VM Migration Support for Secure Out-of-band VNC with Shadow Devices

Tomoya Unoki
Kyushu Institute of Technology
unoki@ksl.ci.kyutech.ac.jp

Kenichi Kourai
Kyushu Institute of Technology
kourai@csn.kyutech.ac.jp

***Abstract*—Out-of-band VNC (OOB-VNC) enables users to remotely manage the systems in virtual machines (VMs) without relying on the target systems. It directly accesses VMs' virtual devices, but existing clouds do not protect them sufficiently. Therefore, secure OOB-VNC has been proposed. It runs *shadow devices* outside the virtualized system and securely accesses the shadow devices. However, users cannot use this management method with VM migration because traditional VM migration cannot deal with shadow devices. This paper proposes *SDmigrate*, which continues secure OOB-VNC after VMs are migrated. SDmigrate enables saving and restoring the states of shadow devices transparently via *fake devices*. A fake device communicates with the corresponding shadow device. In addition to the internal states of a shadow keyboard, mouse, and video card, SDmigrate deals with the external state of a shadow video card, i.e., the video memory in a migrated VM. We have implemented SDmigrate in Xen and confirmed that the degradation of migration performance was negligible.**

***Index Terms*—out-of-band VNC, VM migration, shadow device, fake device**

1. Introduction

Users can run their systems using virtual machines (VMs) in Infrastructure-as-a-Service (IaaS) clouds. To manage those systems, they often run VNC servers [1] inside the VMs to access the systems in VMs remotely. Since this method relies heavily on the target systems in VMs, clouds provide the other management method called *out-of-band VNC (OOB-VNC)*. Using this method, users can access the systems in VMs indirectly via their virtual devices, e.g., virtual keyboards, mice, and video cards, running outside VMs. As a result, they can manage their systems even at boot time and on system failures.

On the other hand, existing clouds do not protect virtual devices sufficiently because untrusted cloud operators may exist [2]–[6]. To protect inputs and outputs in OOB-VNC, VSBypass [7] has been proposed. It runs the virtualized system in a VM using *nested virtualization* [8]. When it intercepts access to virtual devices, it forwards the access to *shadow devices*. Shadow devices handle all the inputs and outputs securely outside the virtualized system. Using shadow devices, VSBypass can provide *secure OOB-VNC*, which prevents information leaks to cloud operators. However, after clouds migrate their VMs to other hosts, users cannot continue to use secure OOB-VNC. The migration utility running inside the virtualized system cannot deal with necessary states because shadow devices run outside it.

In this paper, we propose *SDmigrate*, which enables users to continue secure OOB-VNC after clouds migrate VMs. In SDmigrate, the migration utility can deal with not only the normal states of a VM but also the states of shadow devices. For this purpose, SDmigrate provides a *fake device* for each shadow device in a virtualized system. Using a fake device, the state of a shadow device is saved and restored as that of a fake device transparently. The migration utility does not need to be modified to support shadow devices. A fake device and the corresponding shadow device pass the state using shared memory securely and efficiently. The states of shadow devices are encrypted by the shadow devices themselves to prevent information leaks.

To migrate VMs managed by secure OOB-VNC, SDmigrate saves and restores the internal states of a shadow keyboard, mouse, and video card via the corresponding fake devices. In addition, it saves and restores the *external* state of a shadow video card, i.e., the video memory (VRAM) allocated in the memory of a VM. It enables a shadow video card to identify and share the VRAM in a VM at the migration time. We have extended VSBypass to implement SDmigrate using Xen 4.8 [9]. In SDmigrate, Xen and KVM [10] are supported as a virtualized system, which runs users' VMs. Using SDmigrate, we confirmed that we could continue secure OOB-VNC after migrating a VM. In addition, we showed that the degradation of migration performance was negligible.

This paper is organized as follows. Section 2 discusses the issue of VSBypass after VMs are migrated. Section 3 proposes SDmigrate, and Section 4 provides the implementation details. Section 5 shows the degradation of migration performance in SDmigrate. Section 6 compares SDmigrate with previous work, and Section 7 summarizes the paper.

2. Secure OOB-VNC

2.1. VSBypass

OOB-VNC is often used as a remote management method for VMs in clouds. VNC clients access virtual

979-8-3503-0460-2/23/$31.00 ©2023 IEEE

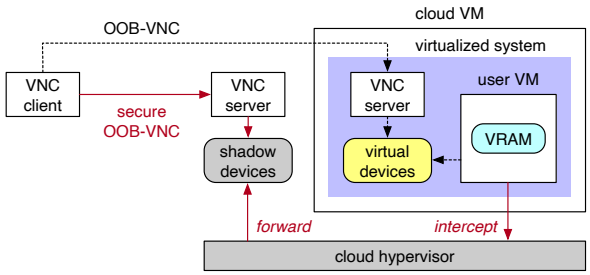


Figure 1. Secure OOB-VNC provided by VSBypass.

devices running outside VMs and manage the systems in the VMs, as illustrated in Fig. 1. They connect to VNC servers provided by clouds and access virtual keyboards, mice, and video cards to provide the GUI system management of the VMs to users. However, cloud operators can easily eavesdrop on the inputs and outputs of OOB-VNC via virtual devices and VNC servers because existing clouds do not protect virtual devices or VNC servers sufficiently. Cloud operators are responsible for the daily management of the entire virtualized system. Unfortunately, all of them are not always trusted [2]–[6]. It is reported that insider attacks occupy 28% of cybercrimes [12] and that sensitive information has been accessed by 35% of system administrators without authorization [11].

For such clouds, VSBypass [7] achieves *secure* OOB-VNC by providing *shadow devices*. Shadow devices run outside the virtualized system, which consists of VMs and their virtual devices. Instead of virtual devices, VSBypass uses shadow devices for OOB-VNC. VNC clients can access VMs via shadow devices as usual. Nevertheless, information leaks are prevented because shadow devices outside the virtualized system cannot be accessed by cloud operators. In VSBypass, VMs can perform I/O to shadow devices outside the virtualized system by using *transparent passthrough*. This mechanism forwards I/O access to the corresponding shadow devices transparently by intercepting the access that VMs issue to virtual devices.

VSBypass trusts only the outside of the virtualized system and does not trust the entire virtualized system. To satisfy this assumption, it uses a technique called *nested virtualization* [8]. The virtualized system runs in an external VM, which is called the *cloud VM*. Users' VMs, named *user VMs*, and virtual devices used by them run in the cloud VM. Outside the cloud VM, VSBypass provides a shadow device for each virtual device. The hypervisor, called the *cloud hypervisor*, runs the cloud VM and shadow devices. Although nested virtualization imposes extra overhead, it is shown that the overhead was 6–8% in some workloads [8]. Also, various mechanisms and hardware extensions have been proposed to reduce the overhead [13]–[17].

When an I/O instruction is executed in a user VM to a virtual device, it is intercepted by the cloud hypervisor. It is emulated by using the shadow device that corresponds to the virtual device. After emulating the instructions for inputs, the cloud hypervisor returns an input value, e.g., a keyboard or mouse input, to the user VM. The input value is received from a VNC client. When emulating the instructions for outputs, the cloud hypervisor writes the specified output value, e.g., a video output, to the shadow device. The output value is sent to a VNC client finally. To handle direct access to the video memory (VRAM) allocated in the memory of a user VM, a shadow video card shares the VRAM and detects writes to the VRAM by the VM using the function of the cloud hypervisor. Then, it sends only the graphics data of the modified regions to a VNC client.

2.2. VM Migration with Secure OOB-VNC

The migration utility running inside the virtualized system can migrate a user VM from a source host to a destination host. First, the migration utility in the source host transfers the memory data of a VM to that in the destination host. Since the memory can be updated by the execution of the VM itself during VM migration, the migration utility re-transfers updated memory data. When the amount of memory data to be re-transferred is sufficiently small, the migration utility suspends the VM and transfers the remaining memory data. In addition, it transfers the states of virtual CPUs and devices. At the destination host, the migration utility restores the VM from the received states and finally resumes the VM.

However, secure OOB-VNC cannot be continued after the VM is migrated. This is because the states of shadow devices are not transferred. Shadow devices are located outside the virtualized system, but the migration utility runs inside it. Therefore, the migration utility is not aware of shadow devices. It neither saves nor restores the states of shadow devices when a VM is migrated. This results in inconsistent states of shadow devices at the destination host. The migrated VM or a VNC client cannot correctly use shadow devices. In addition, a shadow video card has an external state, i.e., the VRAM, inside a user VM. This state is transferred with the memory of the user VM, but a shadow video card at the destination host cannot identify the location of the VRAM at the resume time.

3. SDmigrate

3.1. Threat Model

Our threat model is the same as that in VSBypass. We do not trust cloud operators, who perform daily management of the virtualized system. Cloud operators can fully control the virtualized system. However, we trust cloud providers, who maintain the trusted computing base (TCB). The TCB consists of shadow devices as well as hardware and the cloud hypervisor outside the virtualized system. Only trusted system administrators maintain the TCB. We assume that they do not attack the TCB. In fact, many systems adopt such an administrative hierarchy of trusted administrators and untrusted operators. For instance, two types of privileges are provided in Oracle Database [18]. Eight types of administrators exist in IBM Domino [19] to restrict privileges.

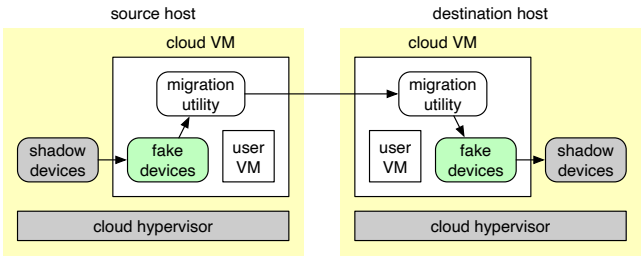


Figure 2. Transferring the states of shadow devices in VM migration.

3.2. VM Migration with Shadow Devices

SDmigrate enables continuing secure OOB-VNC using shadow devices after clouds migrate VMs. Unlike VSBypass, the migration utility can handle the states of shadow devices, which are located outside the virtualized system. When VM migration is started, SDmigrate first creates new shadow devices with initial states at the destination host. Then, the migration utility running in the source host saves the states of shadow devices in addition to the other states of a user VM. After that, it transfers the saved states to the destination host. When the transferred states of the shadow devices are received by the migration utility, the states of the new shadow devices are restored using them. Finally, SDmigrate stops the shadow devices at the source host.

It seems to be natural that the migration utility directly saves and restores the states of shadow devices. However, this design is not acceptable because the migration utility needs to be modified. Since there are various migration utilities, e.g., command-line and GUI tools, it is desirable to use existing migration utilities as they are. Therefore, SDmigrate introduces special devices called *fake devices* and additionally runs them inside the virtualized system. It uses fake devices to save and restore the states of shadow devices transparently. It is not necessary to modify the migration utility. Note that fake devices need to be developed for each virtualized system.

Fig. 2 shows the overview of transferring the states of shadow devices in SDmigrate. Like a normal virtual device, the migration utility running in the source host tries to save the state of a fake device. At this time, the fake device sends a request to the corresponding shadow device and obtains the state of the shadow device. That state is saved by the migration utility as that of the fake device. Similarly, the migration utility running in the destination host tries to restore the state of a fake device like a virtual device. At this time, the fake device sends a request to the corresponding shadow device and restores the state of the shadow device.

It is not easy for fake devices to communicate with shadow ones. Traditionally, the migration utility uses interprocess communication when it saves and restores the states of virtual devices. This is possible because virtual devices and the migration utility run on the same operating system (OS). However, we cannot use this communication method between fake and shadow devices. Completely different OSes run these devices. For these devices, a possible communication method is to use a virtual network, but shadow devices need to permit access from the network. This allows access from not only fake devices but also untrusted cloud operators in the virtualized system. This means that the attack surface to shadow devices increases. For example, shadow devices can suffer from the buffer overflow attack. For efficiency, the virtual network imposes communication overhead. It can take longer to save and restore the states of shadow devices.

For efficient and secure communication between shadow devices and fake devices, SDmigrate shares the memory of fake devices with shadow devices. To establish such shared memory without relying on the virtualized system in the cloud VM, a fake device directly invokes the cloud hypervisor. Then, the corresponding shadow device is invoked by the cloud hypervisor. Next, the shadow device and the fake device share a buffer via the cloud hypervisor. Since the shadow device just accesses the memory of the fake device, it can avoid active attacks from untrusted cloud operators. To prevent information leaks via the states of shadow devices, each shadow device encrypts its state and writes it to the shared memory.

In addition to restoring the states of shadow devices, SDmigrate needs to share the VRAM in the migrated user VM with a new shadow video card again at the destination host. However, this is not easy because the VRAM is identified by trapping memory-mapped I/O executed at the boot time of a user VM. This memory-mapped I/O is not executed at the resume time. To identify the VRAM without memory-mapped I/O, SDmigrate saves and transfers the address and size of the VRAM as an additional state of a shadow video card. At the destination host, SDmigrate has to translate the address of the VRAM in the user VM into that in the cloud VM using the extended page tables (EPT) of the user VM. To obtain the EPT without memory-mapped I/O, SDmigrate traps the configuration of the EPT and records the address of the EPT in advance.

4. Implementation

We have extended VSBypass to implement SDmigrate. As illustrated in Fig. 3, SDmigrate uses the hypervisor of Xen 4.8.0 [9] as the cloud hypervisor. SDmigrate uses Dom0 in Xen as the cloud control VM and runs shadow devices in it. SDmigrate uses DomU in Xen as the cloud VM and runs an existing virtualized system in it. In the current implementation, SDmigrate supports Xen and KVM [10] as virtualized systems. For Xen, the virtualized system consists of the guest hypervisor, the guest control VM, and user VMs. The migration utility, virtual devices, and fake devices run in the guest control VM. For KVM, the virtualized system consists of the guest hypervisor running inside the Linux kernel and user VMs. The migration utility and fake devices run on top of the kernel. In addition, SDmigrate runs a proxy VM as DomU when a user VM is created.

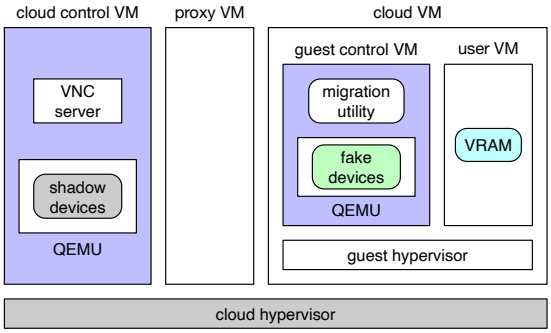


Figure 3. The system overview of SDmigrate using Xen as a virtualized system in the cloud VM.

4.1. Shadow Devices

Shadow devices are created as virtual devices of proxy VMs. Like normal virtual devices, we have implemented shadow devices in QEMU running in the cloud control VM. We assign only the minimum amount of resources to proxy VMs and use them only for providing shadow devices. When a user VM executes an I/O instruction, that instruction causes a VM exit to the cloud hypervisor. Then, the cloud hypervisor checks the port number specified for I/O-mapped I/O or the memory address specified for memory-mapped I/O. If that is a target of transparent passthrough, the cloud hypervisor sends an I/O request to the corresponding shadow device. When the request handling is completed, the cloud hypervisor executes a VM entry to the user VM.

To handle direct access to the VRAM of a user VM, a shadow video card shares the VRAM and uses the log-dirty mechanism provided by the cloud hypervisor. This mechanism is originally used for the guest hypervisor to detect such direct access. Similarly, a shadow video card registers the VRAM region in a user VM to the log-dirty mechanism, which protects the registered memory region in a read-only manner. If a user VM writes data to the region, that write causes a VM exit to the cloud hypervisor and that access is recorded in it. Then, a shadow video card obtains the recorded information and identifies updated memory regions.

4.2. Fake Devices

Fake devices are created as virtual devices of a user VM. Like normal virtual devices, we have implemented fake devices in QEMU running in the cloud VM. A fake device just forwards requests for saving and restoring the state to the shadow device corresponding to it. When the state of a shadow device is saved, a fake device prepares a buffer and shares it with the corresponding shadow device. When the fake device sends a request to the shadow device, the shadow device collects its own state and stores it in the shared buffer. That state is received by the fake device using the buffer. Then, the migration utility obtains that state via the fake device. Next, it transfers the obtained state to the migration utility running in the destination host. Using the received state, the migration utility restores the state of the fake device. The fake device does not change its state but stores the restored state in the shared buffer. Finally, the shadow device restores its state after it receives the state from the shared buffer.

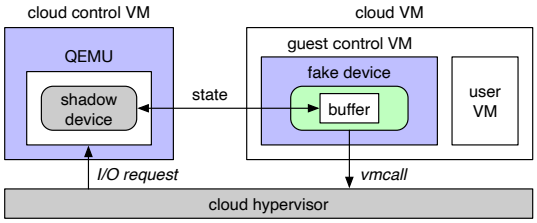


Figure 4. Communication between fake and shadow devices.

4.3. Interaction between Shadow and Fake Devices

When the state of a shadow device is saved or restored, the cloud hypervisor is invoked by a fake device using the vmcall instruction, as illustrated in Fig. 4. When this instruction is executed in the guest control VM, it causes a VM exit. For this instruction, a fake device stores two parameters in CPU registers. One is the virtual address of a buffer used to store the state of a shadow device, and the other is the size of the buffer. A fake device uses the mmap system call to allocate an anonymous memory page to the buffer. It also uses the mlock system call to pin the page so that the buffer is paged out to swap space.

Upon the VM exit, the cloud hypervisor performs the address translation to enable a shadow device to access the buffer of the fake device. The fake device uses a virtual address, but a shadow device requires the corresponding physical address. The cloud hypervisor first accesses the virtual CPU of the cloud VM and reads the CR3 register. This register contains the address of the page tables. Then, the cloud hypervisor traverses the page tables. As a result, the virtual address of the buffer is translated into the physical one.

Next, the cloud hypervisor invokes the shadow device corresponding to the fake device. To achieve this, a special I/O request is sent to QEMU running in the cloud control VM. For the requests to shadow devices, QEMU uses dedicated I/O port numbers to distinguish them from normal requests. The port number is also used to determine the type of shadow device. Each I/O request consists of a request type, the physical address of the buffer, and the buffer size in addition to a port number. The request type is either save or restore. When the request is completed by the shadow device, the cloud hypervisor returns the size of the saved or restored state to the fake device.

4.4. Save/Restore of Internal States

For saving and restoring the state of a shadow device, SDmigrate leverages the interface provided by QEMU. Tra-

TABLE 1. THE STATES OF SHADOW DEVICES.

device type	state
keyboard	4 registers, 4 internal states, and a queue (1 register and 1 internal state when necessary)
mouse	1 register, 10 internal states, and a queue
video card	324 registers, 9 internal states, a color palette, PCI configuration registers, and PCI IRQ states

ditionally, VM migration uses this interface to transfer the state of a VM to another host. VM checkpointing also uses it to save the state to a disk. Therefore, the traditional QEMU can handle the state of a VM only for network sockets or files. In SDmigrate, we have added a new interface to QEMU. Using this interface, our QEMU can handle the state of a VM for memory. When the state of a shadow device is saved, QEMU stores the state in a buffer of a fake device. When the state is restored, QEMU reads the state from the buffer.

For secure OOB-VNC, SDmigrate saves and restores the states of a shadow keyboard, mouse, and video card. Table 1 shows the list of the saved and restored states. When the states of shadow devices are saved, each shadow device reads the device registers and collects the other internal states. Then, it encrypts the collected states using the AES algorithm and writes the encrypted states to the buffer of a fake device. When the states of shadow devices are restored, each shadow device sets the saved state to its own device registers and the other internal states. At this time, the saved state is decrypted using AES.

4.5. Save/Restore of External States

To emulate the virtual video card of a user VM, a shadow video card shares the memory region used for the VRAM of that VM, as illustrated in Fig. 5. Since a shadow video card is created as a virtual video card of a proxy VM, it usually accesses the VRAM in the proxy VM. Therefore, it unmaps that VRAM and re-maps the VRAM in a user VM. To re-map that VRAM, SDmigrate needs to first identify its physical address used in the user VM and then translate the address into the one used in the cloud VM. Since it is difficult to identify the address of the VRAM at the resume time, SDmigrate embeds this external state into the internal state of a shadow video card and transfers it to the shadow video card at the destination host. For the address translation, SDmigrate needs the EPT of a user VM, which is created by the guest hypervisor.

To find the EPT of a user VM at the resume time, the cloud hypervisor traps a write of an EPT pointer to the virtual machine control structure (VMCS), which is performed by the guest hypervisor. VMCS is a data structure used by Intel VT-x. This write is at least done at the boot and resume times of a user VM. Since a VM exit does not occur in this event by default, the cloud hypervisor configures the VM-exit control fields in the VMCS. Upon this VM exit, the cloud hypervisor records the address of the EPT. After that, a shadow video card issues a newly added hypercall to

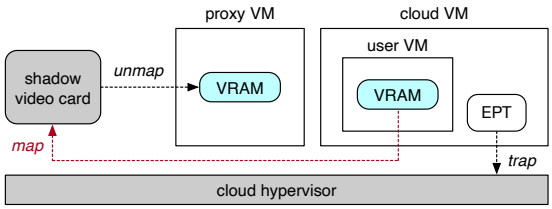


Figure 5. Re-sharing the VRAM of a user VM.

the cloud hypervisor and obtains information on the pages used for the VRAM in the user VM. At this time, the cloud hypervisor translates the guest physical address of each page in the user VM into the host one using the recorded EPT. Finally, it maps these pages by invoking a hypercall to the cloud hypervisor and shares the VRAM with the user VM.

For KVM, this mechanism alone does not work well because the EPT is expanded and shrunk on demand. Until the VRAM is accessed by a user VM, the EPT does not have entries for the pages used for the VRAM. This means that the cloud hypervisor cannot always translate the physical address of the VRAM. To address this issue, we have slightly modified the QEMU and the guest hypervisor inside the virtualized system. The modified QEMU issues a new hypercall to the guest hypervisor and pre-allocates all the pages used for the VRAM by causing page faults. Then, the guest hypervisor immediately invokes the cloud hypervisor and translates the physical address of the VRAM before the EPT is shrunk. Since this modification is only for memory pre-allocation, no new security issues arise.

5. Experiments

We conducted experiments to show the usefulness of SDmigrate. In addition to SDmigrate, we used the traditional system with single virtualization, that with nested virtualization, and VSBypass. We created a cloud VM with two virtual CPUs and 3 GB of memory and a proxy VM with two virtual CPUs and 1 GB of memory. We used Xen 4.4.0 and KVM with QEMU 2.4.1 as virtualized systems running in the cloud VM. In the cloud VM, we created a user VM with two virtual CPUs. We changed the memory size of the user VM between 256 MB and 2 GB. To run these VMs, we used two identical PCs with an Intel Xeon E3-1226 v3 processor, 8 GB of memory, and Gigabit Ethernet.

5.1. Secure OOB-VNC after VM Migration

To show that SDmigrate can continue secure OOB-VNC after a VM is migrated, we performed VM migration while we used secure OOB-VNC. First, we connected to the VNC server running in the cloud control VM using a VNC client. Then, we performed login to the user VM via the VNC client. When we used secure OOB-VNC, transparent passthrough redirected the access to the user VM. Next, we performed the migration of the user VM. When the virtualized system was Xen, we migrated a VM using the

TABLE 2. THE SAVED SIZE OF THE STATES OF SHADOW DEVICES.

shadow device	size (bytes)
shadow keyboard	288
shadow mouse	304
shadow video card	1408

xl command with the migrate subcommand in the guest control VM. When KVM was used, we migrated a VM with the virsh command with the migrate subcommand. After the VM was migrated, we connected the VNC client to the VNC server running in the cloud control VM again at the destination host. In VSBypass, we could not continue access to the user VM using secure OOB-VNC. In SDmigrate, in contrast, access to the user VM was continued.

5.2. Performance of Saving and Restoring States

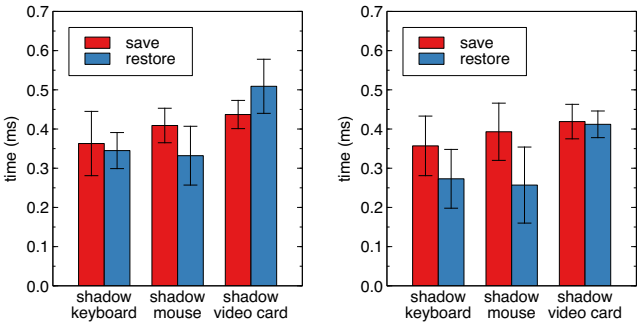
We saved and restored the states of a shadow keyboard, mouse, and video card and investigated the performance. In this experiment, we directly saved and restored the states of these shadow devices without using fake devices. First, we measured the size of the saved states of the shadow devices. As shown in Table 2, the sizes were about 300 bytes for a shadow keyboard and mouse, whereas the size was 1.4 KB for a shadow video card. Even the size of these states was 2.0 KB in total. This was negligible compared with the memory size of a user VM. These sizes were the same regardless of the virtualized system running in the cloud VM because shadow devices are virtual devices provided for proxy VMs.

Next, we measured the time needed for saving and restoring the states. We saved and restored the states of the shadow devices 10 times and obtained the average time with the standard deviation. Fig. 6(a) shows the measured time when we used Xen as a virtualized system. The time basically depended on the size of the state but was not proportional to that. For example, the size of the state of a shadow video card was much larger than the others, but the save time just increased slightly. This means that the fixed processing time was long for saving and restoring the state of a shadow device. That includes the invocation of the cloud hypervisor and the communication between the cloud hypervisor and QEMU in the cloud control VM. The total save and restore times of these three shadow devices were 1.2 ms and 1.3 ms, respectively.

Fig. 6(b) shows the time when we used KVM as a virtualized system. The save time of each shadow device was almost the same as when we used Xen. However, the restore time was 81 μs shorter on average. In total, the restore time was reduced to 0.9 ms. This is probably caused by the difference in the scheduling algorithm between Xen and KVM.

5.3. Performance of VM Migration

We investigated the performance of migrating a user VM. SDmigrate ran three shadow devices for secure OOB-VNC. It transferred their states to the destination host. For VSBypass, the states of these shadow devices were not transferred. We changed the amount of memory assigned to a user VM between 256 MB and 2 GB. We measured the migration time, which is the time taken to execute the migration command at the source host. In addition, we measured the downtime, which is from stopping a user VM at the source host to resuming it at the destination host. For each memory size, we migrated a user VM 5 times.



(a) Xen (b) KVM

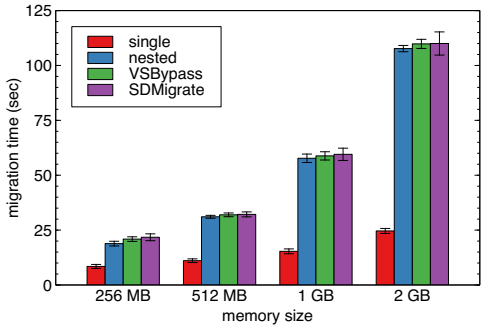
Figure 6. The save/restore time of the states of shadow devices.

Fig. 7 shows the average and the standard deviation of the migration time when Xen and KVM were used as virtualized systems, respectively. The migration time in SDmigrate was increased only by 482 ms in Xen and 29 ms in KVM, compared with VSBypass. The larger the memory size was, the smaller this impact on the migration time was because the migration time depended on the memory size of a VM. The overhead was 4.0-5.7% even when the memory size was 256 MB. For 2 GB of memory, that was negligible. Compared with the traditional system with nested virtualization, the average increase in migration time was 2.0 seconds in Xen and 140 ms in KVM. This included the overhead of VSBypass because SDmigrate was implemented on top of VSBypass. The overhead of nested virtualization caused the difference between Xen and KVM. This overhead is shown in the performance difference between the traditional systems with single and nested virtualization. The overhead in KVM was much smaller than in Xen.

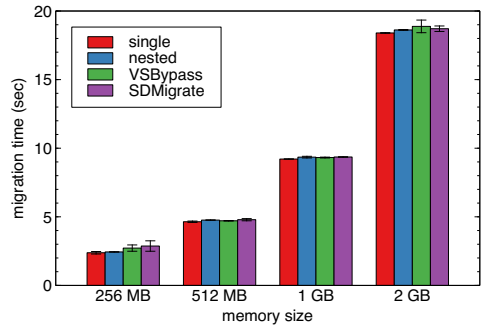
Fig. 8(a) shows the average and standard deviation of the downtime when Xen and KVM were used as virtualized systems, respectively. The downtime in SDmigrate was 73 ms longer than that in VSBypass when we used KVM. This is probably because of handling the states of shadow devices in SDmigrate. In contrast, the downtime was 62 ms shorter on average when we used Xen. However, it was 63 ms longer on average than that in nested virtualization. These reasons are under investigation, but the performance degradation was small.

6. Related Work

A shadow device in SDmigrate is similar to a passthrough device of VMs. A passthrough device is a

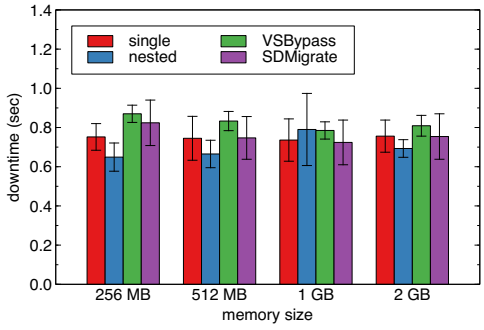


(a) Xen

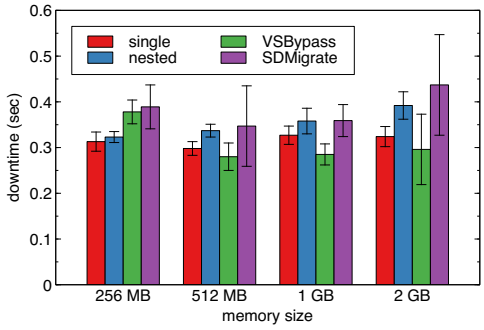


(b) KVM

Figure 7. The migration time in SDmigrate.



(a) Xen



(b) KVM

Figure 8. The downtime in SDmigrate.

virtual device used to improve I/O performance by directly accessing hardware. In contrast, a shadow device directly accesses virtual hardware, which is provided to the virtualized system by using nested virtualization. However, migrating VMs that use passthrough devices is difficult because passthrough devices strongly depend on physical hardware. For passthrough NICs, several systems have been proposed to migrate VMs.

The bonding drivers inside VMs are useful for migrating VMs that use passthrough NICs [20]. In this migration method, a passthrough NIC is bonded to a paravirtual NIC using the Linux bonding driver. Before a VM is migrated, it hot-unplugs the passthrough NIC and thereby the bonding driver automatically uses the paravirtual NIC. Since the VM uses only the paravirtual NIC at this time, it can be easily migrated. After the VM is migrated, the passthrough NIC is hot-plugged, so that the bonding driver uses it again. This can be easily applied only by a configuration change in a VM, but the network downtime occurs on hot-unplugging.

Instead of the bonding driver, the network plugin architecture (NPA) [21] is also used inside VMs to migrate VMs that use passthrough NICs. It enables network drivers to add and remove plugins. Before a VM is migrated, the plugin of a passthrough NIC is removed and that of a paravirtual NIC is added. After the VM is migrated, the plugin of a passthrough NIC is used again. This switching time is shorter than that in the bonding driver, but it is necessary to modify network drivers largely. In addition, NPA supports only SR-IOV NICs.

The transparent migration of VMs that use passthrough NICs is also achieved by using shadow drivers [22]. A shadow driver is used to record access to a network driver. During VM migration, it processes requests to the network driver, instead of a real driver. At the destination host, it redirects the recorded requests to a real driver after a new network driver starts. However, using a shadow driver needs large modifications to the hypervisor and the guest OS inside a VM.

CompSC [23] enables the device registers of passthrough NICs to be restored completely after a VM is migrated. This is difficult in general because there are registers whose values are erased after reads, that are read-only, and that result in some actions to NICs after writes. CompSC solves this problem by recording access to passthrough NICs in the network driver at the source host and replaying it at the destination host. In addition, it performs emulation in the hypervisor and also restores statistical information. In SDmigrate, the states of shadow devices can be completely saved and restored because shadow devices are virtual devices.

SRVM [24] restores the minimal states that are needed to run passthrough NICs, instead of restoring the complete states. It first finds memory regions where packets are stored by NICs. Then, it obtains these packets from the found regions and forwards them to the destination host. It is not necessary to modify the guest OS in a VM because it is implemented only in the hypervisor. However, it requires SR-IOV NICs in the passthrough mode.

USShadow [25] supports VM migration for a secure virtual serial console, which is used to manage a VM by accessing its serial console via a shadow serial device. It is similar to SDmigrate, but a shadow serial device is very simple and has a state of only 16 bytes. In contrast, SDmigrate deals with the states of more complex shadow devices, whose size is 2 KB in total. For a shadow video card, it deals with the external state, i.e., the VRAM, in addition to the internal one. The contribution of this paper is to show that VM migration is possible with secure OOB-VNC, which is much more popular than a virtual serial console in real clouds.

For the system with nested virtualization, fast communication between the guest and cloud control VMs is used in Xen-Blanket [14]. Xen-Blanket runs the Blanket driver in the OS of the guest control VM. The driver invokes the guest hypervisor using a hypercall and the hypervisor communicates with the cloud control VM. In SDmigrate, in contrast, a process in the guest control VM bypasses the guest hypervisor and directly invokes the cloud hypervisor. Then, the cloud hypervisor communicates with the cloud control VM. The design policy of Xen-Blanket is to modify only the virtualized system inside the cloud VM, whereas that of SDmigrate is not to modify the virtualized system.

7. Conclusion

In this paper, we proposed SDmigrate, which continues to use secure OOB-VNC after VMs are migrated. SDmigrate runs fake devices in the virtualized system to save and restore the states of shadow devices transparently. Fake devices securely and efficiently communicate with the corresponding shadow devices outside the virtualized system. SDmigrate deals with the states of a shadow keyboard, mouse, and video card for secure OOB-VNC. In addition to the internal states, it deals with the external state of a shadow video card, i.e., the VRAM. We have implemented SDmigrate in Xen and confirmed that we could continue to use secure OOB-VNC after a VM was migrated. Also, it is shown that the degradation of migration performance was negligible.

Our future work is that SDmigrate supports various virtualized systems for running in the cloud VM. Currently, SDmigrate supports Xen and KVM as virtualized systems. Both use QEMU as a device emulator. Since our implementation depends on QEMU, we need to show that SDmigrate can be applied to other types of virtualized systems without using QEMU, e.g., Hyper-V.

Acknowledgment

This work was partially supported by JST, CREST Grant Number JPMJCR21M4, Japan. These research results were partially obtained from the commissioned research (No.05501) by National Institute of Information and Communications Technology (NICT), Japan.

References

[1] T. Richardson, Q. Stafford-Fraser K. R. Wood, and A. Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

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

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

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

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

[6] M. Li, W. Zang, K. Bai, M. Yu, and P. Liu. MyCloud: Supporting User-configured Privacy Protection in Cloud Computing. In *Proc. Annual Computer Security Applications Conf.*, pages 59–68, 2013.

[7] S. Futagami, T. Unoki, and K. Kourai. Secure Out-of-band Remote Management of Virtual Machines with Transparent Passthrough. In *Proc. Annual Computer Security Applications Conf.*, pages 430–440, 2018.

[8] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proc. Symp. Operating Systems Design and Implementation*, pages 423–436, 2010.

[9] 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*, pages 164–177, 2003.

[10] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux Virtual Machine Monitor. In *Proc. Ottawa Linux Symp.*, pages 225–230, 2007.

[11] CyberArk Software. Trust, Security & Passwords Survey, 2009.

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

[13] C. Tan, Y. Xia, H. Chen, and B. Zang. TinyChecker: Transparent Protection of VMs against Hypervisor Failures with Nested Virtualization. In *Proc. Int. Workshop Dependability of Clouds, Data Centers and Virtual Machine Technology*, 2012.

[14] D. Williams, H. Jamjoom, and H. Weatherspoon. The Xen-Blanket: Virtualize Once, Run Everywhere. In *Proc. European Conf. Computer Systems*, pages 113–126, 2012.

[15] S. Doddamani, P. Sinha, H. Lu, T. Cheng, H. Bagdi, and K. Gopalan. Fast and Live Hypervisor Replacement. In *Proc. Int. Conf. Virtual Execution Environments*, pages 45–58, 2019.

[16] Intel Corp. 4th Generation Intel Core vPro Processors with Intel VMCS Shadowing, 2013.

[17] J. T. Lim, C. Dall, S. Li, J. Nieh, and M. Zyngier. NEVE: Nested Virtualization Extensions for ARM. In *Proc. Symp. Operating Systems Principles*, pages 201–217, 2017.

[18] Oracle Corp. Oracle Database 2 Day DBA.

[19] IBM Corp. IBM Domino 10.0.1 Documentation.

[20] E. Zhai, G. Cummings, and Y. Dong. Live Migration with Passthrough Device for Linux VM. In *Proc. Ottawa Linux Symp.*, pages 261–267, 2008.

[21] P. Thakkar. RFC: Network Plugin Architecture (NPA) for vmxnet3. http://lkml.iu.edu/hypermail/linux/kernel/1005.0/01142.html.

[22] A. Kadav and M. Swift. Live Migration of Direct-access Devices. *SIGOPS Oper. Syst. Rev.*, 43(3):95–104, 2009.

[23] Z. Pan, Y. Dong, Y. Chen, L. Zhang, and Z. Zhang. CompSC: Live Migration with Pass-through Devices. In *Proc. Int. Conf. Virtual Execution Environments*, pages 109–120, 2012.

[24] X. Xu and B. Davda. SRVM: Hypervisor Support for Live Migration with Passthrough SR-IOV Network Devices. In *Proc. Int. Conf. Virtual Execution Environments*, pages 65–77, 2016.

[25] T. Unoki and K. Kourai. VM Migration for Secure Out-of-band Remote Management with Nested Virtualization. In *Proc. Int. Conf. Cloud Computing*, pages 517–521, 2020.