Efficient VM Introspection in KVM and Performance Comparison with Xen

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

Kousuke Nakamura
Department of Creative Informatics
Kyushu Institute of Technology
Fukuoka, Japan
work02@ksl.ci.kyutech.ac.jp

***Abstract*—Intrusion detection system (IDS) offloading is useful for securely executing IDSes. It runs a target system in a virtual machine (VM) and enables IDSes to monitor the VM from the outside using *VM introspection*. Although VM introspection is well studied, its performance has not been reported in detail. The performance becomes important when users choose virtualization software, e.g., Xen and KVM. However, the performance comparison is difficult because there is no efficient implementation of VM introspection in KVM. In this paper, we first propose KVMonitor for efficient VM introspection in KVM. Using KVMonitor, we have ported Transcall for offloading legacy IDSes. For memory introspection, KVMonitor was 32 times faster than the existing LibVMI. Then we present performance comparison between Xen and KVM on VM introspection. The experimental results showed that checking the kernel memory with KVMonitor was 118 times faster than that in Xen. Even for legacy chkrootkit, the execution time with KVMonitor was 63% shorter than that in Xen.**

***Keywords*-Virtual machine introspection, virtualization software, intrusion detection systems, IDS offloading, security**

I. Introduction

Attacks against networked servers are increasing. As one of the methods for detecting such attacks, intrusion detection systems (IDSes) are used. IDSes monitor the system and network of servers and alerts to administrators if they detect symptoms of attacks. Recently, however, attackers attempt to disable or tamper with IDSes after they intrude into servers. If IDSes are compromised, they cannot detect attacks. To counteract such attacks against IDSes, a technique of offloading IDSes using virtual machines (VMs) has been proposed [1]. *IDS offloading* runs a target system in a VM and executes IDSes in the outside of the VM. This technique can prevent IDSes from being compromised even if attackers intrude into the VM and increase the security of IDSes.

The enabling technology of IDS offloading is *VM introspection* [1]. VM introspection is a technique for monitoring the memory, disks, and networks of VMs from the outside. It has been studied for various kinds of virtualization software. In Livewire [1], VM introspection was first implemented using VMware Workstation [2]. Later, many researchers use Xen [3] to implement VM introspection [4]–[13]. VMware provides the VMsafe API for VM introspection and several products such as Trend Micro Deep Security [14] have been released for VMware vSphere [15]. For other virtualization software, there is relatively less work on VM introspection for each.

Although VM introspection is well studied, the performance of VM introspection has not been reported in detail. However, it becomes important when users choose one of various kinds of virtualization software. To the best of our knowledge, there is no performance comparison among different virtualization software in terms of VM introspection. Most of the researches target only one virtualization software although some claim that the proposed systems are applicable to various virtualization software. Exceptionally, VMwatcher [4] was implemented in Xen, QEMU [16], VMware Server, and User-mode Linux (UML) [17], but the performance was reported only for UML. EXTERIOR [18] was implemented in KVM [19] and QEMU, but the performance was not different because the experiments were done only for the memory snapshot of a VM. VM introspection for memory snapshots is not affected by the difference of virtualization software.

For performance comparison, Xen and KVM are indispensable because they are widely used open source virtualization software. For Xen, the implementation of VM introspection is well explained in many literatures and there are several efficient open source implementations [20], [21]. For KVM, however, the implementation details of VM introspection are unclear in literatures although studies on VM introspection have been actively done recently [18], [21]–[25]. Since KVM uses custom QEMU, studies on VM introspection for QEMU [4], [26], [27] may be useful, but the implementation details are also unclear. Among them, LibVMI [21] is promising because it is an open source implementation of VM introspection for Xen and KVM. Unfortunately, the performance of memory introspection in KVM is very low due to implementation issues. To compare the performance between Xen and KVM, we cannot use LibVMI because it is not fair for KVM.

In this paper, we first propose KVMonitor for efficient VM introspection in KVM. KVMonitor enables offloading IDSes in a VM onto the host operating system outside the VM. It provides the functions for introspecting the memory, disks, and networks of VMs. For efficient memory introspection, KVMonitor and offloaded IDSes share a file created for the physical memory of a VM with negligible overhead. For disk introspection, KVMonitor supports qcow2, which is a disk format unique to KVM (and QEMU). For network introspection, KVMonitor captures packets from the tap network device for a VM. To enable legacy IDSes to be offloaded with KVMonitor, we have ported Transcall [28], which we have developed for Xen. We have implemented KVMonitor for KVM 1.1.2 and confirmed that KVMonitor was 32 times faster than LibVMI on memory introspection. In addition, several legacy IDSes offloaded with KVMonitor could detect intrusions by single runs and cross-view diff [29].

Next, we present performance comparison between Xen and KVM on VM introspection. We examined the performance of memory, disk, and network introspection as microbenchmarks and the execution times of offloaded IDSes as macrobenchmarks. Our results show that checking the kernel memory with KVMonitor was 118 times faster than that in Xen. The performance of the offloaded Tripwire [30] was almost the same when we used the same disk format. Even when we offloaded chkrootkit [31] using Transcall, the execution time with KVMonitor was 63% shorter than that in Xen. Our conclusion from various experiments is that VM introspection in KVM is more efficient than or the same as that in Xen.

The rest of this paper is organized as follows. Section II describes IDS offloading in Xen. Section III proposes KVMonitor for enabling efficient IDS offloading in KVM. Section IV describes the effectiveness of IDS offloading using KVMonitor and Section V reports the performance comparison between IDS offloading in KVM and Xen. Section VI describes related work and Section VII concludes the paper.

II. Background

A. VM Introspection in Xen

Xen is a type I virtual machine monitor (VMM), which runs directly on hardware, as shown in Fig. 1. On top of the VMM, Xen runs two types of VMs: DomU and Dom0. DomU is a regular VM, while Dom0 is a VM that has privileges for managing DomU. For DomU, Xen provides two virtualization modes: para-virtualization and full virtualization. Para-virtualization requires modification to the guest operating system for efficiency, while full virtualization does not. In either mode, IDSes are usually offloaded from DomU to Dom0 because Dom0 can access resources in DomU. In general, IDSes can be offloaded to the VMM. Recently, VM introspection from other VMs has been also proposed. For example, SSC [12] allows IDSes to be offloaded to special-purpose VMs called service domains. In VMCoupler [32], IDSes are offloaded to a guard VM, which can be co-migrated with its target VM. In this paper, we focus on offloading IDSes to Dom0 because the performance

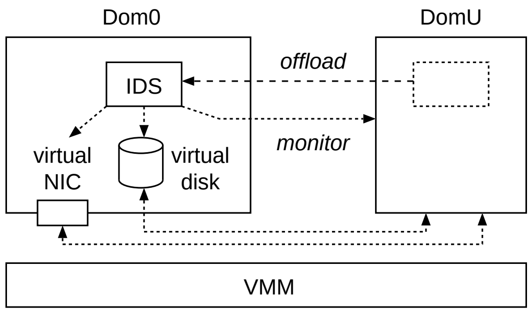

Figure 1. VM introspection by an offloaded IDS in Xen.

differences between Dom0 and a guard VM have been already reported in [32].

To introspect the memory of DomU, an IDS offloaded to Dom0 maps the memory pages of DomU onto its address space. It issues a hypercall provided by the VMM and adds entries for those memory pages to its page tables. After it finishes to introspect those pages, it unmaps them. To examine specific kernel data, an offloaded IDS first translates the virtual address of the kernel data and finds a physical page in which the kernel data is located. For this address translation, the IDS issues a hypercall to the VMM to obtain the value of the CR3 register of a virtual CPU in DomU. The CR3 register points to the page directory of the current process. Then, it starts with the physical address stored in the CR3 register and traverses the page directory in DomU. During the traversal, it repeatedly maps several memory pages in which page directory entries or page table entries are stored. Finally, it maps the target page and accesses the kernel data.

To introspect the virtual disk of DomU, an offloaded IDS accesses the disk image for DomU. In Xen, a disk image for a virtual disk is created as a file or a disk partition in Dom0. By default, the raw format is used for a file-backed disk image. The disk image with the raw format can be directly mounted in a loopback mode if the kernel in Dom0 supports the file system used in a virtual disk. If the logical volume manager (LVM) is used in a virtual disk, the disk image can be mounted after the logical volume is activated. When the disk image is used by running or suspended DomU, it has to be mounted in a read-only manner to prevent metadata in the file system from being corrupted. Another method is introspecting disk accesses caused by DomU [5]. It intercepts all the low-level disk traffic and infers high-level file system operations. In this paper, we focus on the monitoring of disk images.

To introspect the network of DomU, an offloaded IDS captures the packets from virtual network interface cards (NICs) for DomU. In Xen, corresponding to a network interface (e.g., eth0) of DomU, a virtual NIC is created in Dom0. A virtual network interface (e.g., vif1.0) is created

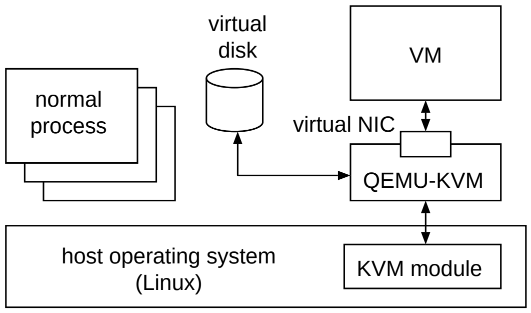

Figure 2. The architecture of KVM.

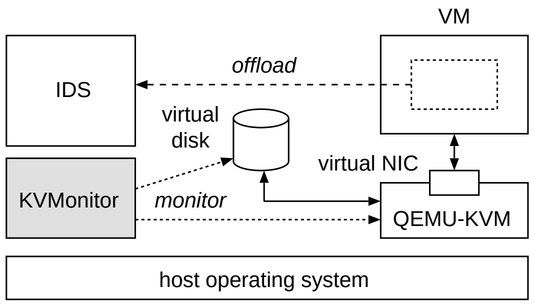

Figure 3. The architecture of KVMonitor.

for para-virtualization, whereas a tap network device (e.g., tap1.0) is created for full virtualization. A virtual NIC is connected to the network bridge in Dom0. When Dom0 receives a packet for DomU from an external host or another VM in the same host, it sends the packet to DomU via the network bridge and the virtual NIC. When DomU sends a packet to the external, Dom0 receives it via the virtual NIC and sends it via the network bridge. As such, virtual NICs can receive all the packets from/to DomU.

B. Architecture of KVM

KVM consists of a kernel module and user-level QEMU-KVM processes, as illustrated in Fig 2. The KVM module assists the virtualization of CPU and memory using hardware support such as Intel VT. QEMU-KVM is QEMU customized for KVM and emulates virtual devices such as disks and networks of a VM. Therefore, KVM is called a hybrid VMM of type I and II. The KVM module is type I VMM running on hardware, whereas QEMU-KVM is type II VMM running on top of the host operating system. In KVM, a QEMU-KVM process is created per VM like other normal processes and a VM runs as a part of the process. Like Xen, a virtual disk is created as a file in the host operating system, but its format is qcow2 unique to QEMU by default.

III. KVMonitor

In this paper, we present KVMonitor for efficient VM introspection in KVM. KVMonitor is a library liked to IDSes and allows offloaded IDSes to introspect the memory, disks and networks of VMs. As illustrated in Fig. 3, an IDS is offloaded onto the host operating system. An offloaded IDS introspects a VM by accessing virtual devices managed by QEMU-KVM via KVMonitor. Another possible architecture is offloading IDSes to a special VM for VM introspection like Xen. For that, however, we have to give the VM privileges for accessing other VMs and it would require large modification to KVM. In addition, IDSes offloaded to such a VM would suffer from virtualization overhead. Therefore, KVMonitor assumes an architecture for offloading IDSes directly to the host operating system.

A. Memory Introspection

To enable offloaded IDSes to introspect the memory of a VM, the KVMonitor-enabled QEMU-KVM creates physical memory allocated to the VM as a file called a *memory file*. Then both QEMU-KVM and IDSes map the file in their address spaces by using the mmap system call, as shown in Fig. 4. Using the memory-mapped file, QEMU-KVM can directly access the physical memory of the VM as usual, while IDSes can introspect the memory. In the original QEMU-KVM, the physical memory of a VM is allocated using malloc by default. Therefore, it cannot be accessed from the outside of the QEMU-KVM process. The original QEMU-KVM also provides an option for using a newly created file as the physical memory of a VM, but it removes the file after it maps the file so that the file is not accessed by the other processes. We modified QEMU-KVM so that it does not remove the file. This may lower VM security but does not affect VM performance.

QEMU-KVM locates the memory file in a directory where the HugeTLB filesystem is mounted. The HugeTLB filesystem uses the huge page mechanism, which extends a page size from 4 KB to 2 MB or 1 GB when a file on it is mapped. We used a page size of 2 MB for memory-mapped memory file. That large page size suppresses the consumption of TLB entries for VM's memory and makes address translations faster. Then the impact on the memory performance in the host operating system is minimized.

To enable IDSes to access the kernel data in the memory

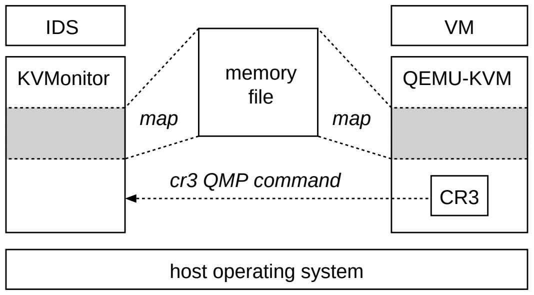

Figure 4. Memory introspection using a memory file.

```
{ "execute": "cr3" }
{ "return": { "CR3": "0x000000001f96e000" } }

{ "execute": "xaddr",
  "arguments": { "addr": "0xffffffff814a8340" } }
{ "return": { "paddr": "0x00000000014a8340" } }
```

Figure 5. The execution of the cr3 and xaddr commands using QMP.

of a VM using virtual addresses, KVMonitor provides an API for translating a virtual address to a physical one. Since the memory file mapped is the physical memory of a VM, IDSes need to access it using physical addresses.

We have implemented two mechanisms for the address translation. One is that KVMonitor translates virtual addresses by itself like Xen. First, KVMonitor obtains the value of the CR3 register in a virtual CPU of a VM by communicating with QEMU-KVM. For this purpose, we added a new cr3 command for obtaining the value to QEMU-KVM because QEMU-KVM provides only the command for dumping the values of all registers in text. It is time-consuming to obtain the dump of all registers and analyze it.

KVMonitor executes the cr3 command using QEMU monitor protocol (QMP). QMP is based on Javascript object notation (JSON). When KVMonitor connects to QEMU-KVM, QEMU-KVM returns version information. To enable QMP, KVMonitor sends the qmp_capabilities command. Then it sends the cr3 command and receives the result, as shown in the upper half of Fig. 5. QEMU-KVM finds the CPU state stored in CPUArchState and obtains the value of the CR3 register.

Using the obtained value, KVMonitor looks up a local address in the memory-mapped file from a virtual address. First, it translates a virtual address to a physical one by traversing the page table. Since our target is x86_64, the four-level page table is used in a VM. It calculates the local address of the page global directory by adding the local address to which the memory file is mapped and the physical address of the page global directory, which is stored in the CR3 register. From the page directory entry, KVMonitor looks up the page upper directory, which is the next level in the page directory. Similarly, it looks up the page middle directory, the page table, and a target page frame number. Then it calculates the physical address from the page frame number and the page offset. Finally, it calculates the local address corresponding the physical address.

The other mechanism is that QEMU-KVM performs address translation. This is easy to implement because QEMU-KVM has a debug function cpu_get_phys_page_debug for that. To use this function, we added the xaddr QMP command to QEMU-KVM. When KVMonitor executes the xaddr command with a parameter of a virtual address using QMP, QEMU-KVM invokes the function and translates the address into a physical one. Then it returns the translated address to KVMonitor, as shown in the bottom half of Fig. 5.

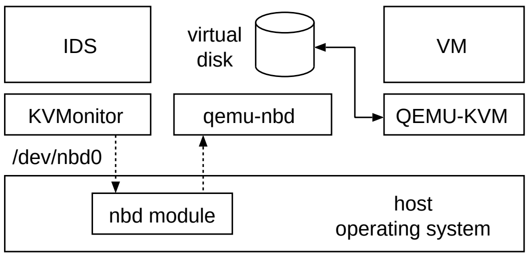

Figure 6. Disk introspection using NBD.

While IDSes analyze the memory of a VM with either mechanism, KVMonitor suspends the VM for consistency. Let us consider that IDSes access the process list by traversing pointers. If the operating system in the VM concurrently modifies some of the pointers to add or remove a process, IDSes may crash due to an invalid pointer. To suspend and resume a VM, we added the stop and cont QMP commands to QEMU-KVM.

There are trade-offs between these two mechanisms. One advantage of the former KVMonitor-centric mechanism is efficiency. KVMonitor communicates with QEMU-KVM only at once because it can translate a series of addresses using the same value of the CR3 register. In the latter QEMU-centric mechanism, KVMonitor has to communicate with QEMU-KVM whenever it translates an address. Another advantage is that the KVMonitor-centric approach can naturally translate virtual addresses of a specific process by traversing its page directory. On the other hand, the advantages of the QEMU-centric mechanism are ease of implementation and portability. The code for address translation is already in QEMU-KVM. In the KVMonitor-centric mechanism, we have to implement address translation for each processor architecture. KVM originally supported only x86 but is ported to various processors such as S/390, PowerPC, IA-64, and ARM.

B. Disk Introspection

To introspect a virtual disk with the qcow2 format, KVMonitor uses the network block device (NBD) for QEMU. The qcow2 format is the default in KVM. It has an advantage of saving a disk space because it allocates a real disk space only to used disk blocks, not to the whole blocks. However, a disk image of the qcow2 format cannot be directly mounted in the host operating system. Using NBD, KVMonitor mounts the disk image as a virtual block device and provides IDSes with an execution environment for introspecting a virtual disk.

NBD for QEMU consists of the nbd kernel module and user-level qemu-nbd processes, as illustrated in Fig. 6. KVMonitor first loads the nbd module into the kernel and creates a block device like `/dev/nbd0`. Next, it binds the device to the disk image for a VM by qemu-nbd, which is an NBD server supporting the qcow2 format. When the logical volume manager (LVM) is used, KVMonitor creates a device map for each disk partition by the kpartx command and activates logical volumes by the vgchange command. Finally, KVMonitor mounts the block device to a specified directory. Whenever IDSes access the directory, a block-level request is transferred to qemu-nbd. Then, qemu-nbd converts the requested block into a block of the raw format and returns it. To maintain the integrity of the filesystem in a virtual disk, KVMonitor mounts a disk image for a VM in a read-only manner.

In addition, KVMonitor can introspect virtual disks with the raw format because KVM supports that format as well. Like Xen, KVMonitor associates an available loopback device (e.g., `/dev/loop0`) to a disk image by the losetup command. Then it creates device maps and mounts them in a similar way to a disk image with qcow2. For the raw format, KVMonitor does not use NBD.

C. Network Introspection

To introspect the network of a VM, an offloaded IDS can capture packets from virtual NICs of the VM. For virtual NICs, QEMU-KVM creates tap network devices (e.g., vnet0) like Xen's full virtualization. The tap devices are connected to the network bridge and all the packets from/to a VM can be captured. As another method, we could use the functionality of QEMU-KVM for logging captured packets in a file. However, this method cannot capture packets when KVM uses an optimization called vhost-net, which bypasses QEMU-KVM in network processing.

D. Transcall with KVMonitor

Transcall [28] provides an execution environment for IDSes to transparently introspect a VM. Using Transcall, legacy IDSes can be offloaded 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, obtains necessary information on the guest kernel from the memory of a VM, and returns it to IDSes. The shadow filesystem provides the same filesystem view in a VM. To achieve this, the shadow filesystem also provides 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. For example, it traverses the process list in the guest kernel, collects process information, and creates pseudo files named `stat` and `status` for each process.

We have ported Transcall developed for Xen to KVM using KVMonitor. We modified the system call emulator and the shadow proc filesystem so that they access the memory of a VM using KVMonitor. In addition, we modified the shadow file system so that it accesses the directory to which the disk image for a VM is mounted by KVMonitor.

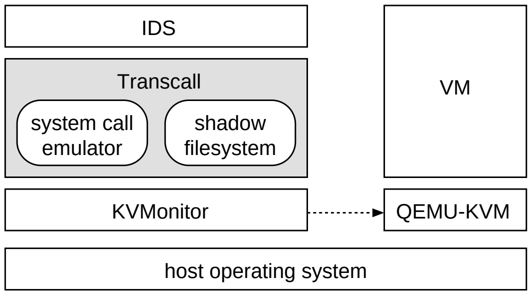

Figure 7. Offloading legacy IDSes using Transcall with KVMonitor.

IV. Effectiveness of KVMonitor

We conducted several experiments to confirm that KVMonitor achieved efficient VM introspection in terms of memory introspection and enabled effective IDS offloading. We used a PC with an Intel Xeon E5630 processor, which was equipped with 12 MB of L3 cache, 6 GB of DDR3 PC3-8500 memory, 250 GB of SATA HDD, and gigabit Ethernet, respectively. We ran modified KVM 1.1.2 as the hypervisor and Linux 3.2.0 as the host operating system. We created a VM with one virtual CPU, 512 MB of memory, a 20 GB disk, and gigabit Ethernet. We ran fully virtualized Linux 2.6.27.35 for the guest operating system and used the ext3 filesystem.

A. Efficiency of Memory Introspection

First, we compared the performance of memory introspection in KVMonitor with that in LibVMI 0.8 [21]. LibVMI is an open source library for VM introspection and supports KVM. In this experiment, we applied the patch provided by LibVMI to QEMU-KVM. The patch is for accessing the physical memory of VMs faster by adding the pmemaccess QMP command because the original QEMU-KVM provides only a command for dumping memory in text. This command returns memory contents in binary for memory reads.

To measure the performance of memory introspection, we have developed an *out-of-VM* memory benchmark for reading memory from the outside of a VM. This benchmark accesses each physical memory page of a VM by 4 KB and copies the contents to a local buffer by memcpy as fast as possible. At this time, the translation from virtual to physical addresses of a VM is not performed. The benchmark results are shown in Fig. 8. KVMonitor was 32 times faster than LibVMI. The cause was that LibVMI had to issue QMP commands to QEMU-KVM for each memory access to the VM. KVMonitor could map the memory file for the VM at first and access the mapped memory directly.

Second, we examined how the performance of memory accesses *inside* a VM was affected in KVM by using a

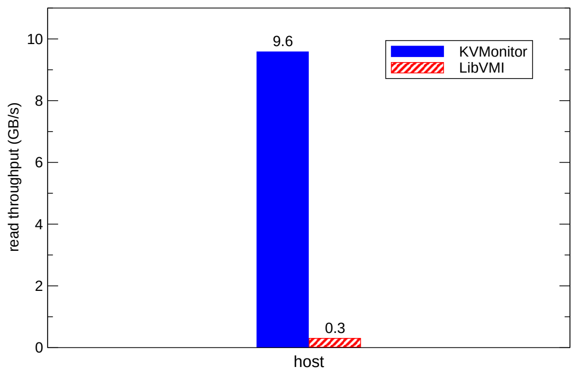

Figure 8. The throughput of memory reads in KVMonitor and LibVMI.

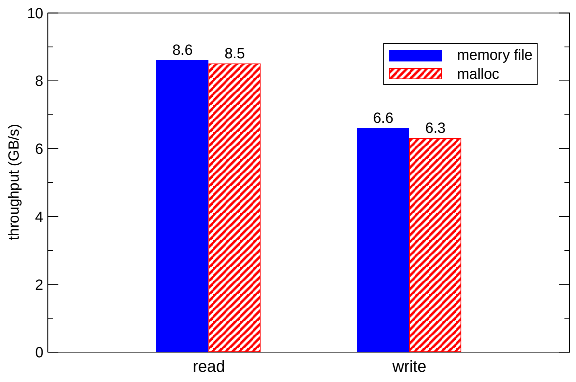

Figure 9. The throughput of memory reads and writes inside a VM.

memory file as the memory of the VM. The original QEMU-KVM allocates the memory of a VM by malloc, while the KVMonitor-enabled QEMU-KVM maps a memory file and uses it as the memory of a VM. Using a memory-mapped file may suffer from extra overhead. For this experiment, we have developed an *in-VM* memory benchmark for accessing memory inside a VM. This benchmark reads or writes 128 MB of the memory allocated by malloc. To avoid the speedup due to processor cache, this benchmark accesses larger amount of memory than the L3 cache of a processor. Fig. 9 shows the performance of memory reads and writes inside a VM. Using a memory file did not degrade the performance for both reads and writes. On the contrary, it was slightly improved probably because the number of TLB misses decreased thanks to the HugeTLB filesystem.

Third, we compared the performance of address translation between KVMonitor using the cr3 command and the xaddr command. Using the cr3 command, KVMonitor obtains the value of the CR3 register from QEMU-KVM and performs address translation by itself. For the xaddr command, QEMU-KVM translates a virtual address and returns a physical one to KVMonitor. For this experiment, we have developed an out-of-VM benchmark for traversing the process list in a VM and obtaining all process names and IDs. Fig. 10 shows the execution time of this benchmark.

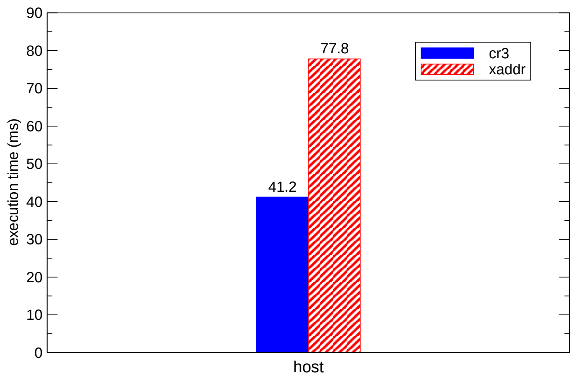

Figure 10. The time for traversing the process list in a VM.

KVMonitor using the cr3 command was 1.9 times faster than that using the xaddr command. This is due to the difference of the number of communication with QEMU-KVM. The xaddr command is executed for each address translation, whereas the cr3 command is executed only at once because the value does not change during this benchmark. In the following experiments, we used KVMonitor using the cr3 command.

B. IDS Offloading

We offloaded three types of IDSes to the host operating system using KVMonitor. First, we executed the offloaded Tripwire 2.4.1.2 [30] and monitored the virtual disk of the VM. Tripwire is a host-based IDS for checking the integrity of disks. In advance, we recorded the normal state of the file system in the VM by running Tripwire. After that, we added, deleted, and modified three files, respectively. Then, we checked the integrity of the disk by running Tripwire again. As a result, the offloaded Tripwire could detect the changes to the three files correctly, as shown in Fig. 11.

Next, we executed the offloaded Snort 2.9.2 [33] and monitored the network of the VM. Snort is a network-based IDS for checking network packets. We configured Snort so that it monitored the tap device for the VM. Then we ran nmap 5.21 [34] and performed portscans against the VM. We confirmed that alert logs were recorded in the host operating system correctly, as shown in Fig. 12.

Finally, we executed the offloaded chkrootkit 0.49 [31] using Transcall. Chkrootkit is a host-based IDS for detecting installed rootkits. It obtains information on processes and networks by using the ps and netstat commands and inspects several files. For the system with the normal state,

```
Rule Name              ... Added Removed Modified
Monitor Filesystems      1      1       1
Total Objects scanned: 67082
Total violations found: 3
```

Figure 11. Inconsistency detections by the offloaded Tripwire.

```
[**] [1:1421:11] SNMP AgentX/tcp request [**]
[Classification: Attempted Information Leak] ...
01/28-10:47:13.406931
  192.168.0.68:47962 -> 192.168.0.81:705
```

Figure 12. An alert log of the offloaded Snort.

```
$ transcall chkrootkit ps netstat
ROOTDOR is '/'
Checking 'ps'...INFECTED
Checking 'netstat'...INFECTED
```

Figure 13. Tamper detections by the offloaded chkrootkit.

the offloaded chkrootkit was executed correctly and was reported no rootkits. When we tampered with ps and netstat in the VM, the offloaded chkrootkit was reported the tamper correctly, as shown in Fig. 13.

C. Cross-view Diff

To detect hidden processes, we compared the result of the offloaded ps command with that of the ps command in the VM. In this experiment, we tampered with the ps command in the VM so that it did not show the init process. Since the offloaded ps command showed the init process correctly as in Fig. 14, we could find the hidden init process by comparing those results.

Similarly, to detect hidden network ports, we compared the results between the offloaded netstat command and the netstat command in the VM. In this experiment, we tampered with the netstat command so that it did not show port 5900. By comparison between those results, we could find the hidden port 5900 because only the offloaded netstat command showed port 5900 as in Fig. 15.

V. Performance Comparison

We compared the performance of VM introspection and the performance of offloaded IDSes in KVM with those in Xen. For Xen, we used the same PC as the one running KVM. In the PC, we ran Xen 4.1.3 as the hypervisor and fully virtualized Linux 3.2.0 as the operating system in

```
$ transcall ps -A
PID TTY          TIME CMD
  1 ?        00:00:00 init
  2 ?        00:00:00 kthreadd
```

Figure 14. The execution result of the offloaded ps command.

```
$ transcall netstat -ant
Proto ... Local Address   Foreign Address   State
tcp       0.0.0.0:22      0.0.0.0:*         LISTEN
tcp       0.0.0.0:5900    0.0.0.0:*         LISTEN
```

Figure 15. The execution result of the offloaded netstat command.

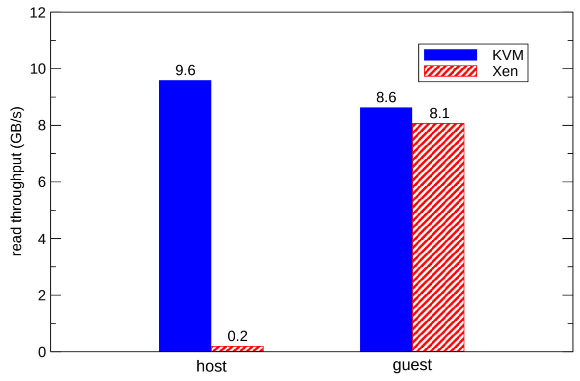

Figure 16. The throughput of memory reads.

Dom0. In this paper, we focused only on full virtualization that was also supported by KVM. We created a VM with the same number of virtual CPU, the same amount of memory, a disk with the same size, and gigabit Ethernet as the VM in KVM. We ran the same guest operating system and used the same filesystem. We did not use PV drivers in both VMs.

To compare the performance of VM introspection between KVM and Xen, we conducted experiments in the following four configurations:

- KVM_host for introspecting the VM from the host operating system in KVM
- Xen_host for introspecting DomU from Dom0 in Xen
- KVM_guest for monitoring resources inside the VM
- Xen_guest for monitoring resources inside DomU

A. Memory Introspection

First, we compared the performance of memory introspection in KVM with that in Xen. For Xen, we have developed a tool similar to LibVMI because LibVMI for Xen is efficient. For KVM_host and Xen_host, we used the out-of-VM memory benchmark in Section IV-A. We have developed a Xen version of the benchmark, which maps each memory page of a VM by 4 KB, copies the contents to a local buffer, and then unmaps the page. For KVM_guest and Xen_guest, we used the in-VM memory benchmark in Section IV-A.

Fig. 16 shows the performance of memory reads. The performance in KVMonitor was 48 times faster than that in Xen. The reason is that Xen has to repeat mapping and unmapping for each memory page of a VM. KVMonitor could map the whole memory at first and access it like heap memory. Xen also provides the API for mapping multiple memory pages of a VM at batch, but the time for mapping multiple pages was proportional to the number of pages. Therefore, the performance of memory introspection was similar even when we performed batch mapping. In addition, we could not map the whole memory of a VM by using the API because of a too large number of memory pages. Even if we could map the whole memory of a VM, we have to

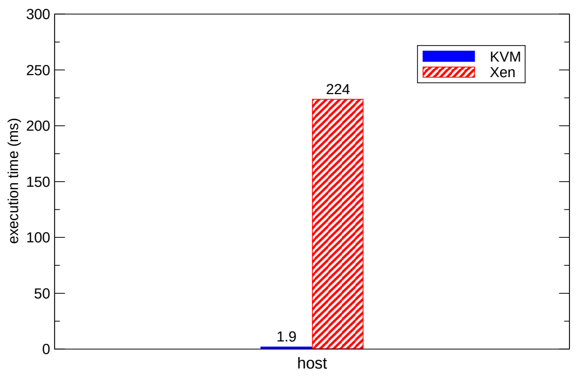

Figure 17. The execution time of the IDS for kernel checking.

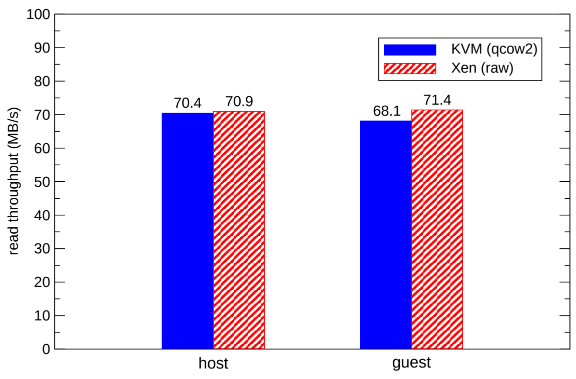

Figure 18. The throughput of file reads.

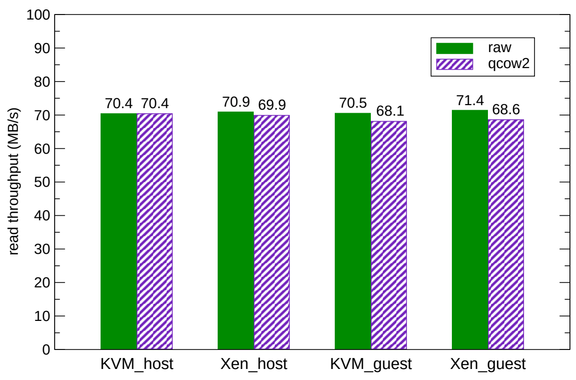

Figure 19. The throughput of file reads in different formats of disk images.

re-map it again whenever the allocation of physical memory to a VM is changed. In KVM, the allocation of physical memory to a mapped memory file may be changed, but the host operating system hides it. As a result, we can continue to use the mapped memory file.

Moreover, KVM_host was 11% faster than KVM_guest although KVM_guest did not need to map memory. This is because KVM_guest suffered from the overhead of CPU and memory virtualization for executing the benchmark. The performance in a VM was 7% better in KVM that that in Xen.

Next, we examined the performance of checking the guest kernel in a VM. We have developed an IDS for reading the code area of the guest kernel. This IDS translates virtual addresses to physical ones by 4 KB and reads the contents from the start of the kernel to the end. For the address translation, this IDS accesses the page tables in a VM. As shown in Fig. 17, KVM_host was 118 times faster than Xen_host in the same reason of the above experiment. In Xen, the IDS had to map many pages for traversing the page directory as well as accessing the code area of the kernel.

B. Disk Introspection

To measure the performance of file reads, we used the IOzone 3.414 [35]. In this experiment, IOzone created 1 GB of a file in the virtual disk of the VM in advance and read it sequentially. We mounted the disk image of the VM and accessed the file in KVM_host and Xen_host, while we directly accessed it in KVM_guest and Xen_guest. To exclude the influences of the filesystem cache in the operating systems, we erased the cache using the proc filesystem for each run of IOzone. We ran IOzone 10 times for each configuration.

Fig. 18 shows the average read throughput. In this experiment, we used the qcow2 format in KVM and the raw format in Xen, which are default formats. The throughputs of Xen_host and KVM_host were almost the same although these used different disk formats. For in-VM benchmarking, KVM_guest was 4.6% worse than Xen_guest. This is due to the qcow2 format. For the comparison between out-of-VM and in-VM benchmarking, KVM_host was 3.4% better than KVM_guest. Xen_host was almost the same as Xen_guest.

Next, we measured the read throughput for two different formats of disk images. In KVM and Xen, we used both the raw and qcow2 formats. Xen also supports the qcow2 format by using the blocktap mechanism. To introspect a virtual disk with the qcow2 format in Xen, we used NBD for QEMU as KVMonitor did. Fig. 19 shows the results for all the combination of disk format and virtualization software. For the out-of-VM benchmarking, the difference between the raw and qcow2 formats was small. In addition, KVM_host and Xen_host were almost the same. For the in-VM benchmarking, the qcow2 format degraded read throughput by 3 or 4%, compared with the raw format.

Finally, we measured the time needed for executing the offloaded Tripwire. As in the above experiment, we used the four combination of disk format and virtualization software. To make Tripwire check the equivalent filesystem in both formats, we converted the disk image in the raw format into that in the qcow2 format. We configured Tripwire so that it checked the whole disk.

As shown in Fig. 20, the execution times were almost the same in KVM and Xen for the same disk format. However,

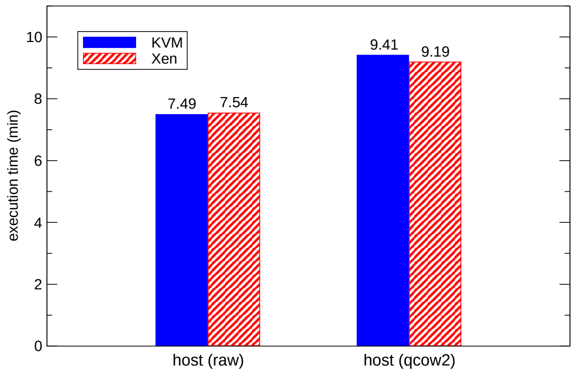

Figure 20. The execution time of the offloaded Tripwire.

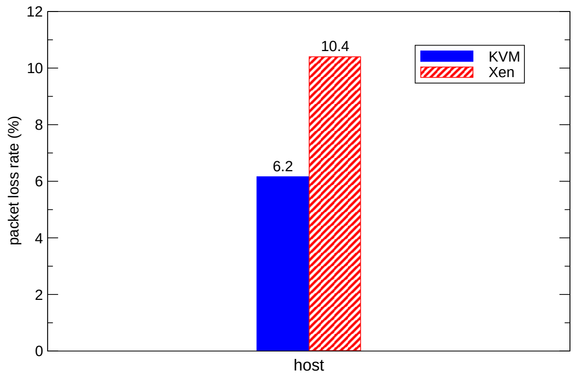

Figure 21. The packet loss rates of the offloaded Snort.

the qcow2 format degraded the performance of the offloaded Tripwire by 18 or 20%, compared with the raw format. As a result, the performance of disk introspection in KVM is 20% lower than that in Xen when the default format is used. This is due to the overhead of NBD used for converting the qcow2 format into the raw format at runtime. Unlike the above experiment using IOzone, Tripwire accesses too many files and directories. When sequential reads are done for a file, the readahead mechanism of the operating system can hide the latency caused by NBD. However, that latency cannot be completely hidden when all the accesses are not sequential.

C. Network Introspection

To examine the performance of the offloaded Snort, we measured a packet loss rate of Snort when we sent as many packets as possible to the VM. We used the traffic generator named D-ITG 2.8.0-rc1 [36]. Since we could not compile D-ITG for the system in our VM, we prepared another VM where we installed the same system as the host in this experiment. The results are shown in Fig. 21. The packet loss rate of Snort in KVM_host was 41% lower than that in Xen_host. Dom0 can access the virtual NIC for the VM without virtualization overhead, but all the resources but the devices are virtualized because Dom0 is

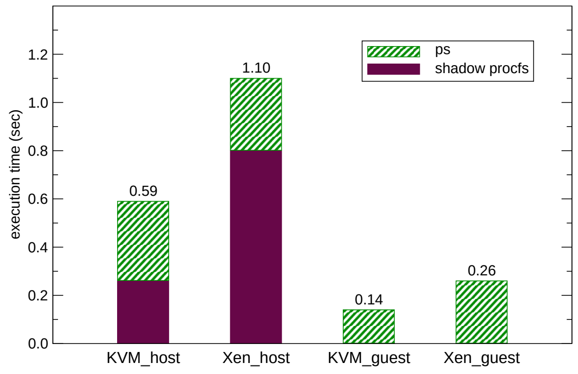

Figure 22. The execution time of the offloaded ps.

also a VM. Therefore, Snort running in Dom0 suffered from virtualization overhead.

D. Ps Command

For KVM and Xen, we measured the execution time of the ps command offloaded using Transcall. The ps command is not an IDS, but it is often used for IDSes to obtain process information. Chkrootkit in the next subsection is an example. For comparison, we also measured the execution times of the ps command running inside the VM. We ran ps 10 times for each configuration.

The average execution times were shown in Fig. 22. KVM_host was 46% faster than Xen_host. The execution times include creating the shadow proc filesystem in Transcall. For KVM_host and Xen_host, The creation times were 0.26 and 0.80 second, respectively. The cause of the differences between KVM and Xen is mainly this time. Compared with KVM_guest and Xen_guest, KVM_host and Xen_host were 4.2 times slower, respectively.

E. Chkrootkit

We measured the execution time of chkrootkit offloaded using Transcall. Since chkrootkit monitors files as well as the internal data of the guest kernel in a VM, we erased the filesystem cache whenever we ran chkrootkit. For disk images, we used the qcow2 format for KVM and the raw format for Xen, which are the defaults. For comparison, we also measured the execution time of chkrootkit running inside the VM. We ran chkrootkit 10 times for each configuration.

The average execution times were shown in Fig. 23. In addition to the time for creating the shadow proc filesystem, KVM_host and Xen_host include the time for mounting the disk image of the VM. The ratio to the total execution time was smaller: 5.8% in KVM_host and 5.4% in Xen_host. Including these setup times, the execution time of KVM_host was 63% shorter than that of Xen_host. Compared with KVM_guest, the execution time of KVM_host was 2 times longer. On the other hand, the execution time of Xen_host was 2.7 times longer than that of Xen_guest. Therefore, the

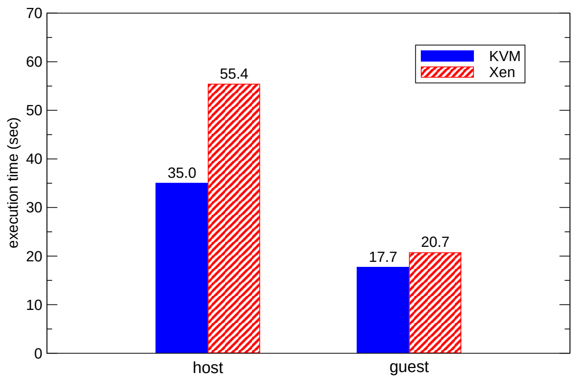

Figure 23. The execution time of the offloaded chkrootkit.

performance degradation due to IDS offloading was smaller in KVM.

VI. Related Work

VM introspection in KVM has been studied by several researchers. However, SIM [22], process implanting [23], and process out-grafting [24] securely run IDSes and their target system in the same VM to reduce the overhead of VM introspection. This is called an in-VM approach, while ours is called an out-of-VM approach. For SIM, the paper compared the performance between the in-VM and out-of-VM approaches, but the comparison was done only for active monitoring by hooks. For example, monitoring process creation and tracing system calls are done. The out-of-VM approach is not good at such active monitoring because VM introspection is done very frequently. For the performance comparison, a hypercall for mapping a memory page like Xen was implemented in the KVM module, but the performance is much low.

There are several studies for enabling memory introspection in QEMU or KVM. VMwatcher [4] monitors a temporary memory file created in QEMU with KQEMU [37] support. For QEMU without KQEMU support, the authors built their own library for allowing VMwatcher to access the physical memory of a VM. Pathogen [25] maps a file for the physical memory of a VM by modifying QEMU-KVM. However, the implementation details of them are not presented and any experiments are not performed. EXTERIOR [18] extended KVM and QEMU to collect the value of the CR3 register, but there are no implementation details in the paper.

LibVMI [21] is an open source library for VM introspection and supports both Xen and KVM. It is an updated version of XenAccess [20] supporting only Xen. LibVMI provides two methods for accessing the physical memory of a VM. One is using the existing QMP command for dumping memory contents in text. The other is adding a new QMP command for obtaining memory contents to QEMU-KVM. The latter method is faster than the former, but it requires a patch for QEMU-KVM. Nevertheless, the performance of memory introspection is low because memory contents are transferred from QEMU-KVM to IDSes. In contrast, KVMonitor can directly access the physical memory of a VM through the memory-mapped file for the memory.

Volatility [38] is an open source memory forensics framework written in Python and supports memory dumps from Windows, Linux, and OS X. It provides virtual address translation, analysis of the kernel memory, an API for adding new functionality. Although it is not used for VM introspection by itself, XenAccess and LibVMI provide Python adapters called PyXa [20] and its successor PyVMI [21], respectively. Using them, Volatility can perform VM introspection for running VMs in Xen and KVM. However, compared with the implementation in C, the performance of VM introspection may be lower.

IDS offloading has been often studied for virtualization at a hardware level, as in Xen and KVM. In contrast, HyperSpector [39] achieves IDS offloading for virtualization at the operating system level, as in Linux Containers (LXC). It runs IDSes and their target system in two different virtual execution environments, respectively. Since these environments share the operating system, IDSes can monitor the target system efficiently. In KVM, it is challenging to efficiently monitor a strongly isolated VM.

VII. Conclusion

In this paper, we first proposed KVMonitor for achieving efficient VM introspection in KVM. KVMonitor enables offloaded IDSes to access the memory, virtual disks, and virtual networks of a VM. For memory introspection, KVMonitor was 32 times faster than LibVMI. Using KVMonitor, we have ported Transcall for Xen to KVM and showed that legacy IDSes could detect intrusions. Next, we compared the performance of VM introspection between KVM and Xen. The kernel memory checking with KVMonitor was 118 times faster than that in Xen. The execution time of the offloaded Tripwire was almost the same for the same disk format. The execution of the offloaded Tripwire was 20% slower in KVMonitor when we used the default disk format, but the execution time was almost the same for the same disk format. From these results, we concluded that VM introspection in KVM was more efficient than or the same as that in Xen.

One of our future work is to compare how offloaded IDSes affect the performance of monitored VMs and how the workloads in monitored VMs affect offloaded IDSes between Xen and KVM. In addition, we need to conduct performance comparison with other virtualization software such as VMware and Microsoft Hyper-V. Another future work is to integrate the memory introspection in KVMonitor with LibVMI. Through the interface of PyVMI, we could improve the performance of Volatility for running VMs in KVM.

Acknowledgment

This work was supported in part by JSPS KAKENHI Grant Number 25330086.

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] VMware, Inc., "VMware Workstation," http://www.vmware.com/products/workstation.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *Proc. Symp. Operating Systems Principles*, 2003, pp. 164–177.

[4] 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.

[5] B. Payne, M. Carbone, and W. Lee, "Secure and Flexible Monitoring of Virtual Machines," in *Proc. Annual Conf. Computer Security Applications*, 2007, pp. 385–397.

[6] B. Hay and K. Nance, "Forensics Examination of Volatile System Data Using Virtual Introspection," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 3, pp. 75–83, 2008.

[7] B. D. Payne, M. Carbone, M. I. Sharif, and W. Lee, "Lares: An Architecture for Secure Active Monitoring Using Virtualization," in *Proc. Symp. Security and Privacy*, 2008, pp. 233–247.

[8] A. Srivastava and J. Giffin, "Tamper-resistant, Application-aware Blocking of Malicious Network Connections," in *Proc. Intl. Symp. Recent Advances in Intrusion Detection*, 2008, pp. 39–58.

[9] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware Analysis via Hardware Virtualization Extensions," in *Proc. Conf. Computer and Communications Security*, 2008, pp. 51–62.

[10] A. M. Azab, P. Ning, E. C. Sezer, and X. Zhang, "HIMA: A Hypervisor-Based Integrity Measurement Agent," in *Proc. Annual Computer Security Applications Conf.*, 2009, pp. 461–470.

[11] 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.

[12] 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.

[13] J. Hizver and T. Chiueh, "Real-time Deep Virtual Machine Introspection and Its Applications," in *Proc. Intl. Conf. Virtual Execution Environments*, 2014, pp. 3–14.

[14] Trend Micro, Inc., "Deep Security," http://www.trendmicro.com/us/enterprise/cloud-solutions/deep-security/.

[15] VMware Inc., "Server Virtualization & Cloud Infrastructure: VMware vSphere," http://www.vmware.com/products/vsphere.

[16] F. Bellard, "QEMU," http://qemu.org/.

[17] J. Dike, "The User-mode Linux Kernel Home Page," http://user-mode-linux.sourceforge.net/.

[18] Y. Fu and Z. Lin, "EXTERIOR: Using a dual-VM Based External Shell for guest-OS Introspection, Configuration, and Recovery," in *Proc. Intl. Conf. Virtual Execution Environments*, 2013, pp. 97–110.

[19] Red Hat, Inc., "Kernel Based Virtual Machine," http://www.linux-kvm.org/.

[20] B. Payne, "XenAccess Library," http://code.google.com/p/xenaccess/.

[21] ——, "LibVMI," http://code.google.com/p/vmitools/.

[22] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-VM Monitoring Using Hardware Virtualization," in *Proc. Conf. Computer and Communications Security*, 2009, pp. 477–487.

[23] Z. Gu, Z. Deng, D. Xu, and X. Jiang, "Process Implanting: A New Active Introspection Framework for Virtualization," in *Proc. Intl. Symp. Reliable Distributed Systems*, 2011, pp. 147–156.

[24] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu, "Process Out-grafting: An Efficient "out-of-VM" Approach for Fine-grained Process Execution Monitoring," in *Proc. Conf. Computer and Communications Security*, 2011, pp. 363–374.

[25] A. Roberts, R. McClatchey, S. Liaquat, N. Edwards, and M. Wray, "Introducing Pathogen: A Real-Time Virtual Machine Introspection Framework," in *Proc. Conf. Computer and Communications Security*, 2013, pp. 1429–1432.

[26] X. Jiang and X. Wang, ""Out-of-the-Box" Monitoring of VM-based High-interaction Honeypots," in *Proc. Intl. Conf. Recent Advances in Intrusion Detection*, 2007, pp. 198–218.

[27] 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.

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

[29] Y. M. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski, "Detecting Stealth Software with Strider GhostBuster," in *Proc. Intl. Conf. Dependable Systems and Networks*, 2005, pp. 368–377.

[30] G. Kim and E. Spafford, "The Design and Implementation of Tripwire: A File System Integrity Checker," in *Proc. ACM Conf. Computer and Communications Security*, 1994, pp. 18–29.

[31] N. Murilo and K. Steding-Jessen, "chkrootkit – Locally Checks for Signs of a Rootkit," http://www.chkrootkit.org/.

[32] K. Kourai and H. Utsunomiya, "Synchronized Co-migration of Virtual Machines for IDS Offloading in Clouds," in *Proc. Intl. Conf. Cloud Computing Technology and Science*, 2013, pp. 120–129.

[33] M. Roesch, "Snort – Lightweight Intrusion Detection for Networks," in *Proc. USENIX System Administration Conf.*, 1999.

[34] G. Lyon, "Nmap – Free Security Scanner For Network Exploration & Security Audits," http://nmap.org/.

[35] W. D. Norcott, "IOzone Filesystem Benchmark," http://www.iozone.org/.

[36] A. Botta, A. Dainotti, and A. Pescapè, "A Tool for the Generation of Realistic Network Workload for Emerging Networking Scenarios," *Computer Networks (Elsevier)*, vol. 56, no. 15, pp. 3531–3547, 2012.

[37] F. Bellard, "QEMU Accelerator," http://wiki.qemu.org/KQemu/Doc.

[38] A. Walters, "Volatility – An Advanced Memory Forensics Framework," http://code.google.com/p/volatility/.

[39] K. Kourai and S. Chiba, "HyperSpector: Virtual Distributed Monitoring Environments for Secure Intrusion Detection," in *Proc. Intl. Conf. Virtual Execution Environments*, 2005, pp. 197–207.