception. To receive a packet from the netback driver, the netfront driver sends a request to the netback driver and registers a grant table reference for storing responses. The VMI engine analyzes the request and records the relationship between the reference and a request identifier. When the netback driver receives a packet, it writes a response with the request identifier to the I/O ring. At that time, the VMI engine finds a grant table reference from the identifier and then saves an Ethernet frame in the memory page corresponding to the reference.

For these purposes, the VMI engine securely identifies the I/O rings and the event channel used between the network drivers. When the netfront driver is initialized, it writes the grant table references for the I/O rings and the port number used for the event channel to the XenStore ring. The XenStore ring is a ring buffer used between a target VM and XenStore, which is a database for VM configuration in the management VM. If the VMI engine detects events sent to XenStore, it analyzes requests in the XenStore ring.

The RemoteTrans server periodically issues a hypercall and obtains packets saved by the VMI engine in the hypervisor. At this time, the VMI engine calculates a MAC from a list of Ethernet frames, a list of these sizes, the last

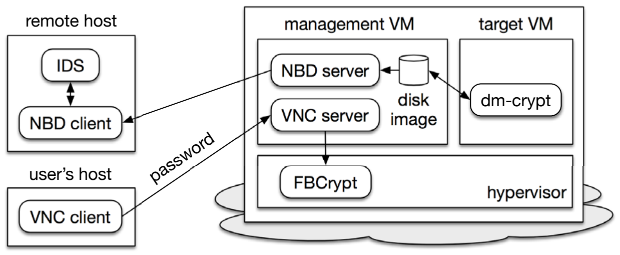


Figure 4. Storage introspection in RemoteTrans.

sequence number, and a secret key. Then the RemoteTrans server sends the MAC with the packet data to the runtime. The RemoteTrans runtime checks the MAC using received data and the secret key to guarantee the integrity. To detect replay attacks, it also checks if the last sequence number is consistent. Finally, it writes the received Ethernet frames to a created tap device.

C. *Remote Storage Introspection*

In RemoteTrans, the operating system in a target VM encrypts its storage using dm-crypt although the hypervisor can encrypt it [20]. This is natural as self-protection in semi-trusted clouds. At boot time, Linux Unified Key Setup (LUKS) requires a password for decrypting storage. A key file can be used, but it has to be located in an unencrypted partition of the storage. To prevent the key file from being stolen, users can input the password interactively using out-of-band remote management. This management method allows users to access booting VMs via VNC and SSH. However, since out-of-band remote management is enabled by the management VM, password inputs can be eavesdropped on by insiders in clouds [21].

To allow users to input passwords securely, RemoteTrans uses FBCrypt [21] and SCCrypt [22]. These systems encrypt all the inputs and outputs of out-of-band remote management between a client at a remote host and the hypervisor in a cloud, as shown in Fig. 4. Even if the management VM eavesdrops on inputs, it cannot obtain passwords. As a result, the management VM cannot illegally decrypt storage of target VMs.

RemoteTrans provides such protected storage to remote hosts using the network block device (NBD). NBD provides remote storage as a virtual block device. The management VM specifies the storage of a target VM and runs the NBD server, while the RemoteTrans runtime at a remote host mounts a created virtual block device. The runtime also decrypts mounted storage using dm-crypt by specifying the same password as in a target VM.

For the integrity of storage, the operating system in a target VM can calculate a hash of every disk block using dm-integrity. In the current implementation, RemoteTrans does not use dm-integrity because it is not included in standard Linux distributions usually used in target VMs.

V. Experiments

We conducted experiments to examine the security and performance of IDS remote offloading. For a target host, we used a PC with one Intel Xeon E3-1290 processor, 16 GB of memory, an HDD of 500 GB, and a Gigabit Ethernet NIC. We ran Xen 4.1.3 and assigned eight virtual CPUs and 12 GB of memory to the management VM. For a target VM, we assigned one virtual CPU, 4 GB of memory, and a virtual disk of 20 GB. In the VM, we ran Linux 2.6.27 in full virtualization. Only for experiments on network introspection, we ran Linux 3.5.0 with the paravirtual network driver. For a remote host, we used a PC with the same hardware specification as the target host. We ran various IDSes on Linux 3.2.0. In addition, we used a faster PC with one Intel Xeon E3-1270v3 processor, 16 GB of memory, an HDD of 2 TB, and a Gigabit Ethernet NIC. At this host, we used Linux 3.13.0.

We connected these hosts using a Gigabit Ethernet switch. Also, we considered network delay between them. We emulated a WAN by inserting network delay with the tc command at a remote host. Since the network delay to the nearest region of Amazon EC2 was reported as several milliseconds, we added network delay between 1 and 10 ms.

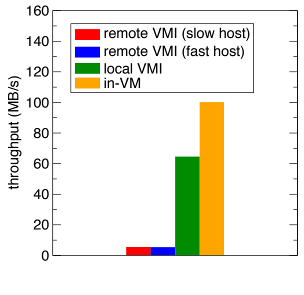
A. *Remote Offloading of Legacy IDSes*

To confirm that IDS remote offloading was done correctly, we ran several legacy IDSes in remote, local, and no offloading. IDSes were offloaded to a remote host in remote offloading, whereas they were offloaded to the management VM at the same host where a target VM ran in local offloading. In no offloading, IDSes ran inside a target VM. First, we executed chkrootkit, which detected rootkits installed in a target VM. In IDS offloading, chkrootkit used memory and storage introspection. The result of chkrootkit was exactly the same between remote and local offloading, while it was slightly different from the result in no offloading due to unsupported features. When we tampered with the ps command in the target VM, the offloaded chkrootkit could detect it correctly.

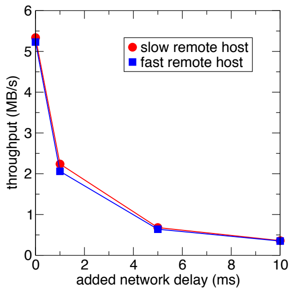
Next, we executed Tripwire, which checked the integrity of the filesystems in a target VM. In IDS offloading, Tripwire used storage introspection. When we used the same policy file, the offloaded Tripwire could scan almost the same number of files as in-VM Tripwire. The reason of the difference was that the files related to ld.so were specially handled in Transcall. Finally, we executed Snort, which captured the network packets from/to a target VM. In IDS offloading, Snort used network introspection. When we mounted portscans to the target VM using nmap, the offloaded Snort could detect the attack correctly.

B. *Insider Attacks in the Management VM*

We confirmed that RemoteTrans could detect insider attacks in the management VM. For remote memory in-

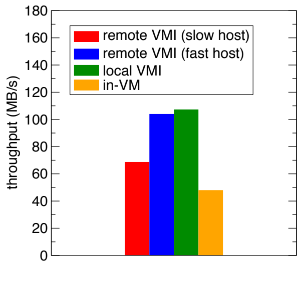


(a) Read throughput

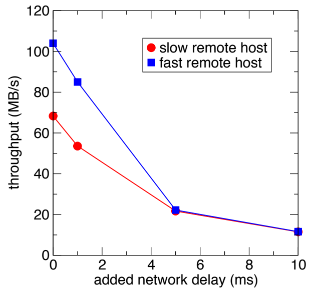


(b) Impact of network delay

Figure 5. The throughput of memory introspection.



(a) Read throughput



(b) Impact of network delay

Figure 6. The throughput of storage introspection.

trospection, we tampered with either requests or responses in a malicious RemoteTrans server. As a result, the RemoteTrans runtime could detect the modification of requests and responses because the MACs did not match. Next, we mounted replay attacks by returning a saved response for successive requests. Thanks to random nonces included in requests, the RemoteTrans runtime could detect the reuse of the old response. Finally, we eavesdropped on the kernel data included in responses, but we could not obtain any meaningful strings.

For remote network introspection, we tampered with packets that were forwarded by a malicious RemoteTrans server. Thanks to the MACs attached to the forwarded packets, the RemoteTrans runtime could detect tampering with the packets. Next, we mounted replay attacks by sending old packet data. Since the sent data contained the same sequence number as the last one, the RemoteTrans runtime could detect such attacks. For remote storage introspection, we eavesdropped on the disk image of a target VM in the management VM. We searched the disk image for password strings, but we could not find them because of full-disk encryption.

C. *Performance of Remote VMI*

To examine the performance of memory introspection, we have developed a benchmark suite for reading the memory of a target VM. The benchmark for remote memory introspection received memory contents via the RemoteTrans server and runtime. The benchmark for local memory introspection mapped memory pages and read the contents in the management VM. The in-VM benchmark read memory from `/dev/mem` in a target VM. As shown in Fig. 5a, the throughput of remote memory introspection was only 8%, compared with local one. This was due to the overhead of communication, MAC verification, and encryption. The difference of a remote host did almost not affect the throughput. When we added network delay, the throughput was inversely proportional as in Fig. 5b.

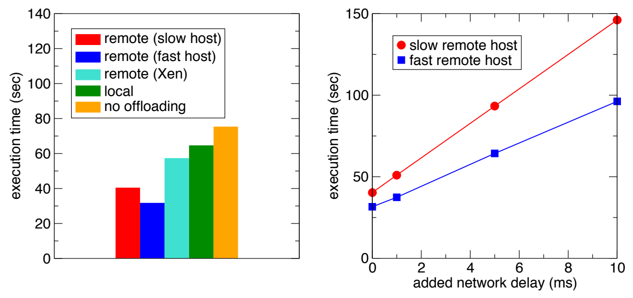
Next, we measured the performance of storage introspection using IOzone 3.430. We created a file of 1 GB in a target VM in advance and read it at a remote host, in the management VM, and in a target VM. Fig. 6a shows the throughput of disk reads. When we used a slow remote host, the performance of remote storage introspection was 64% lower than that of local one. However, using a faster remote host, the performance degradation was only 3.1%. This means that storage decryption is dominant in the throughput. When the delay was more than 5 ms, there was no difference between two remote hosts, as shown in Fig. 6b, because the network became a bottleneck. Surprisingly, the in-VM performance was the lowest. The reason is the virtualization overhead of storage.

Finally, we measured the packet loss rate under a heavy workload using D-ITG 2.8.1. We sent 2,000 packets of 1 KB to a target VM and captured them at a remote host, in the management VM, and in a target VM. No packets were lost in any types of network introspection.

D. *Performance of Legacy IDSes*

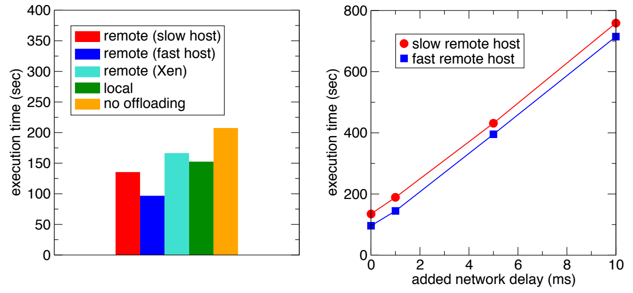
First, we measured the execution time of chkrootkit and Fig. 7a shows the results. Surprisingly, the execution time in remote offloading was the shortest. Even when we used a slow remote host, remote offloading was 60% and 87% faster than local and no offloading, respectively. For a faster remote host, remote offloading was 104% and 138% faster, respectively. This is mainly because there is no virtualization overhead at a remote host. It took more time to execute many system calls issued by chkrootkit and Transcall. To verify this, we ran Xen at a remote host and executed chkrootkit in its management VM. The execution time became longer, but remote offloading was still 13% faster than local offloading. The reason is under investigation. When we added network delay, the execution time increased as shown in Fig. 7b. Under no delay, the execution time was only 8.7 seconds longer at a slow remote host. Under 10 ms of delay, however, the difference became 50 seconds. This is because chkrootkit is CPU intensive. A fast remote host could execute chkrootkit more efficiently.

Next, we measured the execution time of Tripwire. As shown in Fig. 8a, the time in remote offloading was the shortest. Even when we used a slow remote host, remote offloading was 13% and 53% faster than local and no



(a) Execution time (b) Impact of network delay

Figure 7. The execution time of offloaded chkrootkit.



(a) Execution time (b) Impact of network delay

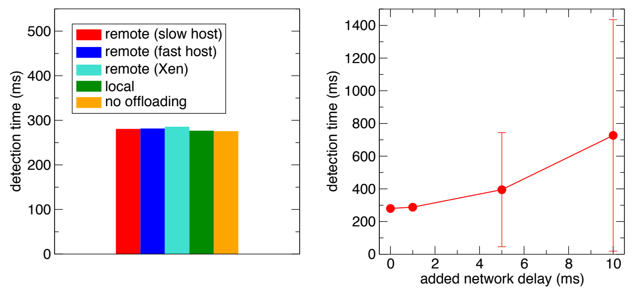
Figure 8. The execution time of offloaded Tripwire.

offloading, respectively. For a faster remote host, remote offloading was 58% and 115% faster, respectively. Like chkrootkit, the reason is the virtualization overhead in the management VM and a target VM. In fact, when we ran Xen at a remote host, remote offloading became 8.4% slower than local offloading. Fig. 8b shows the execution time when we added network delay. The time increased in proportion to the delay, but the difference between two remote hosts was constant. This means that remotely offloaded Tripwire is network intensive and the network performance determines the performance of Tripwire.

Finally, we measured the time in which Snort detected a portscan after we started it. As shown in Fig. 9a, the detection time in local and no offloading was almost the same, while that in remote offloading was 5 ms longer. This is because remote offloading has to forward packets captured by the VMI engine to a remote host and write them to a tap device. However, this delay increased the detection time only by 1.8%. Fig. 9b shows the average detection time and the standard deviation when we added network delay. The reason why the average increased is that it sometimes took a much long time to detect a portscan. For example, it took 2.4 seconds at worst when network delay was 10 ms. The detection time for most of the portscans did not change.

VI. Related Work

The idea of using remote hosts with IDSes is not new. Copilot [9] can remotely monitor the integrity of the kernel memory by using a PCI add-in card inserted in a target host.



(a) Detection time (b) Impact of network delay

Figure 9. The detection time of offloaded Snort.

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. The attackers at the target host cannot compromise the Copilot monitor or the remote host. Unlike IDS remote offloading, IDSes themselves do not run at a remote host.

HyperCheck [12] uses System Management Mode (SMM) to remotely monitor memory and CPU registers. In SMM, the CPU can securely execute code in System Management RAM (SMRAM), which cannot be accessed in the normal mode. In HyperCheck, a network device driver running in SMM makes a NIC read memory using DMA and send memory contents to a remote host. The remote host checks the integrity of the hypervisor and the operating system by running IDSes. However, while a CPU runs in SMM, all the regular tasks are suspended to maintain the integrity.

HyperSentry [19] allows a measurement agent inside the hypervisor to be securely executed using SMM even if the hypervisor have 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.

Also, secure execution of local IDSes has been proposed. HyperGuard [11] enables local IDSes to securely monitor the integrity of the hypervisor using SMM. It triggers an IDS in SMM by timer interrupts and the IDS checks the hypervisor memory. Like HyperCheck, all the regular tasks are suspended while the 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.

Flicker [10] 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.

A self-service cloud (SSC) computing platform [8] 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, the TCB is larger than RemoteTrans because a VM called DomB has to be also trusted.

VII. Conclusion

This paper proposed IDS remote offloading with remote VMI. This technique enables legacy IDSes to securely run at trusted remote hosts outside semi-trusted clouds. Remote VMI is initiated by remote hosts and introspects VMs using a minimal VMI engine in the trusted hypervisor inside clouds. It preserves the integrity and confidentiality of introspected data between the VMI engine and remote hosts. Thanks to the VMI engine, secure VM execution provided by the hypervisor can be securely bypassed. We have developed RemoteTrans for achieving IDS remote offloading in cooperation with Transcall. We confirmed that RemoteTrans could achieve surprisingly efficient execution of legacy IDSes in remote offloading.

One of our future work is to examine the performance of remotely offloaded IDSes when we run many VMs on a host. The VMI engine and the communication between the RemoteTrans server and runtime can become performance bottlenecks. Another direction is to enable offloaded IDSes to introspect target VMs more efficiently even under large network delay. For remote memory introspection, we could offload the analysis of kernel data to the VMI engine.

Acknowledgment

This work was supported in part by JSPS KAKENHI Grant Number JP16K00101.

References

[1] T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," in *Proc. Network and Distributed Systems Security Symp.*, 2003, pp. 191–206.

[2] X. Jiang, X. Wang, and D. Xu, "Stealthy Malware Detection through VMM-based "Out-of-the-box" Semantic View Reconstruction," in *Proc. Conf. Computer and Communications Security*, 2007, pp. 128–138.

[3] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection," in *Proc. Symp. Security and Privacy*, 2011, pp. 297–312.

[4] Y. Fu and Z. Lin, "Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection," in *Proc. Symp. Security and Privacy*, 2012, pp. 586–600.

[5] C. Li, A. Raghunathan, and N. K. Jha, "Secure Virtual Machine Execution under an Untrusted Management OS," in *Proc. Int. Conf. Cloud Computing*, 2010, pp. 172–179.

[6] F. Zhang, J. Chen, H. Chen, and B. Zang, "CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization," in *Proc. Symp. Operating Systems Principles*, 2011, pp. 203–216.

[7] H. Tadokoro, K. Kourai, and S. Chiba, "Preventing Information Leakage from Virtual Machines' Memory in IaaS Clouds," *IPSJ Online Trans.*, vol. 5, pp. 156–166, 2012.

[8] S. Butt, H. A. Lagar-Cavilla, A. Srivastava, and V. Ganapathy, "Self-service Cloud Computing," in *Proc. Conf. Computer and Communications Security*, 2012, pp. 253–264.

[9] N. Petroni, Jr., T. Fraser, J. Molina, and W. Arbaugh, "Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor," in *Proc. Conf. USENIX Security Symp.*, 2004.

[10] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki, "Flicker: An Execution Infrastructure for TCB Minimization," in *Proc. European Conf. Computer Systems*, 2008, pp. 315–328.

[11] J. Rutkowska and R. Wojtczuk, "Preventing and Detecting Xen Hypervisor Subversions," Black Hat USA, 2008.

[12] J. Wang, A. Stavrou, and A. Ghosh, "HyperCheck: A Hardware-assisted Integrity Monitor," in *Proc. Int. Symp. Recent Advances in Intrusion Detection*, 2010, pp. 158–177.

[13] T. Iida and K. Kourai, "Transcall," http://www.ksl.ci.kyutech.ac.jp/oss/transcall/.

[14] PwC, "US Cybercrime: Rising Risks, Reduced Readiness," 2014.

[15] TechSpot News, "Google Fired Employees for Breaching User Privacy," http://www.techspot.com/news/40280-google-fired-employees-for-breaching-user-privacy.html, 2010.

[16] CyberArk Software, "Global IT Security Service," 2009.

[17] Y. Oyama, T. Giang, Y. Chubachi, T. Shinagawa, and K. Kato, "Detecting Malware Signatures in a Thin Hypervisor," in *Proc. Symp. Applied Computing*, 2012, pp. 1807–1814.

[18] N. Santos, K. P. Gummadi, and R. Rodrigues, "Towards Trusted Cloud Computing," in *Proc. Workshop Hot Topics in Cloud Computing*, 2009.

[19] A. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. Skalsky, "HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity," in *Proc. Conf. Computer and Communications Security*, 2010, pp. 38–49.

[20] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato, "BitVisor: A Thin Hypervisor for Enforcing I/O Device Security," in *Proc. Int. Conf. Virtual Execution Environments*, 2009, pp. 121–130.

[21] T. Egawa, N. Nishimura, and K. Kourai, "Dependable and Secure Remote Management in IaaS Clouds," in *Proc. Int. Conf. Cloud Computing Technology and Science*, 2012, pp. 411–418.

[22] K. Kourai and T. Kajiwara, "Secure Out-of-band Remote Management Using Encrypted Virtual Serial Consoles in IaaS Clouds," in *Proc. Int. Conf. Trust, Security and Privacy in Computing and Communications*, 2015, pp. 443–450.