IPSJ Transactions on Advanced Computing Systems Vol.7 No.4 45–55 (Dec. 2014)

Regular Paper

Co-migration of Virtual Machines with Synchronization for IDS Offloading

Kenichi Kourai[1,a)] Hisato Utsunomiya[1,b)]

Received: April 4, 2014, Accepted: September 11, 2014

Abstract: Since Infrastructure-as-a-Service (IaaS) clouds contain many vulnerable virtual machines (VMs), intrusion detection systems (IDSes) should be run for all the VMs. *IDS offloading* is promising for this purpose because it allows IaaS providers to run IDSes outside of VMs without any cooperation of the users. However, offloaded IDSes cannot continue to monitor their target VM when the VM is migrated to another host. In this paper, we propose *VMCoupler* for enabling co-migration of offloaded IDSes and their target VM. Our approach is running offloaded IDSes in a special VM called a *guard VM*, which can monitor the internals of a target VM using *VM introspection*. VMCoupler can migrate a guard VM together with its target VM and restore the state of VM introspection at the destination. The migration processes of these two VMs are synchronized so that a target VM does not run without being monitored. We have confirmed that the overhead of monitoring and co-migration was small.

Keywords: IaaS clouds, virtual machines, migration, intrusion detection systems

1. Introduction

Infrastructure as a service (IaaS) such as Amazon EC2 provides virtual machines (VMs) for users. They set up their own operating systems and applications in the VMs. Unfortunately, the systems inside VMs are not always well maintained and can be penetrated by attackers. To protect such systems, intrusion detection systems (IDSes) are useful. They can monitor the operating systems, networks, and storage of VMs and then alert administrators to attacks if they detect symptoms of intrusion. However, it is difficult for IaaS providers to enforce users to install IDSes in their VMs. Even if users install IDSes, intruders into VMs can easily disable such IDSes running in the VMs before attacking against the systems in them.

To solve these problems, IaaS providers can use *IDS offloading* with *VM introspection* [1], [2], [3], [4], [5]. This technique enables IDSes to run outside of their target VM and monitor the VM securely. IDS offloading allows IaaS providers to run IDSes for VMs without any cooperation of users. Using VM introspection, offloaded IDSes can directly obtain detailed information inside VMs. They are protected from intruders in VMs. However, when the target VM of the IDSes is migrated to another host, the IDSes cannot continue to monitor the VM because they are not migrated together with the VM. Consequently, IaaS providers cannot use both IDS offloading and VM migration.

In this paper, we propose *VMCoupler* [6], which is the system for enabling co-migration of offloaded IDSes and their target VM. Our idea is running offloaded IDSes in a special VM called a *guard VM* and migrating a guard VM together with its target VM. A guard VM enables IDSes to monitor the internals of a target VM using VM introspection. VMCoupler performs co-migration of a guard VM and its target VM, while the guard VM continues to monitor the target VM. To achieve this, VMCoupler preserves the state of VM introspection in a guard VM during co-migration. In addition, VMCoupler synchronizes the migration processes of these two VMs for security. This guarantees that a guard VM always monitors its target VM while the target VM is running.

We have implemented VMCoupler in Xen 4.0.1 [7]. For memory monitoring, VMCoupler allows a guard VM to map memory pages of its target VM. After the co-migration, it restores the memory-mapping state at a destination host. For network monitoring, VMCoupler performs port mirroring at a virtual switch for a guard VM to capture the packets to/from a target VM. It sets up port mirroring again after co-migration. By using networked storage, a guard VM can monitor the storage of its target VM even after co-migration. We conducted several experiments to examine the overhead of monitoring and co-migration and confirmed that the overhead was small.

The rest of this paper is organized as follows. Section 2 describes the issues in IDS offloading and VM migration and then discusses the existing approaches. Section 3 proposes the system for achieving co-migration of offloaded IDSes and their target VM, which is called VMCoupler. Section 4 describes the implementation of VMCoupler in Xen. Section 5 reports the performance of offloaded IDSes and co-migration. Section 6 concludes the paper.

2. Background

2.1 IDS Offloading and VM Migration

Although IDSes play an important role in IaaS clouds as well

[1] Kyushu Institute of Technology, Iizuka, Fukuoka 820–8502, Japan
[a)] kourai@ci.kyutech.ac.jp
[b)] U_SAN@ksl.ci.kyutech.ac.jp

as in traditional systems, it is difficult that IaaS providers enforce users to install IDSes in their VMs. In IaaS clouds, providers just provide VMs, while users determine installed software. Therefore providers cannot install any software including IDSes without users' cooperation. They can require users to install IDSes, but some users may not follow such a requirement for various reasons, e.g., performance overhead or less administrative skills. Even if users cooperatively install IDSes, such IDSes can be disabled easily by intruders into VMs. Intruders with sufficient privileges can stop IDSes or make IDSes ineffective.

IDS offloading is attractive to IaaS providers in that they can deploy IDSes without any cooperation of users. It enables modular and secure monitoring of VMs. It runs IDSes outside of their target VM and prevents interferences from intruders in the VM. Using VM introspection, offloaded IDSes can monitor the internals of the operating system, network packets, and the file systems of a target VM with no agent software installed. For example, the integrity checker of the kernel memory can detect tampering with the kernel by intruders in a VM. Tripwire [8] can detect tampering with the file systems by intruders in a compromised VM. Snort [9] can detect malicious packets sent from outside attackers and intruders in a compromised VM. In this paper, we consider both interval-based IDSes, which are run periodically such as integrity checkers, and real-time IDSes, which are driven by events such as network traffic. Such offloaded IDSes are often run in a privileged VM called the *management VM*, which is used for managing VMs, e.g., in Xen.

On the other hand, IaaS clouds migrate VMs for various purposes. VM migration allows a running VM to be moved from a source to a destination host. In particular, live migration [10] almost does not stop a VM during the migration process by transferring most of its states with the VM running. Using VM migration, IaaS providers can maintain physical hosts without interrupting services provided by VMs. They can perform load balancing by migrating heavily loaded VMs to other lightly loaded hosts. Conversely, they can save power if they consolidate lightly loaded VMs into a fewer hosts.

When a VM is migrated, the IDSes offloaded from the VM are not migrated together to the same destination host. Unlike IDSes running inside a VM without offloading, offloaded IDSes are not considered at VM migration. As a result, the target VM at a destination host would run without monitoring by offloaded IDSes, as shown in **Fig. 1**. If attackers intrude into the VM, IaaS providers cannot detect that intrusion. To avoid such an insecure situation, IaaS clouds cannot use VM migration with IDS offloading. This would make IaaS clouds lose various advantages of using VMs.

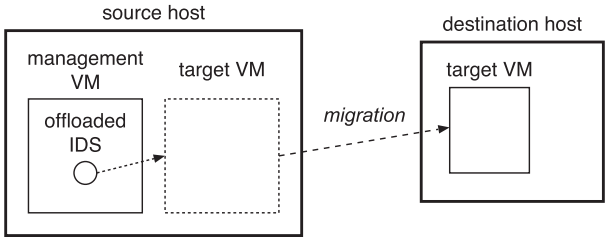


Fig. 1 VM migration in IDS offloading.

To use both IDS offloading and VM migration securely, it is important to guarantee that offloaded IDSes can *always* monitor their target VM. This is obvious for real-time IDSes. For example, network-based IDSes would fail to capture packets if they cannot monitor a target VM for a certain period. For interval-based IDSes, some readers may think that such a strong guarantee is not necessary. However, there is a tradeoff between monitoring delays and detectability in such IDSes. As a monitoring delay is shorter, IDSes can detect malicious activities earlier and more correctly. Then damages to their target VM can be minimized. To support IDSes with maximum detectability, a mechanism for continuous monitoring of a target VM is necessary.

For continuous monitoring, offloaded IDSes should be co-migrated with their target VM at the same time, but migrating offloaded IDSes is not so easy. One possible approach is migrating the management VM where IDSes are offloaded. However, the management VM is a special VM and is not migratable. Since only one management VM has to exist in one host, it cannot be moved out or in. Another approach is offloading IDSes to a regular VM that is different from their target VM. Although a regular VM can be migrated, IDSes in a regular VM cannot monitor the target VM because a regular VM does not have such privileges.

2.2 Existing Approaches

We describe the existing approaches about three aspects: (1) privileged VMs usable for offloading IDSes, (2) co-migration of offloaded IDSes and their target VM, and (3) continuous monitoring of a target VM without migrating offloaded IDSes.

2.2.1 Privileged VMs for Offloading IDSes

For Xen, various privileged VMs have been proposed to divide the privileges of the management VM called Dom0. Driver domains [11] run device drivers in VMs different from Dom0. IDSes can be run in driver domains to monitor networks and storage. Stub domains [12], [13] enable running QEMU used for device emulation. Since they are allowed to access the memory of regular VMs called DomUs, IDSes in them can monitor the memory. In addition, they can intercept device accesses and check the integrity. DomB [14] is used to boot DomU, instead of Dom0. It loads a kernel image into the DomU's memory and sets up DomU. Xoar [15] disaggregates Dom0 into many single-purpose VMs called service VMs. However, all of these privileged VMs are not designed to be migratable because they are helpers for Dom0.

A self-service cloud (SSC) computing platform [16] 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. In SSC, user's meta-domain consists of DomUs, SDs, and other related VMs. These VMs should be migrated together because of their strong association, but SSC does not support such co-migration.

2.2.2 Co-migration of Offloaded IDSes

Instead of migrating a VM that runs offloaded IDSes, migrating only IDS processes is possible. Even if the management VM cannot be migrated, its processes can be migrated. Process migration has been well studied [17], but it cannot preserve the monitor-

ing states of a target VM during the migration of IDS processes. Although offloaded IDS processes map the memory of another VM, the operating system in the management VM cannot preserve the mapping state across VM migration. In addition, most of the systems supporting process migration such as libckpt [18] and BLCR [19] do not preserve process states completely. For example, open files and sockets are usually closed. Distributed operating systems such as Amoeba [20] and MOSIX [21] can migrate processes with their states preserved, but I/O requests are simply forwarded to a source host. Therefore the source host cannot be stopped.

Compute capsules [22] and the pod abstraction in Zap [23] enable a group of processes to be migrated as a unit. They provide a thin virtualization layer on top of the operating system and group processes with a private namespace. In Zap, particularly, migrated processes can preserve network connections and interprocess communication between them, including shared memory. However, like the other systems supporting process migration, it cannot migrate offloaded IDS processes with the memory-mapping state of a target VM.

For concurrent migration of multiple co-located VMs, live gang migration [24] has been proposed. It transfers memory contents that are identical across VMs only once to reduce the migration overhead. It tracks identical memory contents across VMs and performs memory de-duplication for all the migrated VMs. It also applies differential compression to nearly identical memory pages. Unlike VMCoupler, live gang migration does not synchronize between the migration processes of multiple VMs. This approach can be incorporated into our VMCoupler to reduce the co-migration time.

2.2.3 Continuous Monitoring

An approach different from co-migrating offloaded IDSes is re-executing offloaded IDSes at the destination host where their target VM is migrated. If IDSes are short-lived, this approach works well. However, re-execution is not feasible for long-running IDSes such as Tripwire. When a target VM is migrated, the execution of such IDSes is aborted at the source host and is restarted from the beginning at the destination host. At this time, for example, Tripwire has to examine many files again. Similarly, memory forensic tools may have to analyze the whole kernel data again. Such wasteful resource consumption should be avoided.

Trend Micro Deep Security [25] can continue to apply its policies after a target VM is migrated. It works with VMware vShield Endpoint [26], which provides security virtual appliances (VMs) for offloading IDSes. Although virtual appliances cannot be migrated, Deep Security hands over its policies between virtual appliances of source and destination hosts when a target VM is migrated. Deep Security can migrate its policies for long-running IDSes, but it cannot migrate the monitoring state of a target VM. In addition, IDSes have to be re-designed for the handover of policies in this approach.

3. VMCoupler

We propose *VMCoupler* [6] for enabling co-migration of offloaded IDSes and their target VM. VMCoupler provides a special VM called a *guard VM* to run offloaded IDSes. It migrates a guard VM and its target VM together.

3.1 Guard VM

A guard VM possesses monitoring capabilities for running offloaded IDSes, as illustrated in **Fig. 2**. For memory monitoring, it allows IDSes to map memory pages of their target VM. IDSes in a guard VM can read the contents of the mapped memory pages and monitor the state of the target VM. Traditionally, only the management VM was allowed this kind of monitoring. For network monitoring, a guard VM allows IDSes to capture the packets from/to their target VM. To achieve this, VMCoupler performs port mirroring at a virtual switch. Port mirroring duplicates the packets of a target VM to its guard VM. A guard VM provides a dedicated network interface for receiving the duplicated packets. For storage monitoring, a guard VM allows IDSes to read the networked storage used by its target VM. To enable a target VM to be migrated, networked storage is usually used so that the VM can access its storage at both source and destination hosts.

VMCoupler gives least privilege to a guard VM so that the VM can monitor only one target VM. The management VM binds a guard VM to its target VM and allows the guard VM to map only memory pages of the target VM. It configures port mirroring at a virtual switch so that only the packets of a target VM are delivered to its guard VM. By the access control of networked storage, a guard VM can access only the storage of its target VM. Even if attackers penetrate a guard VM, they can steal information only from its target VM. In this sense, IDS offloading to a guard VM is more secure than that to the management VM. Since the management VM has full privileges for all the VMs, the whole system is compromised if the management VM is compromised.

3.2 Co-migration with Continuous Monitoring

For the continuity of the monitoring, VMCoupler co-migrates a guard VM and its target VM. It groups these two VMs and migrates them in parallel. If we migrate a guard VM just like a regular VM, most of the monitoring states that the guard VM has would be lost. The mapping state of the target VM's memory is not migrated because the traditional migration mechanism assumes that a VM is self-contained. In other words, it is assumed that a VM maps only its own memory pages. The state of port mirroring is also not migrated because the configuration is done in a virtual switch, which is located outside of the VM, at the source host.

VMCoupler restores all the monitoring states at a destination host so that a guard VM continues to monitor its target VM. If

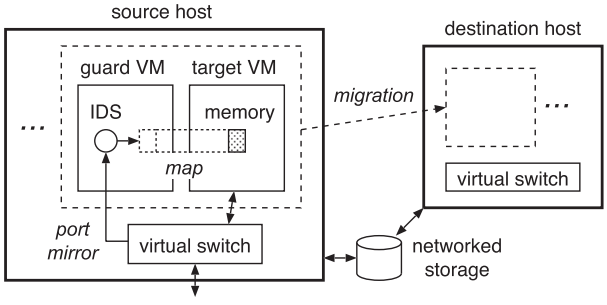


Fig. 2 Co-migration of a guard VM and its target VM in VMCoupler.

a guard VM maps memory pages of the target VM at a source host, VMCoupler transfers the mapping state to a destination host. Then it remaps memory pages of the target VM to the address spaces of offloaded IDS processes. In addition, VMCoupler reconfigures port mirroring for the packets to/from the target VM at the virtual switch of the destination host. For storage monitoring, a guard VM can continue to monitor the networked storage used by the target VM after the migration. Since the network connection to the networked storage is preserved, VMCoupler does not need to provide any special mechanism for restoring the monitoring state of storage.

3.3 Synchronized Co-migration

To migrate a single VM, a migration manager in the management VM transfers all the VM state from a source to a destination host. First, it creates a new empty VM at a destination host to store the transferred state. Then it transfers the memory of the original VM to a destination host. In live migration, particularly, the migration manager repeats to transfer dirty pages modified inside the original VM because the original VM at a source host is running during migration. When the number of dirty pages to be transferred becomes small enough, the migration process enters the final stage of live migration. The migration manager stops the original VM at a source host and transfers the remaining dirty pages and the other states. Finally, it restarts the new VM at a destination host and terminates the original VM at a source host.

To achieve secure and safe co-migration of a guard VM and its target VM, there are two requirements. One is that a guard VM can *always* monitor its target VM while its target VM is running. If either a guard VM or a target VM has been migrated earlier, the guard VM could not monitor the target VM running at a different host. Since we assume that the memory size of a guard VM is usually smaller than that of a target VM, a guard VM would have been migrated earlier. A guard VM requires a small amount of memory to run only offloaded IDSes, while a target VM requires a larger amount of memory to run various services. The migration time strongly depends on the memory size of a VM. For example, when a target VM is an r3.8xlarge instance with 244 GiB in Amazon EC2, it would take a very long time to migrate the VM. Even for interval-based IDSes, it is too dangerous if offloaded IDSes cannot monitor their target VM for such a long time.

The other requirement is that the migration manager for a guard VM can always obtain the necessary information on a target VM. If a target VM has been migrated and been terminated earlier than a guard VM, the migration manager for the guard VM at a source host could not examine information on the target VM after that.

To satisfy these two requirements, VMCoupler synchronizes the migration processes of both a guard VM and a target VM, as illustrated in **Fig. 3**. For security, there are two synchronization points S_1 and S_4 at source and destination hosts, respectively. S_1 is the point to wait for target VM's stop before stopping a guard VM. A migration manager reaches this point when it has transferred most of the memory of a VM and is ready for the final stage of live migration. The synchronization at this point guarantees that a guard VM stops after a target VM and that it can monitor a target VM as long as the target VM is running. In contrast,

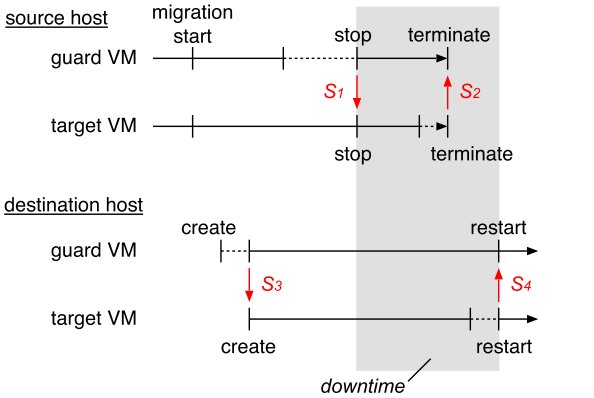


Fig. 3 Four synchronization points S_1 to S_4 during co-migration. A dashed line stands for a period waiting for synchronization. A gray period is the downtime of both a guard VM and a target VM.

S_4 is the point to wait for guard VM's restart before restarting a target VM. A migration manager reaches this point when it completes reconstructing a migrated VM and is ready for the restart. The synchronization at this point guarantees that a target VM is restarted after a guard VM and that it is monitored by a guard VM just after its restart.

For safety, there are also two synchronization points S_3 and S_2 at destination and source hosts, respectively. S_3 is the point to wait for target VM's creation before creating a guard VM. To restore the mapping of the target VM's memory in a guard VM, a target VM has to have been created. The synchronization at this point guarantees that. In contrast, S_2 is the synchronization point to wait for guard VM's termination before terminating a target VM. A migration manager reaches this point when all states have been transferred. The synchronization at this point guarantees that the migration manager for a guard VM can continue to obtain information on a target VM until the migration of a guard VM completes.

4. Implementation

We have implemented VMCoupler in Xen 4.0.1 [7]. In Xen, the virtual machine monitor (VMM) runs on top of hardware and executes VMs. As described in Section 2.2.1, the management VM is called *Dom0* and a regular VM including a target VM is called *DomU* in Xen. We have developed a guard VM by extending DomU and we call it *DomM*. DomM runs para-virtualized Linux for monitoring the memory of DomU. In the current implementation, VMCoupler supports para-virtualized Linux running in DomU and targets the x86-64 architecture.

4.1 Memory Monitoring

In Xen, the VMM distinguishes machine memory and pseudo-physical memory to virtualize memory resources. Machine memory is physical memory installed in a host and consists of a set of machine page frames. For each machine page frame, a machine frame number (MFN) is consecutively numbered from 0. Pseudo-physical memory is the memory allocated to VMs and gives the illusion of contiguous physical memory to VMs. For each physical page frame in each VM, a physical frame number (PFN) is consecutively numbered from 0. The VMM maintains the machine-to-physical (M2P) table for the translation from

MFNs to PFNs. Para-virtualized Linux maintains the physical-to-machine (P2M) table for translating PFNs to MFNs.

Traditionally, Dom0 maps memory pages of DomU by issuing the update_va_mapping hypercall to the VMM. A hypercall is a mechanism to invoke the function of the VMM. The VMM modifies the page table in Dom0 to map the page specified by an MFN. To obtain such an MFN, Dom0 usually translates a DomU's virtual address by traversing the page tables in DomU. The page directory entry in DomU is obtained by the domctl hypercall. However, VMs except Dom0 could not map memory pages of DomU because only Dom0 could issue these hypercalls.

We modified the VMM so that it allows not only Dom0 but also DomM to issue these hypercalls. In VMCoupler, Dom0 registers a pair of DomM and its target DomU to the VMM by the domctl hypercall. Thereby DomM can issue these hypercalls only to the specified DomU. In addition, we modified the Linux kernel in DomM so that IDS processes can map the memory of DomU (**Fig. 4**). IDS processes execute such memory mapping through the privcmd interface provided by the Linux kernel para-virtualized for Xen. Since the original implementation of privcmd allowed only Dom0 to use its functions, we modified it so that DomM can also use it.

4.2 Network Monitoring

When DomU is created, a pair of virtual network interfaces (e.g., `vif1.0` and `eth0`) is created. These are assigned to Dom0 and DomU, respectively, as illustrated in **Fig. 5**. In the bridge-networking mode, `vif1.0` is connected to a network bridge in Dom0, which is also connected to physical network interfaces (e.g., `peth0`). When a packet is sent from DomU, it is delivered to `vif1.0` and is transmitted to the outside via the network bridge. When a packet to DomU is sent from the outside, it is delivered to `vif1.0` via the network bridge. When a packet is sent to another DomU at the same host, it is delivered from `vif1.0` to another virtual network interface via the network bridge. Therefore, Dom0 can easily capture all the packets from/to DomU via `vif1.0`. However, it is not easy for VMs except Dom0 to capture packets because any packets are not delivered via these VMs.

To enable DomM to capture the packets from/to DomU, Dom0 duplicates the packets by achieving port mirroring with traffic control [27]. This idea comes from port mirroring in KVM [28]. **Figure 6** depicts port mirroring in Dom0. The traffic shaping in Linux is performed in queuing disciplines (qdisc), which is attached to a network device. A qdisc enqueues all the packets and dequeues them according to registered filters. For port mirroring, Dom0 first creates an additional virtual network interface (e.g., `vif2.1`) as a mirror port for DomM. Then it attaches a qdisc to the virtual network interface for DomU (e.g., `vif1.0`). The qdisc receives all the packets via the virtual network interface from/to DomU. It duplicates these packets to the mirror port and DomM can receive them via its additional network interface for port mirroring (e.g, `eth1`).

More specifically, Dom0 issues traffic control commands as in **Fig. 7**. Commands (1) and (2) are used for duplicating outgoing packets from DomU, whereas commands (3) and (4) are used for incoming packets to DomU. By command (1), Dom0 attaches a new qdisc of type `ingress` for inbound packets to `vif1.0`. By command (2), Dom0 creates a new filter matching all IP packets and attaches it to the `ingress` qdisc, which is specified by the ID `ffff:`. When the filter matches, the action `mirred` is executed and all inbound packets are copied to `vif2.1`. By command (3), Dom0 replaces the top-level qdisc with a new qdisc of type `prio` so that a filter can be attached. By command (4), Dom0 creates a new filter matching all IP packets and attaches it to the `prio` qdisc whose ID is, for example, `8002:`. When the filter matches, all outbound packets are copied to `vif2.1`.

4.3 Storage Monitoring

Since DomU has to be migrated, its virtual disks are usually located in networked storage such as an NFS server. The simplest storage configuration for monitoring such virtual disks is

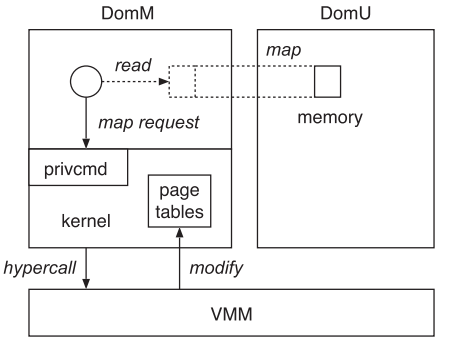


Fig. 4 Memory monitoring via the privcmd interface.

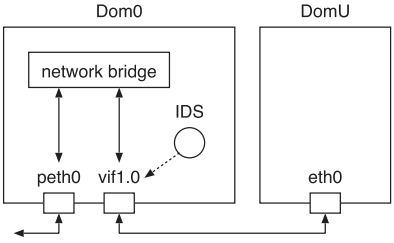


Fig. 5 Traditional network monitoring in Dom0.

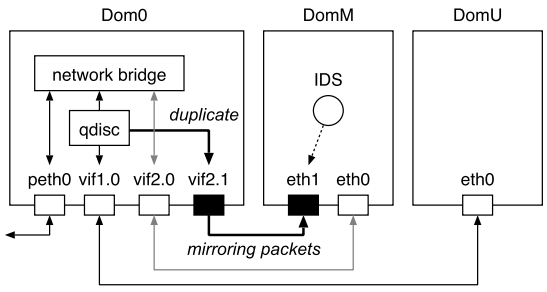


Fig. 6 Network monitoring in DomM with port mirroring.

```
(1) tc qdisc add dev vif1.0 ingress
(2) tc filter add dev vif1.0 parent ffff: \
      protocol ip u32 match u8 0 0 \
      action mirred egress mirror dev vif2.1
(3) tc qdisc replace dev vif1.0 parent root prio
(4) tc filter add dev vif1.0 parent 8002: \
      protocol ip u32 match u32 0 0 \
      action mirred egress mirror dev vif2.1
```

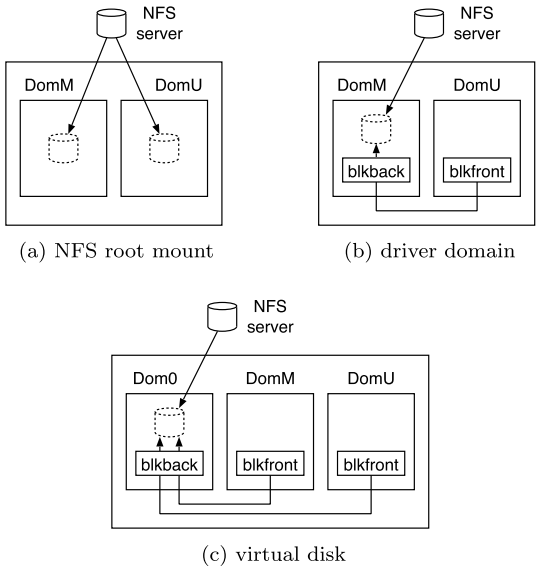
Fig. 7 Traffic control commands for port mirroring.

that both DomU and DomM mount the NFS root file system for DomU, as shown in **Fig. 8** (a). One disadvantage of this configuration is that DomU has to be configured so that it uses NFS. In IaaS clouds, it is often difficult for IaaS providers to enforce using specific file systems to the users. Users would like to use their favorite file systems.

Another configuration is that DomM mounts an NFS volume including the disk images of DomU and provides them to DomU as virtual disks, as shown in Fig. 8 (b). This means that DomM is configured as a driver domain [11] to serve virtual disks to DomU. DomU accesses the virtual disks through the blkfront driver in DomU and the blkback driver in DomM. DomM can easily monitor the disk images as Dom0 can traditionally. However, this configuration can affect the storage performance of DomU. To access virtual disks, DomU has to communicate with DomM. Then DomM has to access an NFS server via Dom0 because the DomM's network is virtualized as in Fig. 5.

A promising configuration is that Dom0 mounts the NFS volume for DomU and provides them to not only DomU but also DomM, as shown in Fig. 8 (c). The advantages of this configuration are that DomU does not need to change its storage configuration and that it can access its disks without the overhead of DomM. However, the monitoring performance in DomM is lower than that when DomM directly mounts the NFS volume, according to our experiment in Section 5.2.

Therefore, we adopted the configuration shown in **Fig. 9**. Dom0 mounts the NFS volume for DomU and provides DomU with the disk images in it as virtual disks. Also, DomM mounts the same NFS volume and monitors the disk images in it by loopback mounts. The storage performance in DomU is the same as that in the traditional system without DomM.



(a) NFS root mount (b) driver domain

(c) virtual disk

Fig. 8 Possible configurations for storage monitoring.

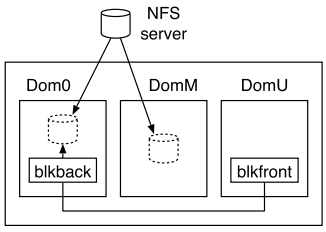


Fig. 9 Storage monitoring using an NFS server.

4.4 Migration of DomM

To migrate DomM, the migration manager in Dom0 transfers the memory of DomM from a source to a destination host. During such memory transfers, the migration manager canonicalizes the page table entries (PTEs) used by the DomM at a source host. This canonicalization is rewriting PTEs so that the page tables do not depend on host-specific memory allocation. Specifically, the migration manager replaces the host-specific MFNs in the PTEs with the corresponding PFNs. This translation is performed with the M2P table in the VMM. At a destination host, the migration manager uncanonicalizes those PTEs so that DomM can run with the host-specific page tables. However, if DomM maps the memory pages of DomU, the uncanonicalization fails at the destination host. After the canonicalization, the page tables of DomM have entries including the PFNs belonging to DomU. When the migration manager uncanonicalizes them, it cannot distinguish DomU's PFNs from DomM's because PFNs are local in each VM.

In VMCoupler, the migration manager for DomM transfers the memory-mapping state on DomU as well. If DomM maps a memory page of DomU, the migration manager sets a *monitor bit* to the corresponding PTE at the canonicalization, as illustrated in **Fig. 10**. To examine if a PTE is used for mapping a memory page of DomU, the migration manager re-translates the translated PFN into an MFN. This is done using the P2M table of DomU, which is stored in DomU's memory. If the obtained MFN is equal to the original one before the canonicalization, the migration manager can determine that the PFN belongs to DomU. Otherwise, it considers that the PFN belongs to DomM. The monitor bits are automatically transferred to the destination with memory pages used for the page tables.

At the destination host, the migration manager correctly restores the memory-mapping state using the monitor bits. If a monitor bit is set in a PTE, the migration manager considers the PFN included in the PTE as the one of DomU and replaces it with the corresponding MFN, which is allocated to DomU. **Figure 11** shows the page table after reconstruction. For this purpose, the migration manager cannot use the P2M table in DomU yet because the P2M table is reconstructed by the guest operating system itself in DomU after DomU is resumed. Therefore, the

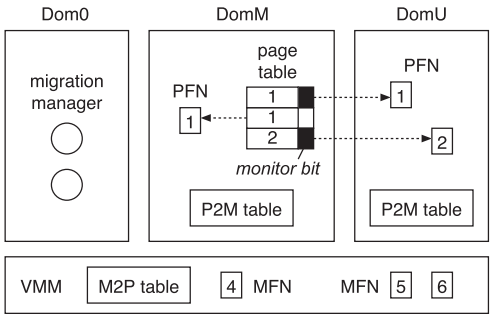


Fig. 10 Saving the memory-mapping state for DomU.

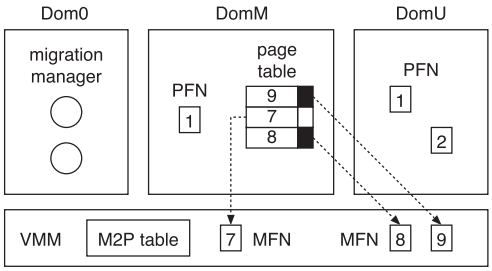


Fig. 11 Restoring the memory-mapping state for DomU.

migration manager constructs the P2M table of DomU from the list of MFNs allocated to DomU and the M2P table.

For network monitoring, the migration manager removes the filters set up for port mirroring at the source host after it stops DomM at the final stage of the migration. At the destination host, it adds the filters for port mirroring again before it restarts DomM. As such, DomM can monitor all the packets that DomU receives. For storage monitoring, DomM can continue to monitor the disk images of DomU after migration because the images are located in an NFS server. The network connection to the NFS server is kept during migration.

4.5 Synchronized Co-migration

In VMCoupler, two migration managers migrate DomU and DomM synchronously as shown in Fig. 3. First, they create new empty VMs at a destination host and synchronize their migration processes at S_3. To wait for new DomU's creation, the migration manager for DomM repeatedly looks up the DomU corresponding to the universally unique identifier (UUID) specified in its configuration. If it can find that DomU, it proceeds the migration of DomM.

When these migration managers enter the final stage of live migration, they synchronize their migration processes at S_1. To wait for DomU's stop, the migration manager for DomM repeatedly obtains information on DomU until DomU becomes the stopped state. Concurrently, it continues to transfer dirty pages of DomM to the destination. It repeats iterations of the migration until DomU stops for the final stage. In each iteration, the migration manager transfers dirty pages modified after the previous iteration. This can keep the number of dirty pages to be transferred at the final stage as small as possible.

In the current implementation, the migration manager for DomU does not wait for DomM when it is ready for the final stage earlier. One reason is that we assume that the memory transfer for DomM completes in a shorter time than that for DomU. Usually the memory size of DomM is smaller than that of DomU because DomM runs only offloaded IDSes. The other reason is that we have implemented one-way synchronization by examining only the VM state, e.g., stopped and running, for simplicity. We believe that it is not so difficult to implement two-way synchronization, but that is our future work.

When the migration managers terminate the original VMs at the source host, they synchronize the tasks at S_2. To wait for DomM's termination, the migration manager for DomU repeatedly checks the existence of DomM. Finally, they synchronize the restarts of the migrated VMs at S_4. The migration manager for DomU repeatedly examines the state of DomM until DomM becomes the running state.

5. Experiments

We conducted experiments to examine the continuity of monitoring across co-migration and to measure the performance of monitoring and co-migration. For server machines hosting VMs, we used two PCs with one Intel Quad Core 2.83 GHz processor, 8 GB of memory, and gigabit Ethernet. We used Xen 4.0.1 and ran Linux 2.6.32 in Dom0, DomM, and DomU. By default, we allocated one virtual CPU and 512 MB of memory to DomM, one virtual CPU and 1 GB of memory to DomU, and four virtual CPUs and the rest of the memory to Dom0. For an NFS server, we used NAS with one Intel Xeon X5640 3.16 GHz processor, 32 GB of memory, 16 TB of RAID-5 disks, and gigabit Ethernet. For a client machine, we used a PC with one Intel Xeon E5630 2.53 GHz processor, 4 GB of memory, and gigabit Ethernet. We used Xen 4.3.2 and ran Linux 3.13.0 in Dom0. These PCs and NAS are connected with a gigabit Ethernet switch.

5.1 Memory Monitoring

We executed the integrity checker of the DomU kernel in DomM. The integrity checker calculates the hash value of the memory area for the kernel text and detects tampering with it. We compared the hash value with the one pre-calculated from the kernel image and confirmed that the integrity checker could correctly monitor the kernel in DomU. Even if we co-migrated DomM and DomU during the integrity check, the checker could continue to run and complete its check at the destination host.

Next, we measured the time needed for the integrity check. For comparison, we executed the integrity checker in Dom0 as traditional and measured the time. We ran the integrity checker 100 times. On average, it took 135 ms and 203 ms for the integrity checks in DomM and Dom0, respectively. Unexpectedly, the integrity check running in DomM was 33% faster than that in Dom0.

According to our analysis of the implementation in Xen and Linux, we found that the number of virtual CPUs allocated to a VM affected the performance of memory mapping. When DomM and Dom0 map memory pages of DomU, they issue the hypercall for obtaining the state of virtual CPUs. At that time, they allocate a buffer passed to the hypercall and lock it by using the mlock system call so that the corresponding memory pages are not paged out. Since the system call waits for all the CPUs to synchronously complete pending operations on memory pages, the execution time is proportional to the number of CPUs.

Figure 12 shows the time for the integrity check when we changed the number of virtual CPUs allocated to DomM and Dom0. The results show that the time is proportional to the number of virtual CPUs. For four virtual CPUs, the performance in DomM was degraded largely. This is probably because the PC used had only four physical CPUs and CPU contention occurred among VMs.

In general, memory monitoring in DomM can be faster than that in Dom0. For DomM, one or a small number of virtual CPUs are sufficient if only one or several IDSes are running. In contrast,

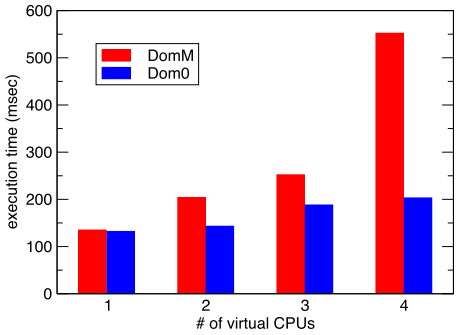


Fig. 12 The impact of various numbers of virtual CPUs on the kernel monitoring.

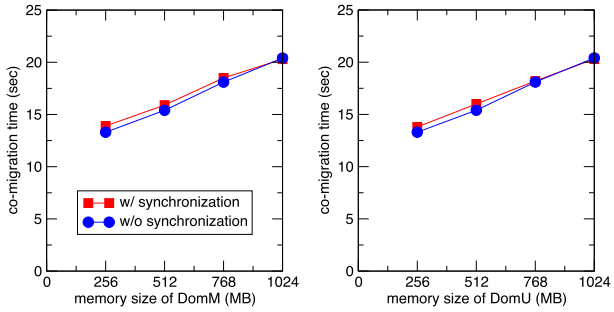


Fig. 13 The co-migration time for various combinations of the memory sizes of DomM and DomU.

Dom0 requires many virtual CPUs because it has to handle I/O requests from all the VMs.

5.2 Storage Monitoring

We executed Tripwire in DomM to scan the DomU's disk. Tripwire records the correct state of the file systems in its database and detects changes to the file systems. We confirmed that Tripwire in DomM could monitor the DomU's disk correctly as it ran inside DomU. Then, we co-migrated DomM and DomU while Tripwire in DomM was monitoring the DomU's disk. Across the migration, Tripwire could complete the integrity check of the entire disk.

Next, we measured the time needed for the integrity check by Tripwire in DomM. For comparison, we also executed Tripwire in Dom0 and measured the time for the check. On average, it took 18.9 seconds and 4.5 seconds for the integrity check in DomM and Dom0, respectively. The time in DomM was 4.2 times longer than that in Dom0. The primary reason is network virtualization. Tripwire in DomM and Dom0 had to access the disk image of DomU in the NFS server. Since the network of DomM is virtualized, its access to the NFS server is performed via Dom0.

To show the efficiency of our storage configuration, we measured the execution time of Tripwire when DomM monitored the virtual disk of DomU provided by Dom0 as illustrated in Fig. 8 (c). The execution time was 30.1 seconds and was 59% lower than that in our configuration in Fig. 9. The primary cause of this overhead is probably that network virtualization is more lightweight than storage virtualization. In the configuration of Fig. 8 (c), DomM suffers from the overhead of storage virtualization to access DomU's virtual disks mounted on Dom0. In our solution, DomM suffers from the overhead of network virtualization to access the NFS server.

5.3 Network Monitoring

We executed Snort in DomM to inspect the DomU's packets. Snort is a signature-based IDS for network traffic. Snort in DomM could capture all the packets to/from DomU. When we mounted portscans to DomU using nmap, Snort in DomM could detect the portscans. Next, we performed co-migration of DomM and DomU while Snort in DomM was monitoring the packets for DomU. Since port mirroring in Dom0 was disabled after DomU stopped and enabled before DomU restarted, Snort did not drop any packets that DomU received.

To examine the overhead of network monitoring in DomM, we measured the increase of the CPU utilization in DomM by running Snort. As a result, the CPU utilization did not increase.

5.4 Co-migration Time

We measured the time needed for synchronized co-migration of DomM and DomU when we changed the size of memory allocated to the VMs. First, we allocated 1 GB of memory to DomU and changed the memory size of DomM from 256 MB to 1 GB. For comparison, we migrated two independent DomUs in parallel without synchronization. We fixed the memory size of one DomU to 1 GB and changed the memory size of the other DomU. Except for the experiment in Section 5.7, we did not run any IDSes in DomM or any active applications in DomU. The time we measured was from when we started co-migration until the migration of both VMs completed. We measured the co-migration time 10 times.

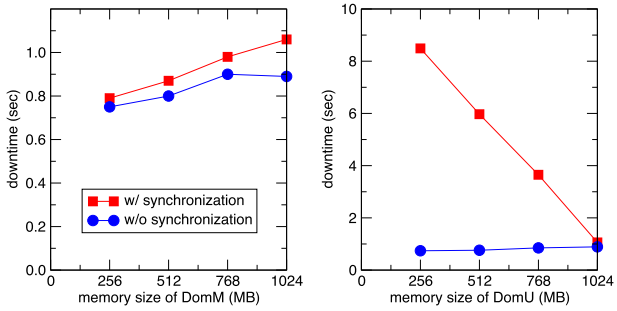
Figure 13 (a) shows the co-migration time. As the memory size of DomM became larger, the co-migration time was increasing. This is because the total migration time of two VMs depends on the total memory size to be transferred. The overhead of synchronization in co-migration was small. The difference between the co-migration times was slightly increasing as the memory size of DomM became smaller. However, even when the memory size of DomM was 256 MB, the synchronization increased the co-migration time only by 0.6 second.

Next, we fixed the memory size of DomM to 1 GB and changed that of DomU from 256 MB to 1 GB. Figure 13 (b) shows the co-migration time in this case. Similar to the above experiment, the time for co-migration with synchronization was almost the same as that without synchronization.

5.5 Downtime

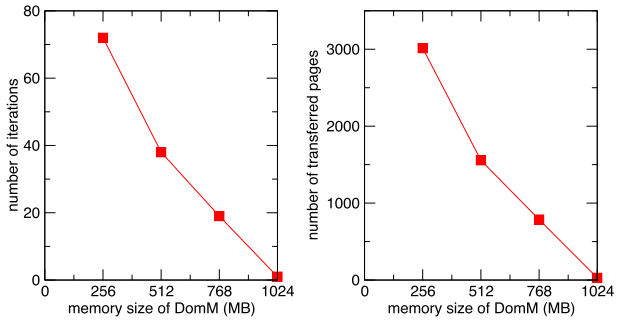
We measured the downtime of DomU during synchronized co-migration with DomM. Since various services are running in DomU, its downtime should be short. The downtime of DomU can increase at the synchronization points S_2 and S_4 in Fig. 3, at which the migration manager for DomU waits for DomM. First, we allocated 1 GB of memory to DomU and changed the memory size of DomM from 256 MB to 1 GB. The downtime we measured was the time in which DomU was not running at either source or destination host. We measured the downtime 10 times.

Figure 14 (a) shows the average downtime. As the memory



(a) DomM (b) DomU

Fig. 14 The downtime of DomU for various combinations of the memory sizes of DomM and DomU.



(a) iterations (b) dirty pages

Fig. 15 The numbers of iterations and dirty pages during synchronization.

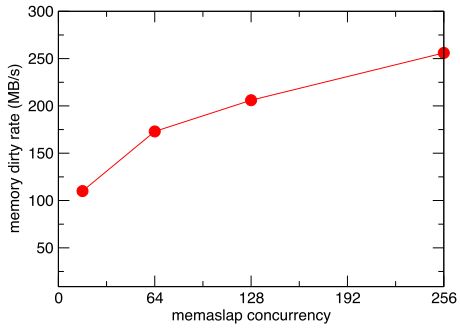
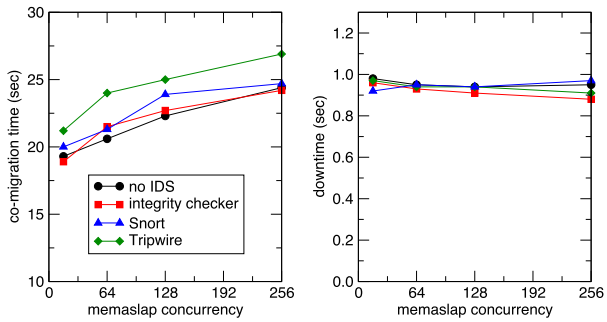


Fig. 16 The page dirty rate when using memcached and memaslap.



(a) co-migration time (b) downtime

Fig. 17 The co-migration time and downtime of DomU when using memcached and an IDS.

size of DomM became larger, the downtime was increasing gradually. However, even for DomM with 1 GB of memory, the downtime increased only by 162 ms due to synchronization. This means that synchronized co-migration does not affect the downtime largely.

Next, we measured the downtime of DomU when we fixed the memory size of DomM to 1 GB. We changed the memory size of DomU from 256 MB to 1 GB. The results are shown in Fig. 14 (b). It is shown that the downtime was dramatically increasing as the memory size of DomU became smaller. This is because our current implementation performs one-way synchronization from DomM to DomU. If we implement two-way synchronization, the downtime would be similar to that without synchronization. However, as described in Section 4.5, we assume that the memory size of DomM is usually smaller than that of DomU because DomM only monitors DomU. Therefore, the configuration of VMs in Fig. 14 (b) is a special case. Reducing the downtime in such a special case is our future work.

5.6 Extra Memory Transfer

We examined the amount of memory transferred while the migration manager for DomM waited for DomU's stop at S_1 in Fig. 3. In this experiment, we allocated the smaller amount of memory to DomM so that DomM is ready for the final stage earlier than DomU. We fixed the memory size of DomU to 1 GB and changed that of DomM from 256 MB to 1 GB. We measured the number of iterations of the migration and the number of transferred dirty pages 10 times.

Figure 15 shows the averages. The number of iterations was inversely proportional to the memory size of DomU. As the memory size of DomM became smaller, it increased dramatically. The number of transferred dirty pages was proportional to that of iterations. When the memory size of DomM was 256 MB, the total size of the transferred data increased only by about 12 MB.

5.7 Co-migration of Memcached and an IDS

To examine the performance of co-migration when we used a real application, we ran memcached 1.4.20 [29] in DomU. Memcached is a distributed memory object caching system. We allocated 512 MB to memcached. To send requests to memcached from a client machine, we used the memaslap benchmark included in libmemcached 1.0.18 [30]. It is well known that the migration performance is largely affected by the rate at which the memory pages of a VM are modified. Therefore we changed the request concurrency of memaslap from 16 to 256 to change the page dirty rate in DomU. The page dirty rate when using memcached and memaslap is shown in **Fig. 16**. It became larger as the concurrency increases.

In addition, to examine the influences of IDSes running in DomM as well, we ran three kinds of IDSes: the integrity checker in Section 5.1, Tripwire, and Snort. The integrity checker was executed every second, and Tripwire was executed repeatedly without waits. For comparison, we measured the performance when we did not run any IDS.

Figure 17 (a) shows the co-migration time. In any cases, the co-migration time became longer as the concurrency of memaslap increased. On average, the co-migration time was long in the order of no IDS, the integrity checker, Snort, and Tripwire. The reason why the co-migration time is longer for Tripwire is that Tripwire frequently modified the page cache in memory by reading new files. Figure 17 (b) shows the downtime of DomU. The downtime was not largely affected by the concurrency of

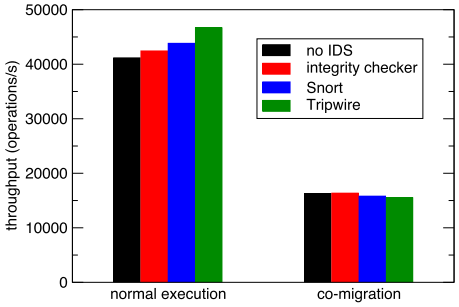


Fig. 18 The throughput of memcached in DomU.

memaslap or the difference of an IDS running in DomM. As the concurrency was increasing, the downtime slightly increased only for Snort, while it slightly decreased for the others.

Next, we measured the performance of memcached in DomU during normal execution and during co-migration. To examine the influences of IDSes in DomM, we also ran one of the three IDSes or no IDS. **Figure 18** shows the throughput of memcached when we used memaslap. During normal execution, the throughput when an IDS ran in DomM was higher than that when no IDS ran. To investigate the cause of this improvement, we ran a simple infinite loop in DomM. Since the throughput was also higher in this case, this improvement was probably due to virtual CPU scheduling. During co-migration, the throughput was reduced largely in any cases, but it was almost not affected by the difference of a running IDS.

6. Conclusion

In this paper, we proposed VMCoupler, which enables synchronized co-migration of offloaded IDSes and their target VM. Offloaded IDSes are run in a guard VM and monitor its target VM using VM introspection. VMCoupler synchronizes the migration processes of a guard VM and a target VM so that a guard VM can always monitor a running target VM. Our experiments showed that the overhead of monitoring and co-migration was small and that the downtime of a target VM was short.

Our future work is decreasing the downtime of a target VM due to the synchronized co-migration. We need to implement two-way synchronization so that a target VM also waits for a guard VM to be ready for the final stage of the migration. Another direction is extending VMCoupler to support various combinations of guard VMs and target VMs as a group. Currently, one guard VM is necessary for one target VM, but one guard VM could monitor multiple target VMs. In this case, VMCoupler has to migrate more than two VMs simultaneously, so that it could impact the migration performance and the downtime of target VMs. Also, we are planning to extend VMCoupler to domains other than IDS offloading, such as out-of-band remote VM management [31].

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

References

[1] Garfinkel, T. and Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection, *Proc. Network and Distributed Systems Security Symp.*, pp.191–206 (2003).
[2] Joshi, A., King, S., Dunlap, G. and Chen, P.: Detecting Past and Present Intrusions through Vulnerability-specific Predicates, *Proc. Symp. Operating Systems Principles*, pp.91–104 (2005).
[3] Jiang, X., Wang, X. and Xu, D.: Stealthy Malware Detection through VMM-based "Out-of-the-box" Semantic View Reconstruction, *Proc. Conf. Computer and Communications Security*, pp.128–138 (2007).
[4] Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J. and Lee, W.: Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection, *Proc. Symp. Security and Privacy*, pp.297–312 (2011).
[5] Fu, Y. and Lin, Z.: Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection, *Proc. Symp. Security and Privacy*, pp.586–600 (2012).
[6] Kourai, K. and Utsunomiya, H.: Synchronized Co-migration of Virtual Machines for IDS Offloading in Clouds, *Proc. Int. Conf. Cloud Computing Technology and Science*, pp.120–129 (2013).
[7] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and the Art of Virtualization, *Proc. Symp. Operating Systems Principles*, pp.164–177 (2003).
[8] Kim, G. and Spafford, E.: The Design and Implementation of Tripwire: A File System Integrity Checker, *Proc. Conf. Computer and Communications Security*, pp.18–29 (1994).
[9] Roesch, M.: Snort – Lightweight Intrusion Detection for Networks, *Proc. USENIX System Administration Conf.* (1999).
[10] Clark, C., Fraser, K., Hand, S., Hansen, J.G., Jul, E., Limpach, C., Pratt, I. and Warfield, A.: Live Migration of Virtual Machines, *Proc. Symp. Networked Systems Design and Implementation*, pp.273–286 (2005).
[11] Fraser, K., Hand, S., Neugebauer, R., Pratt, I., Warfield, A. and Williamson, M.: Safe Hardware Access with the Xen Virtual Machine Monitor, *Proc. Workshop on Operating System and Architectural Support for the on demand IT InfraStructure* (2004).
[12] Nakajima, J. and Stekloff, D.: Improving HVM Domain Isolation and Performance, *Xen Summit September 2006* (2006).
[13] Thibault, S.: Stub Domains, *Xen Summit Boston 2008* (2008).
[14] Murray, D.G., Milos, G. and Hand, S.: Improving Xen Security through Disaggregation, *Proc. Int. Conf. Virtual Execution Environments*, pp.151–160 (2008).
[15] Colp, P., Nanavati, M., Zhu, J., Aiello, W., Coker, G., Deegan, T., Loscocco, P. and Warfield, A.: Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor, *Proc. Symp. Operating Systems Principles*, pp.189–202 (2011).
[16] Butt, S., Lagar-Cavilla, H.A., Srivastava, A. and Ganapathy, V.: Self-service Cloud Computing, *Proc. Conf. Computer and Communications Security*, pp.253–264 (2012).
[17] Milojíčić, D.S., Douglis, F., Paindaveine, Y., Wheeler, R. and Zhou, S.: Process Migration, *ACM Comput. Surv.*, Vol.32, No.3, pp.241–299 (2000).
[18] Plank, J.S., Beck, M., Kingsley, G. and Li, K.: Libckpt: Transparent Checkpointing under Unix, *Proc. USENIX Winter 1995 Technical Conf.*, pp.213–223 (1995).
[19] Duell, J., Hargrove, P. and Roman, E.: The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart, Technical report, LBNL (2002).
[20] Mullender, S.J., v. Rossum, G., Tanenbaum, A.S., v. Renesse, R. and v. Staveren, H.: Amoeba – A Distributed Operating System for the 1990s, *IEEE Comput.*, Vol.23, No.5, pp.44–53 (1990).
[21] Barak, A. and Wheeler, R.: MOSIX: An Integrated Multiprocessor UNIX, *Proc. USENIX Winter 1989 Technical Conf.*, pp.101–112 (1989).
[22] Schmidt, B.K.: Supporting Ubiquitous Computing with Stateless Consoles and Computation Caches, PhD Thesis, Stanford University (2000).
[23] Osman, S., Subhraveti, D., Su, G. and Nieh, J.: The Design and Implementation of Zap: A System for Migrating Computing Environments, *Proc. Symp. Operating Systems Design and Implementation*, pp.361–367 (2002).
[24] Deshpande, U., Wang, X. and Gopalan, K.: Live Gang Migration of Virtual Machines, *Proc. Int. Symp. High Performance Distributed Computing*, pp.135–146 (2011).
[25] Trend Micro, Inc.: Deep Scrurity, available from ⟨http://www.trendmicro.com/us/enterprise/cloud-solutions/deep-security/⟩.
[26] VMware, Inc.: VMware vShield Endpoint: Virtualization Security, available from ⟨http://www.vmware.com/products/vsphere/features-endpoint⟩.
[27] Hubert, B.: Linux Advanced Routing & Traffic Control, available from ⟨http://www.lartc.org/⟩.
[28] Jansen, G.: Network security monitoring with KVM, available from ⟨http://geertj.blogspot.jp/2010/12/network-security-monitoring-with-kvm.html⟩.
[29] Fitzpatrick, B.: Distributed Caching with Memcached, *Linux J.*,

Vol.2004, No.124, p.5 (2004).
[30] Aker, B.: libMemcached, available from ⟨http://libmemcached.org/⟩.
[31] Egawa, T., Nishimura, N. and Kourai, K.: Dependable and Secure Remote Management in IaaS Clouds, *Proc. Int. Conf. Cloud Computing Technology and Science*, pp.411–418 (2012).

Kenichi Kourai is an associate professor in the Department of Creative Informatics at Kyushu Institute of Technology. He received his Ph.D. degree from the University of Tokyo in 2002. He has been working on operating systems. His current research interest is in dependable and secure systems using virtual machines.

Hisato Utsunomiya received his B.Sc. and M.Sc. degrees from Kyushu Institute of Technology in 2011 and 2013, respectively. His current research interest is in cloud computing and virtual machines.