Inokuchi and Kourai *Journal of Cloud Computing: Advances, Systems and Applications* (2020) 9:3
https://doi.org/10.1186/s13677-020-0152-9

Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH Open Access

Secure VM management with strong user binding in semi-trusted clouds

Keisuke Inokuchi and Kenichi Kourai*

Abstract

In Infrastructure-as-a-Service (IaaS) clouds, remote users access provided virtual machines (VMs) via the management server. The management server is managed by cloud operators, but not all the cloud operators are trusted in semi-trusted clouds. They can execute arbitrary management commands to users' VMs and redirect users' commands to malicious VMs. We call the latter attack the *VM redirection attack*. The root cause is that the binding of remote users to their VMs is weak. In other words, it is difficult to enforce the execution of only users' management commands to their VMs. In this paper, we propose *UVBond* for strongly binding users to their VMs to address this issue. UVBond boots user's VM by decrypting its encrypted disk *inside* the trusted hypervisor. Then it issues a *VM descriptor* to securely identify that VM. To bridge the semantic gap between high-level management commands and low-level hypercalls, UVBond uses *hypercall automata*, which accept the sequences of hypercalls issued by commands. We have implemented UVBond in Xen and created hypercall automata for various management commands. Using UVBond, we confirmed that a VM descriptor and hypercall automata prevented insider attacks and that the overhead was not large in remote VM management.

Keywords: Virtual machines, Clouds, Remote management, Hypercall automata, Disk encryption

Introduction

Infrastructure-as-a-Service (IaaS) clouds provide users with virtual machines (VMs). Users can install their own operating system and applications as they like. They manage provided VMs from remote hosts via the web interface or API. When they perform remote management of their VMs, they first connect to the management server provided in a cloud and then access their VMs via the server. For example, the management server has the ability for booting new VMs, shutting down running VMs, and migrating VMs to other hosts. In addition, users can log in VMs using a feature called out-of-band remote management, which allows users to access virtual serial and graphical consoles without servers running inside the VMs.

Although the management server is managed by cloud operators, not all the operators are trusted in semi-trusted clouds [1–8]. Cloud operators are hired by cloud providers and perform daily management in clouds. In semi-trusted clouds, their providers are trusted but some of the cloud operators may be untrusted. This is a usual situation according to several reports [9, 10]. Untrusted cloud operators can abuse the privileges of the management server and mount attacks against users' VMs. They can execute arbitrary management commands to VMs and eavesdrop on and tamper with their sensitive information. In addition, they can redirect users' commands to malicious VMs. We call this attack *VM redirection attack*, which is a kind of man-in-the middle attack. Using this attack, they can steal users' console input inside the malicious VMs. The root cause of these attacks is that the binding of remote users to their VMs is weak. It is difficult to enforce the execution of only users' management commands to their VMs.

In this paper, we propose UVBond, which strongly binds users to their VMs via encrypted disks of VMs. In UVBond, the trusted computing base (TCB) includes only the hypervisor and hardware. UVBond boots user's VM by decrypting its encrypted disk *inside* the trusted hypervisor and issues a *VM descriptor* to securely identify that VM. Using this descriptor, UVBond guarantees that management commands specified by the user are executed only to the user's VM. Untrusted cloud operators

*Correspondence: kourai@ksl.ci.kyutech.ac.jp
Kyushu Institute of Technology, 680-4 Kawazu, Iizuka, 8208502 Fukuoka, Japan

cannot execute commands to users' VMs. They cannot redirect users' access to their malicious VMs. To control the execution of management commands only in the hypervisor, UVBond uses *hypercall automata* as a whitelist, which accept only the sequences of hypercalls issued to the hypervisor by users' commands. A hypercall is the interface from management software and VMs to the hypervisor and executes privileged operations. As long as a hypercall sequence is not rejected, UVBond permits user's access to the VM corresponding to a VM descriptor.

We have implemented UVBond in Xen 4.4.0 [11]. UVBond encrypts and decrypts virtual disks of VMs only in the hypervisor. It supports paravirtual disk drivers, which are mandatory for efficient disk access but difficult to handle only in the hypervisor. For its secure implementation, UVBond duplicates I/O rings and grant pages in Xen. To distinguish multiple management commands executed simultaneously, UVBond identifies each process in the hypervisor and applies one hypercall automaton to one process. We have created hypercall automata for various management commands. UVBond also supports secure VM resumption and migration. According to our experiments, it was confirmed that cloud operators could not execute management commands to user's VM or redirect user's commands to a malicious VM. In addition, it was shown that the overhead of using hypercall automata was negligible and the degradation of disk performance was up to 9.5%.

The main contributions of this paper are as follows:

- We identify that untrusted cloud operators can execute management commands to arbitrary VMs because the binding of remote users to their VMs is weak.
- We enable strong binding of remote users to their VMs via encrypted disks of the VMs.
- We use hypercall automata to bridge the semantic gap between high-level management commands and low-level hypercalls.
- We implement UVBond for paravirtual disk drivers in Xen and created various hypercall automata.
- We confirm that cloud operators cannot execute management commands to arbitrary VMs with acceptable overhead.

This paper is an extended version of our previous conference paper [12]. In this paper, we clarified how we have created hypercall automata for various management commands through an experiment. We showed six complete hypercall automata and explained the relationship between command behavior and hypercall sequences in detail. Also, we conducted several experiments to further investigate the execution overhead of management commands in UVBond. To make it easier to serialize hypercall automata, we have developed an automaton converter. In addition, we added related work and made our contribution clearer.

The organization of this paper is as follows. The next sections describe the abuse of VM management by untrusted cloud operators and various systems related to UVBond. Then, we propose UVBond for strongly binding users to their VMs and explains the implementation details. Next, the paper shows our experimental results with UVBond. Finally, we conclude this paper and describe future work.

VM management in semi-trusted clouds

In IaaS clouds, users manage their VMs via the management server, which is part of the cloud management system. A user first sends a management command with a VM name and other parameters to the management server. Then the server communicates with the agent running in one of the compute nodes and the agent executes privileged operations to the virtualized system, especially the hypervisor. Hereafter, we regard the agents in compute nodes as part of the management server. The management server often runs in a privileged VM called the management VM. For example, the management server can boot new VMs, shut down running VMs, and migrate VMs to other hosts. Through the management server, users can log in VMs using out-of-band remote management, which enables indirectly managing VMs via their virtual serial and graphical consoles.

The management server is managed by cloud operators, but not all the operators are trusted in semi-trusted clouds [1–4, 6–8]. Semi-trusted clouds are provided by reputable cloud providers and are basically trusted. However, since they hire many operators for daily management, it is difficult to guarantee that all of the operators are trusted. In fact, it is reported that 28% of cybercrimes are caused by insiders [9]. Malicious system administrators attack systems actively. As a real example, a site reliability engineer in Google violated user's privacy [13]. In addition, curious but honest system administrators may eavesdrop on attractive information that they can easily obtain from VMs. It is revealed that 35% of system administrators have accessed sensitive information without authorization [10]. Therefore, many commercial systems assume the existence of untrusted operators. For example, Oracle Database provides two types of administrative privileges: SYSDBA for full administration and SYSOPER for basic operations [14]. IBM Domino can restrict access privileges to eight types of administrators [15].

Such untrusted cloud operators can abuse the management server or its privileges and attack users' VMs. This is because the binding of remote users to their VMs is weak. First, cloud operators can execute arbitrary management commands to VMs. For example, they can send magic system requests to VMs, which emulates pressing magic

SysRq keys. The magic SysRq key is a key combination for the user to perform various low-level commands to the operating system. Then they can show all the register values, which may contain sensitive information, to their serial consoles. Using a technique called *VM introspection* (VMI) [16], cloud operators can access the memory and disks of VMs from the outside of the VMs. Then they can analyze memory data and disk data and monitor the internal data of the operating system and the file system. If untrusted cloud operators abuse this mechanism, they can eavesdrop on sensitive information, e.g., cryptographic keys, in the memory and disks of VMs. Similarly, they can tamper with the memory and disks. In addition, they can eavesdrop on and tamper with console input and output of out-of-band remote management.

Second, cloud operators can redirect users' management commands to their malicious VMs. We call this attack the *VM redirection attack*. This attack is a kind of man-in-the-middle attack. Untrusted cloud operators can alter the communication between remote users and the hypervisor. As illustrated in Fig. 1, the VM redirection attack changes a VM accessed by a user in the management server. To eavesdrop on sensitive information, cloud operators create a malicious VM in which malware is installed and execute user's command to the VM. For example, they can steal login passwords in out-of-band remote management by using a malicious login program or a key logger. Since these malicious activities are done inside VMs, they are difficult to prevent even if console input and output are encrypted between remote users and the hypervisor [17, 18].

In addition, the VM redirection attack can be mounted for preventing VMI-based monitoring systems from detecting malicious activities in VMs. If cloud operators prepare a VM with a legitimate memory image, monitoring systems are fooled as normal even when their target VMs have been compromised. To mount the VM redirection attack to a cloud management system such as OpenStack [19], cloud operators can modify the source code of a cloud management system so that the specified operations are redirected to the specified VMs. This is easy because they have the privilege of updating and restarting a cloud management system.

Related work

Secure VM management on trusted hypervisors

Self-Service Cloud (SSC) [4, 20] can prevent cloud operators from illegally accessing users' VMs. For each user, it provides a dedicated management VM called Udom0, which is not interfered by cloud operators. Since each user's VMs can be managed only via his Udom0, cloud operators cannot eavesdrop on or tamper with the VMs. The disk integrity of Udom0 is verified with vTPM, which runs in a domain builder called domB. The user accesses Udom0 using a management server called a dashboard through an SSL channel, which is securely established at the boot time of Udom0. User's VMs are securely created via domB.

However, the TCB of SSC is quite large because it includes not only the hypervisor and hardware but also several privileged VMs such as Udom0, domB, and a dashboard VM. Udom0 is not a system-level TCB but a client-level one, whose compromise affects user's VMs. Udom0 is protected by the trusted hypervisor, but information leakage and tampering can occur if the system inside Udom0 is compromised by exploiting its vulnerabilities. Similarly, the systems inside domB and a dashboard VM can be compromised. The system including the operating system in such privileged VMs has a much larger attack surface than the hypervisor. The TCB of UVBond is smaller because it does not include any privileged VMs.

As such, the disaggregation of the management VM can minimize the privilege of cloud operators. SSC splits the management VM into system-wide Sdom0 and user-specific Udom0. Disaggregated Xen [21] moves complex VM-building functionality to another VM called DomB, as done in SSC. Furthermore, Xoar [22] breaks the management VM into many single-purpose components called service VMs. Compared with the original management VM, such service VMs need less privilege and can limit necessary hypercalls. If cloud operators are permitted to manage only the minimal number of service VMs, the ability of executing management commands can be restricted. However, the management server needs to be managed by cloud operators and to execute various management commands to users' VMs. To minimize the privilege of the management VM, MyCloud [5] runs the

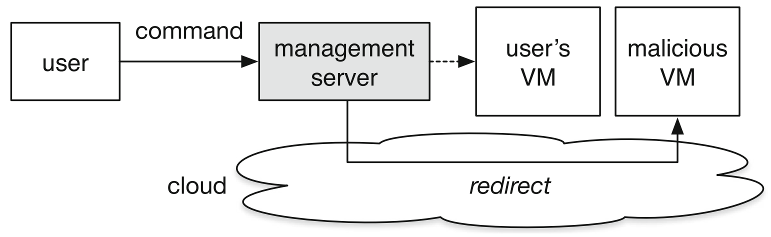

Fig. 1 The VM redirection attack

management VM in the less-privileged mode and removes it from the TCB. Then the hypervisor controls access from the management VM to users' VMs. Permitted access can be configured by users.

Secure Runtime Environment [2, 23] and VMCrypt [24] prevent information leakage from the memory of VMs to cloud operators. These systems encrypt the memory of VMs only when cloud operators access it from the management VM, while the systems do not when the VMs themselves access its memory. This memory encryption is done in the trusted hypervisor. Since VM migration, suspension, and resumption do basically not need to understand data structure inside the memory of VMs, these systems can support such management commands securely. One drawback is that even legitimate users cannot introspect the memory of their VMs using VMI. RemoteTrans [6] enables users to securely introspect their VMs from trusted remote hosts. Remote clients communicate with the trusted hypervisor using encryption and obtain memory data, disk blocks, and network packets of their VMs. However, RemoteTrans cannot prevent the VM redirection attack in remote VMI. It is not guaranteed that VMs specified by users are actually introspected.

FBCrypt [17] and SCCrypt [18] prevent cloud operators from eavesdropping on console input and output of out-of-band remote management using VNC and SSH, respectively. A remote client encrypts input data and then the trusted hypervisor decrypts it transparently when a VM attempts to obtain the data from a virtual keyboard or serial device. When a VM attempts to write output data to a virtual video or serial device, the hypervisor encrypts it and then the remote client decrypts it. FBCrypt can also detect tampering with input data using message authentication code. However, these systems cannot also prevent the VM redirection attack.

Weatherman [25] intercepts management commands at the authorization proxy and relays it to the management server only if no security is violated. Therefore, untrusted cloud operators can modify the commands at either the proxy or the management server unless we assume that these are trusted. In addition, we need to assume that compute nodes running the hypervisor are trusted. OpenStack Congress [26] simulates a new policy and enforces it after its impact is verified. It also requires that both the Congress component, the management server, and compute nodes need to be trusted.

Secure VM management on untrusted hypervisors

HypSec [27] partitions the hypervisor into the trusted corevisor and the untrusted hostvisor to protect the confidentiality and integrity of VM data in a small TCB. At the boot time of a VM, the corevisor checks the signature of a VM image using user's public key. The signature and key have to be stored in secure storage such as ARM TrustZone and TPM in advance. This enables only pre-registered user's VMs to be booted. However, the signature changes after the VM image is modified by the execution of the VM. In clouds, it is not realistic to calculate a new signature from a large VM image and send the VM image with the signature whenever a VM is booted. The issue of a hypercall is first trapped by the corevisor and then delegated to the hostvisor. At this time, the corevisor can check the target VM of the hypercall, but HypSec does not provide any mechanism to prevent untrusted operators from maliciously executing hypercalls.

HA-VMSI [28] protects VMs even from a compromised hypervisor by running a trusted security monitor in the ARM TrustZone secure world. Like HypSec, the security monitor checks the hash value of a VM image at the boot time. After the verification, it returns a VM identifier, which is used for interaction between the hypervisor and the security monitor. The VM identifier is similar to our VM descriptor, which is returned to a remote user. However, HA-VMSI just provides isolation of VM's memory against an untrusted hypervisor and cannot control the execution of management commands. In addition, it requires ARM CPUs with the TrustZone feature, which are not yet widely used in clouds.

Fidelius [29] protects VMs from hardware-level memory eavesdropping using AMD SEV. SEV enables each VM to selectively encrypt the memory against an untrusted hypervisor. To address several security issues of SEV, Fidelius provides a software-based extension to SEV and introduces an isolated context with non-bypassable memory protection to manage critical resources. This isolated context is built in the same privilege level as the hypervisor. Besides, Intel SGX can be used to securely execute programs in enclaves on an untrusted hypervisor [30]. It enables remote clients to establish secure connections with enclaves using remote attestation. However, enclaves cannot execute management commands because they prohibit privileged operations such as the issue of hypercalls.

Nested virtualization [31] has been also used to achieve secure VM management. It is a technique for enabling the entire virtualized system to be run in a VM. Using this technique, the hypervisor can be excluded from the TCB. CloudVisor [3] securely controls access to the memory of users' VMs in the trusted security monitor running below the untrusted hypervisor. Cloud operators inside the virtualized system cannot access the memory using VMI. Like the above systems [2, 23, 24], CloudVisor securely encrypts the memory when users' VMs are migrated. However, the overhead of nested virtualization is not small.

VSBypass [8] enables secure out-of-band remote management outside the virtualized system to prevent information leakage to cloud operators. It runs the entire virtualized system in an outer VM and processes I/O requests of out-of-band remote management outside the VM. To identify users' VMs running in the virtualized system from the outside, each VM securely registers a VM tag to the outer trusted hypervisor and the remote user specifies the tag to manage his VM. This can prevent the VM redirection attack inside the virtualized system, but it is necessary that remote users securely obtain information on VM tags randomly generated in VMs.

Secure disk encryption

Several techniques for secure disk encryption have been proposed. SSC [4] enables users to encrypt the disks of VMs using special VMs called service domains (SDs). Each user can run his own SDs, while cloud operators cannot interfere with user's SDs. Like Udom0, SDs are also a client-level TCB and are protected by the trusted hypervisor. However, once the systems inside SDs are compromised, disk data is leaked or tampered with.

BitVisor [32] can encrypt the disk of a VM using a parapass-through driver in the trusted hypervisor. The hypervisor intercepts only minimum hardware access needed for disk encryption, while the other access passes through the hypervisor. The driver for the ATA host controller supports not only PIO but also DMA transfers by using shadow DMA descriptors and shadow buffers. UVBond can also support DMA transfers using the same technique. Unlike UVBond, BitVisor supports only fully virtualized operating systems and cannot use para-virtual disk drivers in a VM.

CloudVisor [3] encrypts the disks of VMs in the security monitor using nested virtualization. It intercepts I/O requests of VMs and encrypts or decrypts data. In addition, CloudVisor checks the integrity of the disks. It provides necessary hash data to the security monitor via the management VM. Using the hash data, it is guaranteed that VMs are booted properly with the disk images specified by users. As pointed out in the above, the large overhead of nested virtualization is often critical for disk performance.

System-call automata

System-call automata [33] are used for intrusion detection systems (IDSes). Such IDSes detect intrusion on the basis of a sequence of system calls issued by a process. They trace the program execution in advance and record normal behavior as a system-call automaton, which is a whitelist. If a process issues a system call that is not accepted by the automaton, the IDSes detect abnormal behavior. A hypercall automaton used in UVBond is an application to the hypervisor.

UVBond

Threat model

We assume that only the hypervisor and hardware are the TCB. To trust hardware, we assume that cloud providers themselves are trusted. This assumption is widely accepted [1–4, 6–8] because bad reputations are critical for them. The trustworthiness of the hypervisor can be confirmed by various techniques. For example, remote attestation with TPM guarantees that the hypervisor is booted correctly. Security checks with the system management mode (SMM) [34–36] and AMD SVM [37] can detect attacks against the hypervisor at runtime. Also, event-driven monitoring with same-privilege isolation [38] can be used to detect attacks.

We do not trust cloud operators or privileged components managed by them. Since cloud providers hire many operators, it is difficult to trusted all of them in large-scale clouds. Such privileged components include the management server and agents in compute nodes. Since they are often run in a privileged VM, e.g., the management VM, we do not trust the entire privileged VM running on top of the hypervisor. We assume that the privileged VM can be abused by untrusted cloud operators. For example, cloud operators can compromise not only the management server but also the operating system and device emulators running for user's VMs in the privileged VM. Since they can take the root access in the privileged VM, they can issue arbitrary hypercalls to the hypervisor to perform VM management. However, we assume that cloud operators can legitimately access the hypervisor using only the hypercall interface. We mainly consider untrusted cloud operators as insiders, but outside attackers can be regarded as insiders after they intrude into clouds and take high privileges.

There is a broad attack surface in virtualized systems [39, 40] and IaaS clouds [41], but this paper focuses on weak binding of remote users to their VMs in clouds. Due to this problem, untrusted cloud operators can execute malicious commands to users' VMs and redirect management commands to malicious VMs. Therefore, we need to combine UVBond with other security mechanisms to prevent complex attacks beyond the simple execution of management commands. For example, indirect information leakage and tampering after command execution and attacks against device emulators in the privileged VM need to be protected by other mechanisms. DoS attacks against command execution are also out of the scope of this paper.

Overview of UVBond

When a new VM is created, UVBond securely binds its user to the created VM. This is guaranteed by the fact that only a user has a cryptographic key for the encrypted disk of a VM. Unlike previous work [27, 28], UVBond does not verify the signature of a disk to avoid signature re-calculation on every VM boot, but it relies on user's confirmation of the correct boot. After a VM is booted, UVBond issues a VM descriptor to the user. Using this descriptor, the user securely executes management commands to his VM. To control the execution of management commands only in the trusted hypervisor, UVBond uses hypercall automata as a whitelist, which accept only the sequences of hypercalls issued by users' commands. As long as a hypercall sequence is not rejected, UVBond permits user's access to the VM corresponding to a VM descriptor.

Strong binding of users to VMs

UVBond strongly binds users to their VMs via encrypted disks of the VMs. It uses disk encryption performed in the trusted hypervisor, instead of traditional disk encryption inside each VM. First, a user securely shares his disk encryption key with the hypervisor at the boot time of his VM. Then the hypervisor associates the key with the VM. Using the registered disk encryption key, UVBond boots the VM by decrypting its encrypted disk inside the hypervisor, as illustrated in Fig. 2. Since the VM cannot be correctly booted using the other virtual disks that do not correspond to the key, it is guaranteed that user's own VM is certainly booted. Note that cloud operators can still boot their VM with a malicious disk and its encryption key as user's VM. To prevent this attack, UVBond enables the user to confirm that his disk encryption key is correctly registered.

After the boot of user's VM, UVBond issues a *VM descriptor* to the user. The descriptor is associated with the VM inside the hypervisor. It is encrypted by the hypervisor and is sent to the user. The user specifies the descriptor when he executes management commands to his VM. On the basis of the descriptor, the hypervisor determines whether hypercalls can be issued to the VM corresponding to the descriptor. Using the descriptor, UVBond can prevent cloud operators from accessing users' VMs. Only the user that has the descriptor is permitted to access his VM. As illustrated in Fig. 3, UVBond can also prevent the VM redirection attack, which forces a user to access a malicious VM. A user can always access the VM that matches the descriptor.

UVBond can prevent direct information leakage and tampering by the execution of management commands. However, it needs to be combined with other security mechanisms to prevent indirect attacks after command execution. This is because the access control with hypercalls is not sufficient for such indirect attacks. For VMI, only a user can map the memory of his VM onto a process using hypercalls, but cloud operators can access the process memory illegally. Therefore it is necessary to obtain memory data of a VM from the trusted hypervisor in an encrypted form and inspect it at remote hosts using RemoteTrans [6]. Note that cloud operators cannot introspect the disk of a VM because the disk is encrypted in UVBond. For out-of-band remote management, cloud operators can eavesdrop on and tamper with console input and output via virtual devices of VMs without issuing hypercalls. FBCrypt [17] and SCCrypt [18] should be used to encrypt the data between the remote user and the hypervisor. As a result, cloud operators can obtain only encrypted data from virtual devices. Since UVBond prevents the registration of that encryption key from being redirected to a malicious VM, cloud operators cannot obtain decrypted data even inside that VM.

Hypercall automaton

A VM descriptor should be able to control the execution of each management command separately, but this is not easy. UVBond assumes that only one management server is shared among all the users and cloud operators, as a traditional system architecture. If each command is exactly equivalent to one hypercall, a user can pass a pair of a command and a VM descriptor to the hypervisor. Then the hypervisor can securely execute the hypercall to the VM corresponding to the descriptor. However, each command usually consists of a set of hypercalls and the other tasks that cannot be executed inside the hypervisor. Since

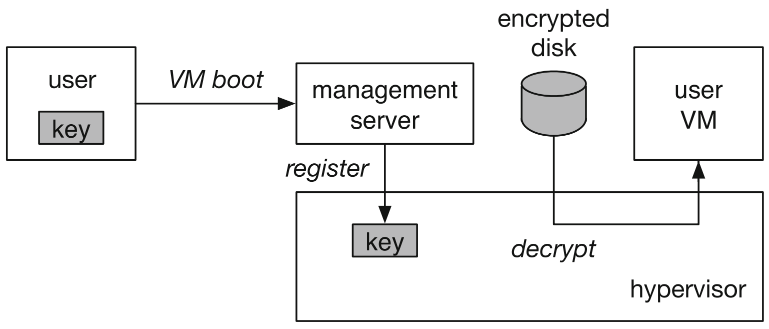

Fig. 2 Booting a VM using an encrypted disk

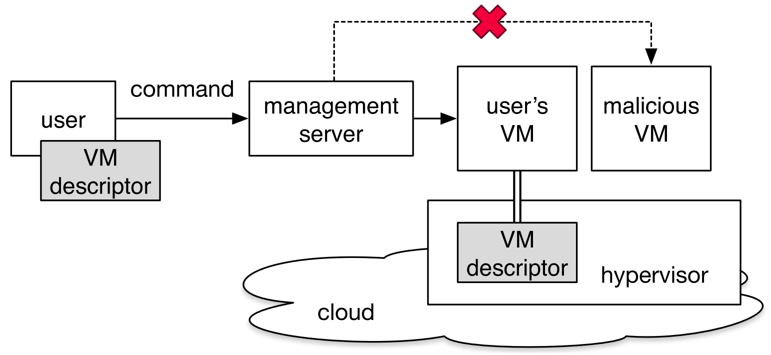

Fig. 3 Command execution with a VM descriptor

the hypervisor can recognize only hypercalls, it is difficult to securely associate a VM descriptor with the execution of each command.

To bridge this *semantic gap*, UVBond identifies each management command by a sequence of hypercalls. For each command, it creates a finite state automaton as a whitelist to accept all the sequences of hypercalls issued by the command in advance. This is called a *hypercall automaton*. In general, each state in a hypercall automaton has multiple transitions. For example, one command can have different hypercall sequences for different parameters or in different execution environment. Therefore its hypercall automaton can be created as illustrated in Fig. 4. Note that the input is a hypercall and, if any, its subdivided operation. When a user accesses his VM, he sends a hypercall automaton as well as the corresponding command and a VM descriptor. The hypervisor permits access to the VM corresponding to the descriptor as long as a hypercall sequence issued by the command is not rejected by the hypercall automaton. If UVBond rejects a hypercall sequence, it returns an "operation not permitted" error to the last hypercall issued by the management command. The command should handle that error as if the user had no sufficient privilege. Note that the abilities of legitimate users are not constrained basically because hypercall automata do not prevent legitimate operations.

We assume that the developers of management systems such as OpenStack identify the hypercall sequences of all the management commands. The hypercall sequence for each management command is collected by running the command with various parameters in various conditions. For example, developers execute commands for both fully virtualized and para-virtualized VMs. This can construct a hypercall sequence including various execution paths.

Even if a hypercall sequence is accepted by the specified hypercall automaton, the management command intended by a user is not always executed. Cloud operators might be able to execute another command whose hypercall sequence is accepted by the hypercall automaton but whose behavior is different. However, commands that generate the same hypercall sequence essentially access a VM in the same manner because a VM can be managed only via hypercalls. Even if an executed command is different from one specified by the user, those commands can be considered as the same in terms of VM management. It may be still possible for attackers to specially craft malicious commands with legitimate hypercall sequences, but hypercall automata at least can make it more difficult to execute malicious commands and raise the bar for attacks.

Using hypercall automata, UVBond can permit some of the management commands even to cloud operators. If only a user has to manage his VMs completely, his burden would become too large. For example, it would be desirable that cloud operators can migrate VMs when the host running the VMs is maintained. To allow cloud operators to execute management commands without a VM

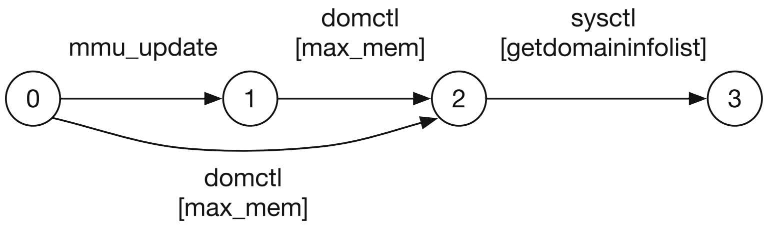

Fig. 4 An example of a hypercall automaton

descriptor, a user registers the corresponding hypercall automata to his VMs in advance. While the commands corresponding to the registered hypercall automata are executed, even cloud operators can access the specified VMs. Users can determine permitted commands at their discretion and take a trade-off between ease of management and security. As such, cloud operators have more abilities than previous work [4] although hypercall automata can restrict the abilities of cloud operators.

Implementation

We have implemented UVBond in Xen 4.4.0 [11]. In Xen, the management server runs in a privileged VM called Dom0 and cloud operators manage users' VMs in Dom0. We have ported AES and RSA in wolfSSL [42] to use them in the hypervisor. We have also developed a management client for UVBond using OpenSSL.

UVBond can be implemented in the other Type-I hypervisors, but it is not easy to implement in Type-II hypervisor such as KVM [43]. In KVM, for example, management commands are executed on top of the host operating system providing the hypervisor functions. To use the hypervisor functions, they access the KVM device provided by the host operating system, instead of hypercalls. In addition, we need to trust the entire host operating system, whose TCB is much larger than the hypervisor.

Overview of secure VM management

When a new VM is created, an AES key for disk encryption is created and the disk image of the VM is encrypted using the key in the management client. The encrypted disk image is uploaded to a cloud and is stored in Dom0 as usual.

Whenever the VM is booted, the management client generates an AES session key and encrypts it and the disk encryption key using the RSA public key of the hypervisor. The public key is obtained from a trusted key server or in the form of a digital certificate from the management server. Then the management client sends the boot command with the encrypted keys to Dom0. The management server in Dom0 issues a new hypercall for key registration and passes the encrypted keys to the hypervisor. The hypervisor decrypts the passed keys using its own RSA private key and registers the decrypted keys to a being booted VM. Cloud operators cannot decrypt the keys.

According to the boot command, the management server boots a VM with the encrypted disk specified by the user. When the VM accesses its virtual disk, the hypervisor intercepts that access and then decrypts data to be read or encrypts data to be written using the registered disk encryption key. At the same time, the management client checks the correctness of the keys registered to the hypervisor.

After the boot of the VM, the management server issues a new hypercall and obtains a VM descriptor for the VM from the hypervisor. This descriptor is encrypted with the registered session key. The management server sends the encrypted descriptor back to the client and then the client decrypts it using the session key. To execute a management command, the client encrypts a pair of this descriptor and the hypercall automaton corresponding to the command using the session key and sends the encrypted pair to the server. After the management server completes executing the command, it obtains the result of transitions encrypted by the session key from the hypervisor and returns it to the client. The keys used by UVBond are listed in Table 1.

Encryption of para-virtualized disk I/O

To access virtual disks of VMs, paravirtual disk drivers are often used. This is because the disk performance is largely improved, compared with using fully virtualized disk drivers. Although Xen supports both para-virtualized and fully virtualized operating systems, Linux uses the paravirtual disk driver by default even in full virtualization. Therefore, the hypervisor has to support disk encryption in para-virtualization. However, this is not easy because the hypervisor cannot trap all accesses to virtual disks.

Traditional disk I/O

Xen uses the split device model, as illustrated in Fig 5. The paravirtual disk driver consists of the front-end driver called *blkfront* running in a VM and the back-end driver called *blkback* running in Dom0. These drivers share the memory region used for a ring buffer called an *I/O ring*. They communicate with each other using the I/O ring and a signaling mechanism called an *event channel*. When a VM performs disk I/O, the blkfront driver writes a request to the I/O ring and sends an event to the blkback driver via the event channel. When the blkback driver receives that event, it obtains the request from the I/O ring and accesses the disk image of the VM. Upon disk read, it reads data from the disk image and writes the data to the specified

Table 1 The keys used by UVBond

Key	Created by	Stored in	Purpose
Disk encryption key	User	Client	Encrypt a virtual disk
Session key	Client	Client	Encrypt a VM descriptor, a hypercall automaton, and a command result
Public key	Provider	Key server	Encrypt a disk encryption key and a session key
Private key	Provider	Hypervisor	Decrypt a disk encryption key and a session key

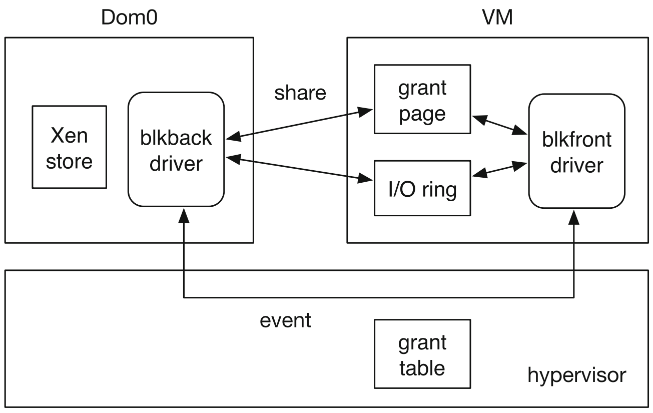

Fig. 5 Xen's split device model

memory page in the VM. Upon disk write, it reads data from the specified memory page in the VM and writes the data to the disk image. Then the blkback driver writes a response to the I/O ring and sends an event to the blkfront driver in the VM.

To share memory pages between Dom0 and a VM, a mechanism called a *grant table* is used in Xen. A VM first registers pages that it intends to share to its grant table. These pages are called *grant pages*. A *grant reference* is assigned to each grant page and Dom0 can map and access a grant page by specifying that grant reference. The grant reference of the page used for the I/O ring is passed from the blkfront driver to the blkback driver via the database called *XenStore* in Dom0. For grant pages used for the buffer of disk I/O, their grant references are passed via requests written to the I/O ring.

Duplication of grant pages

To prevent untrusted cloud operators in Dom0 from eavesdropping on data in grant pages after decryption and before encryption by the hypervisor, UVBond duplicates grant pages used for the I/O buffer, as depicted in Fig. 6. For each grant page, it provides an unencrypted page to a VM and an encrypted page to Dom0. A page provided to a VM is called a *guest grant page* and one provided to Dom0 is called a *shadow grant page*. Data in a guest grant page is encrypted and written to the corresponding shadow grant page, while that in a shadow grant page is decrypted and written to the corresponding guest grant page.

UVBond also duplicates the grant page used for the I/O ring but does not encrypt its shadow one. The I/O ring provided to a VM is called a *guest I/O ring*, whereas that provided to Dom0 is called a *shadow I/O ring*. This duplication is used only for synchronization between the guest and shadow I/O rings. The hypervisor copies a request from the guest I/O ring to the shadow one after it completes encrypting data in guest grant pages. Conversely, the hypervisor copies a response from the shadow I/O ring to the guest one after it completes decrypting data in shadow grant pages.

For compatibility with the original Xen, UVBond enables Dom0 to access a shadow grant page using the grant reference of the corresponding guest grant page. For a VM, it associates a grant reference with a guest grant page as usual. For Dom0, in contrast, it associates the same grant reference with the shadow grant page. Therefore, when Dom0 attempts to map a guest grant page passed from a VM, its shadow grant page is mapped instead. Similarly, when Dom0 attempts to unmap the guest grant page, its shadow grant page is unmapped. This gives an illusion to Dom0 as if Dom0 can access a guest grant page.

Disk I/O in UVBond

To perform encryption and decryption of disk I/O, the hypervisor identifies the page used for the I/O ring and creates a shadow I/O ring at the boot time of a VM. For this purpose, it analyzes the communication between a VM and Dom0. The blkfront driver in a VM registers the grant reference of the page used for the I/O ring to XenStore on initialization. For this registration, it writes a request to a ring buffer called the *XenStore ring* and sends an event to XenStore. UVBond intercepts that event in the hypervisor and obtains the grant reference for the I/O ring. The information on the page used for the XenStore ring is passed to the hypervisor at the boot time of a VM. After the hypervisor creates a shadow I/O ring, it copies the guest I/O ring to the shadow one. In addition, it obtains information on the event channel used by the blkfront driver from the request to XenStore.

When the blkfront driver sends a request to the blkback driver, the hypervisor encrypts the requested data using the disk encryption key if the request is for disk write. Since the hypercall for sending an event is called at

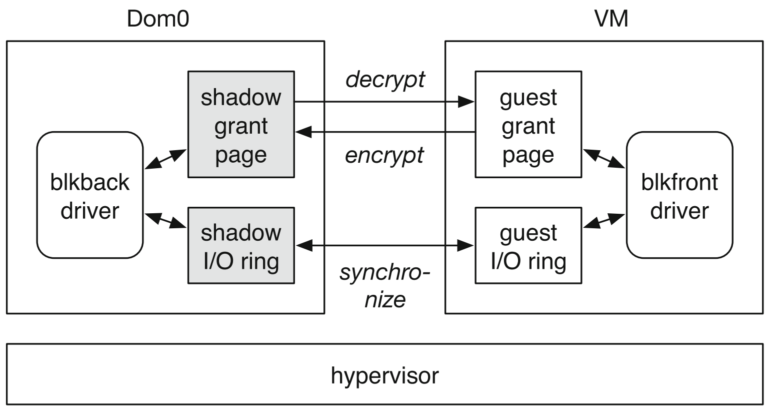

Fig. 6 The duplication of grant pages

that time, the hypercall analyzes the request written to the guest I/O ring. It creates a shadow grant page if there is no such a page corresponding to the grant reference included in the request. Then the hypervisor encrypts data stored in the guest grant page and writes it to the shadow grant page. Finally, it copies the request to the shadow I/O ring.

On the other hand, when the blkback driver sends a response to the blkfront driver, the hypervisor decrypts the returned data if the response is for disk read. Since the response includes no grant reference, the hypervisor saves the corresponding request in advance and obtains a grant reference from it. Then the hypervisor decrypts data in the shadow grant page that corresponds to the grant reference and writes it to the guest grant page. Finally, it copies the response to the guest I/O ring. When the guest grant page is released in a VM, the hypervisor releases the shadow one as well.

Using AES-NI in the hypervisor

We have ported the AES functions with AES-NI from wolfSSL, but special treatment was required to use AES-NI in the hypervisor. AES-NI is a CPU instruction set for AES to improve the performance of encryption and decryption. It needs to use XMM registers, which causes a hardware exception in the hypervisor. This is because the hypervisor in Xen defers the restoration of the XMM registers on CPU scheduling. When the hypervisor accesses one of the XMM registers and an exception occurs, it restores the XMM registers.

To prevent this exception, UVBond clears the TS bit in the CR0 register just before using AES-NI. This bit is set to cause an exception when XMM registers are accessed. At the same time, UVBond saves the XMM registers. After the use of AES-NI, it restores the XMM registers.

Encryption of fully virtualized disk I/O

UVBond supports not only para-virtualized but also fully virtualized disk I/O. Even for the operating systems using paravirtual disk drivers, fully virtualized disk I/O is used during the execution of BIOS, which is used before booting the operating system. When BIOS performs disk read by 512-byte data using the IN instruction, the hypervisor emulates that instruction. The read of 512-byte data causes two traps to the hypervisor when BIOS executes the IN instruction for the first 4-byte data and the repeat of the instruction for the remaining 508-byte data. Since the block length of AES is 16 bytes, UVBond decrypts 512-byte data as a whole on the latter trap. Currently, UVBond supports only programmable I/O (PIO).

Confirming registered keys

UVBond confirms that the disk encryption key and the session key registered to the hypervisor are user's when it returns a VM descriptor to the user. When the management server issues the hypercall for obtaining the VM descriptor for a started VM, the hypervisor appends the disk encryption key to the VM descriptor and encrypts them using the session key. While the management client receives the encrypted data, it decrypts the data using its own session key. If the disk encryption key is extracted correctly, the user can confirm that both the keys registered to the hypervisor are the same as user's.

Confirming encrypted disks

UVBond confirms that the used encrypted disk and the registered disk encryption key match correctly when a VM is booted. For this purpose, UVBond checks the boot sector of the disk. The boot sector is always read on a VM boot and a pre-defined magic number is stored in it. If the hypervisor cannot obtain that magic number after the decryption of the boot sector, UVBond can automatically detect illegal replacement of the entire encrypted disk.

To completely guarantee that an encrypted disk is user's, it is necessary to check the integrity of the entire disk, e.g., using a Merkle hash tree [44]. However, it is not realistic to store hash data in the hypervisor because extra hash data is needed in addition to the original disk data and the size of hash data becomes large. For example, 4 GB of hash data is needed for 1 TB of disk if we use a 256-bit hash value for 4-KB data. Without such integrity checking, users can confirm that correct encrypted disks are used if a VM is booted correctly. In practice, it is difficult for malicious cloud operators to create one encrypted disk image by combining correct and malicious data. As proposed in CloudVisor [3], it is possible to cache only a small amount of hash data inside the hypervisor. This is our future work.

Secure execution of management commands

Serialization of hypercall automata

When a user executes a management command to his VM, the management client sends an encrypted pair of a VM descriptor and a hypercall automaton as well as a command and a target VM name to the server. Since a hypercall automaton is a directed graph, UVBond represents it as one-dimensional array to make encryption and network transfers easy and efficient, as illustrated in Fig. 7. This array consists of a set of state information, each of which is a pair of an input for transiting to that state and the states to which the automaton can transit from that state. A state is represented as an array index and each input is a hypercall number and, if any, the number of its subdivided operation. For example, the domctl and sysctl hypercalls need to specify an operation because they are collective hypercalls and provide various functions. Note that this array format assumes that any transitions to each state are caused only by the same hypercall. This is because the management commands for which we have created hypercall automata had only such state transitions. If this condition is not satisfied, we can divide one state into multiple.

In the example of Fig. 7, array elements with indexes 0 and 1 mean two transitions from state 0 to 1 and 2, respectively. The number of transitions is determined by -1 stored in an element with index 2. The former transition occurs when the hypercall with the number stored in an element with index 3 is issued. The latter transition occurs when the hypercall with the number stored in an element with index 6 is issued and its operation is equal to the number stored in an element with index 7. Similarly, an element with index 4 means one transition from state 1 to 2, while one with index 8 means one transition from state 2 to 3. The latter transition occurs when the hypercall number is the value stored in an element with index 10 and its operation number is that stored in an element with index 11.

To make it easier to convert a hypercall automaton to such an array, we have developed an automaton converter. Using the converter, users can input state information with one input and multiple transitions. For each state, they are asked a hypercall number and, if any, an operation number as an input and state numbers as destinations of transitions, except for the initial state. The input is not asked for the initial state, while the transition should be -1 for the final states. The automaton converter generates an array for a hypercall automaton from the inputted state information. Thanks to the converter, users do not care about index numbers used in the array. The current converter provides only a simple interface and it is our future work to provide a domain-specific language for defining a hypercall automaton.

The management server translates the received VM name into the ID of a running VM and passes it and the received pair to the hypervisor before the execution of the specified management command. First, the hypervisor decrypts the pair using the session key registered to the specified VM. Then, it compares the decrypted VM descriptor with that registered to the VM. If these are the same, the hypervisor registers the decrypted hypercall automaton to the VM. Otherwise, it considers that

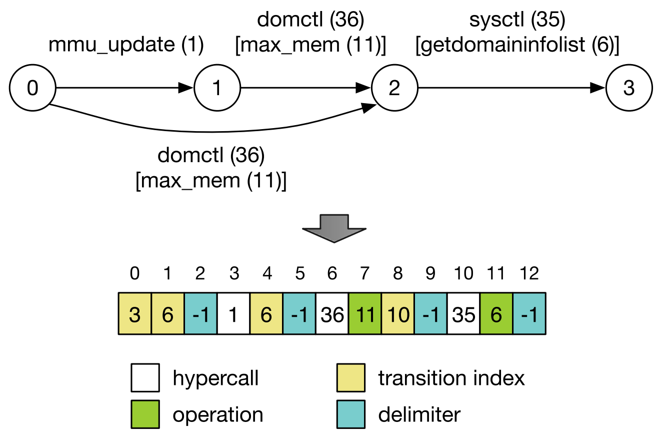

Fig. 7 The serialization of a hypercall automaton

as illegal access or corruption by attacks and returns an "operation not permitted" error encrypted by the session key. To prevent replay attacks, a monotonic counter can be used between the management client and the hypervisor.

Validity check with hypercall automata

Using the registered hypercall automaton, UVBond checks the validity of issued hypercalls while the command is executed. When a hypercall is issued, the hypervisor examines the states to which the automaton can transit from the current state. The current state is pointed by the index of the array for the hypercall automaton. If the issued hypercall is a possible input, the hypervisor updates the current index and transits to the next state. Then it permits the execution of the hypercall. If the hypercall is not permitted in the hypercall automaton, UVBond considers the issue of that hypercall as illegal and denies the execution. When the execution of the management command is completed, the management server obtains the status of the hypercall automaton from the hypervisor and returns it to the management client. The status is whether the hypercall sequence is accepted or not and is encrypted by the hypervisor. The management client can know whether the command is completed or not.

To distinguish hypercalls issued simultaneously by various management commands, UVBond applies the hypercall automaton only to the process that registers it. Since the hypervisor cannot identify processes of the operating system, UVBond uses the value of the CR3 register, as proposed in [45]. In this register, the physical address of the page directory of a process is stored and is unique to each process. When the hypervisor registers a hypercall automaton to a VM, it associates the current value of this register with the hypercall automaton. When a hypercall is issued, the hypervisor searches for the hypercall automaton on the basis of the value of the CR3 register and uses it, as illustrated in Fig. 8.

Secure VM resumption and migration

UVBond enables users to continue to securely manage their VMs after suspended VMs are resumed. VM suspension saves the states of a VM to a disk, while VM resumption restores them. Upon VM resumption, the user registers the disk encryption key of the target VM to the hypervisor again. At this time, cloud operators could register their key and resume the VM with their malicious disk. To prevent this attack, the hypervisor encrypts the CPU state of the VM using the disk encryption key on VM suspension, as illustrated in Fig. 9. Then, it decrypts the state using a newly registered key on VM resumption. If a malicious key is registered, the CPU state cannot be restored correctly and the VM cannot be restarted. Note that the user registers a new session key and receives a new VM descriptor.

For VM migration, UVBond enables VMs to be migrated without explicit key re-registration by users. Unlike VM resumption, a user cannot manually register his disk encryption key to the hypervisor at the destination host because he can send the migration command only to the source host. At the source host, UVBond obtains the disk encryption key and the session key from the hypervisor and transfers them to the destination host, as illustrated in Fig. 10. These keys are encrypted by the public key of the destination hypervisor. To use such a public key by specifying an IP address, UVBond registers pairs of an IP address and a public key to the hypervisor in advance. At the destination host, UVBond automatically registers the disk encryption key and the session key to the hypervisor again. These keys are decrypted by the private key of that hypervisor. Malicious

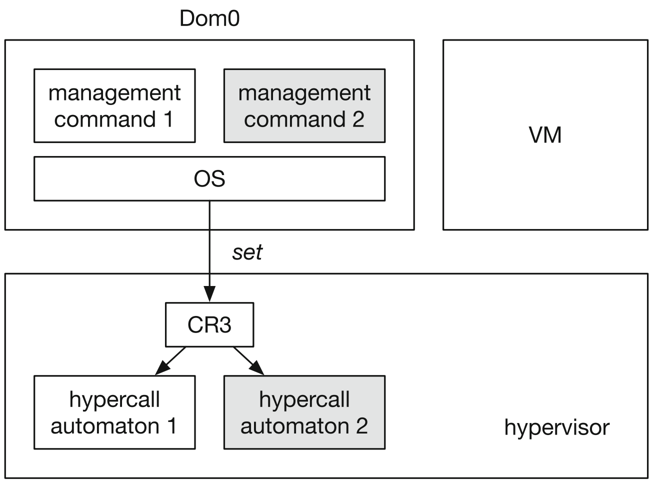

Fig. 8 Binding hypercall automata to management commands

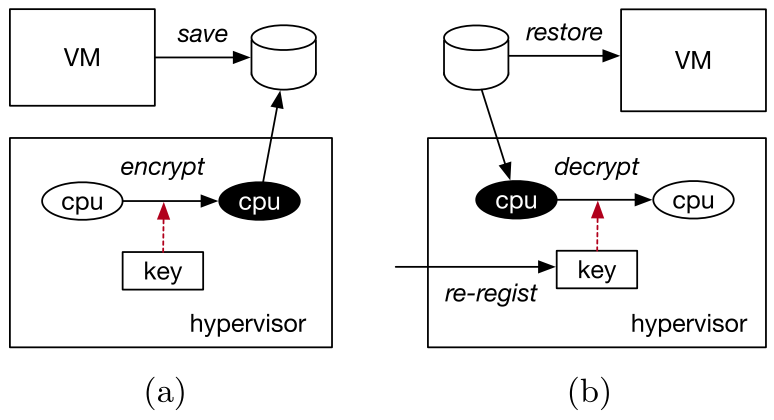

Fig. 9 Secure VM resumption using encrypted CPU state. **a** Suspension. **b** Resumption

replacement of the keys is prevented by the same mechanism as VM resumption. Note that the user can use the same VM descriptor before VM migration.

Experiments

We conducted experiments to confirm the effectiveness of UVBond. We used a PC with an Intel Xeon E3-1290 v2 processor, 8 GB of memory, and 1 TB of HDD. For a VM, we assigned two virtual CPUs, 1 GB of memory, and 20 GB of virtual disk. We used Xen 4.4.0 modified for UVBond and ran Linux 3.16 in Dom0 and Linux 3.13 in a VM. For comparison, we used vanilla Xen without modification. As a client host or a destination host of VM migration, we used a PC with the same spec and software as the above. These two hosts were connected with Gigabit Ethernet.

Creation of hypercall automata

To confirm that hypercall automata can be described for management commands, we have created various hypercall automata. First, we injected code for logging hypercalls into libxc, which is a core library provided by Xen. Then, we executed various commands with various parameters using Xen's xl tool and collected hypercall sequences. Using these hypercall sequences, we created hypercall automaton. This is not so difficult task because most of the commands always issued the same hypercall sequences even when we executed them with different parameters. Fig. 11 shows part of the created hypercall automata and Table 2 lists the used hypercalls.

Figure 11a shows the hypercall automaton for the pause command, which temporarily stops the execution of a VM. It first issues eight xen_version hypercalls. The xl tool always issues these hypercalls before issuing command-specific hypercalls. These hypercalls are used for executing different operations. Next, the command can issue the sysctl hypercall with the getdomaininfolist operation, which is hereafter denoted by sysctl [getdomaininfolist]. This hypercall is used to convert a VM name to its ID. It is always issued when a VM is specified by name. If the ID is directly specified, this hypercall is skipped. Finally, the command issues the domctl [pausedomain] hypercall, which stops all the virtual CPUs of the specified VM.

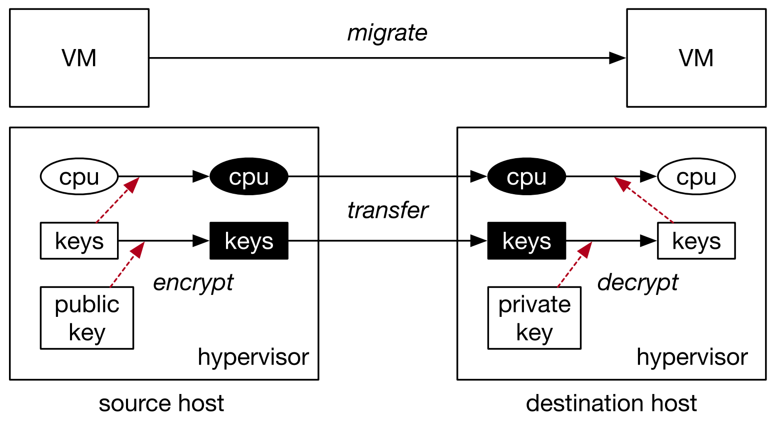

Fig. 10 Secure VM migration with keys

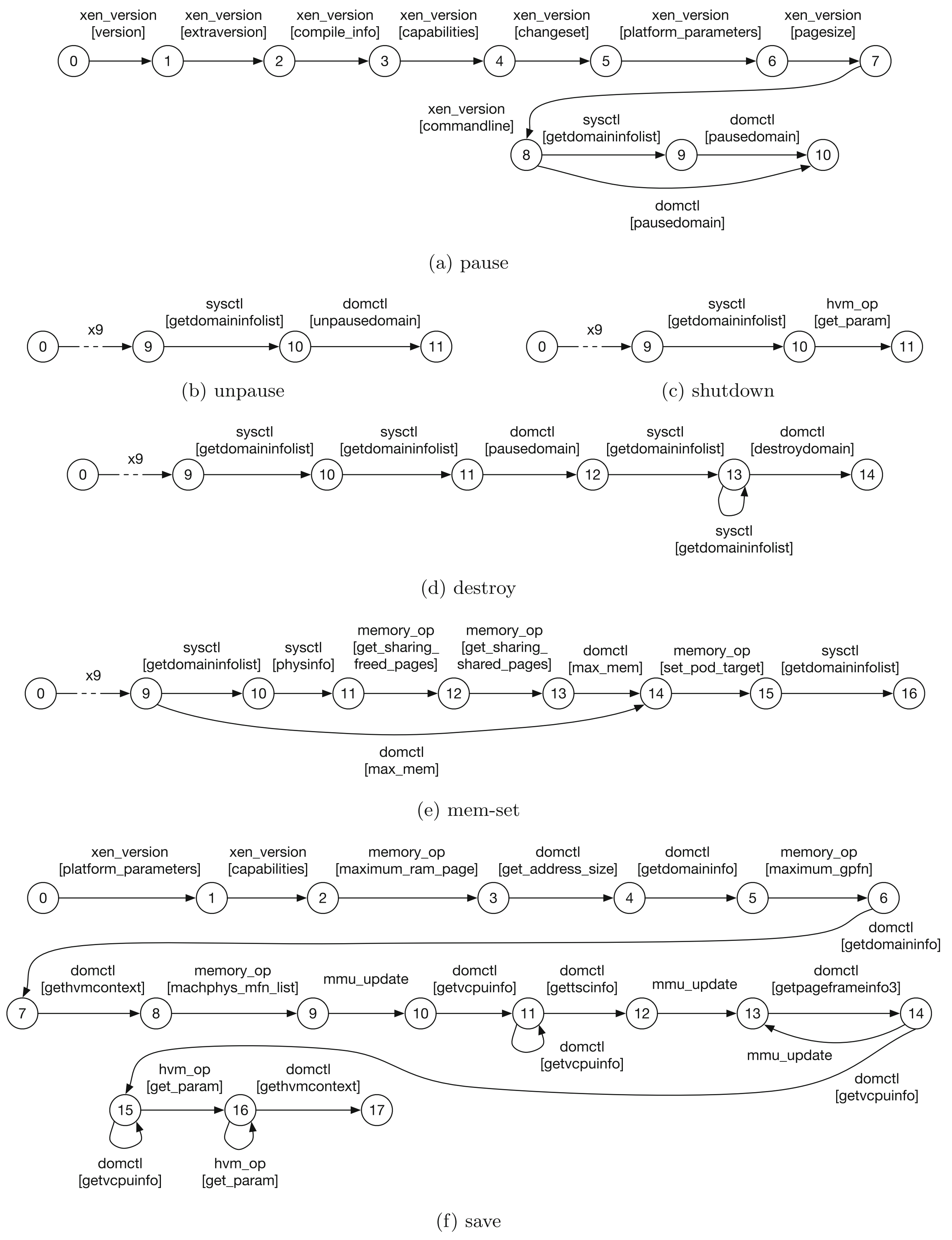

Fig. 11 Various hypercall automata

Table 2 The summary of the hypercalls used in Fig. 11

Hypercall	Description of operations
xen_version	version and extraversion (get the version number), compile_info (get compile information), capabilities (get supported interfaces), changeset (get the changeset), platform_parameters (get platform-specific parameters), pagesize (get the page size), and commandline (get command-line options)
sysctl	getdomaininfolist (get information on VMs) and physinfo (get information on virtual hardware)
domctl	pausedomain (stop the execution of a VM), unpausedomain (restart a paused VM), destroydomain (terminate a VM), max_mem (set the maximum memory size), get_address_size (get the address size), getdomaininfo (get information on a VM), gethvmcontext (get information on a fully virtualized VM), getvcpuinfo (get the state of virtual CPUs), gettscinfo (get timestamp counters), and getpageframeinfo3 (get page types)
hvm_op	get_param (get the parameters used for a fully virtualized VM)
memory_op	get_sharing_freed_pages (get the number of pages released by memory sharing), get_sharing_shared_pages (get the number of shared pages), set_pod_target (set the number of pages used for population-on-demand), maximum_ram_page (get the maximum number of pages), maximum_gpfn (get the maximum guest PFN), and machphys_mfn_list (get the list of MFNs)
mmu_update	map the memory of a VM

Figure 11b shows the hypercall automaton of the unpause command, which restarts executing a stopped VM. The first eight xen_version hypercalls and the following one sysctl hypercall are folded because they are the same as those used for the pause command. After issuing these hypercalls, the command issues the sysctl [getdomaininfolist] hypercall. This hypercall returns a VM type of full virtualization or para-virtualization. If the VM is fully virtualized, the command resumes the device model of the VM. Finally, the command issues the domctl [unpausedomain] hypercall and wakes up the VM.

Figure 11c shows the hypercall automaton for the shutdown command, which sends a shutdown signal to a VM. The first ten hypercalls are the same as those in the unpause command. This command executes the hvm_op [get_param] hypercall and obtains callback IRQ used for a para-virtualized interrupt mechanism. The parameter obtained by the get_param operation can be also specified in the hypercall automaton, but the current implementation does not support operation parameters. If the VM can accept a shutdown signal, the hypervisor writes a shutdown request to XenStore running in Dom0. Since XenStore can be accessed without issuing any hypercalls, the hypervisor automaton does not include any transition for that. This may result in the same hypercall automaton for several commands. For example, UVBond cannot distinguish hypercall sequences for the shutdown and reboot commands. The difference between these two commands is only the value written to XenStore. We can extend hypercall automata so as to include access to XenStore because UVBond can securely monitor the communication with XenStore in the hypervisor.

Figure 11d shows the hypercall automaton for the destroy command, which terminates a VM without shutdown. This command issues two sysctl [getdomaininfolist] hypercalls. The first one is used to check the existence of the specified VM and the second one is used to obtain a VM type. Then, the command stops the VM using the sysctl [pausedomain] hypercall. After that, it issues several sysctl [getdomaininfolist] hypercalls, depending on the number of virtual devices of the VM. Finally, the command destroys the VM using the domctl [destroydomain] hypercall.

Figure 11e shows the hypercall automaton for the memset command, which changes the memory size of a VM. This hypercall automaton accommodates two hypercall sequences. When this command is executed to Dom0 for the first time after the boot, it issues the sysctl [getdomaininfolist] hypercall to obtain the current and maximum memory sizes and write them to XenStore. After that, it issues the sysctl [physinfo] hypercall and two memory_op hypercalls to confirm that physical information of the VM can be obtained. These hypercalls are not issued when the command is executed to Dom0 later or to regular VMs. Then, the command issues the domctl [max_mem] hypercall and changes the memory size of the VM.

Figure 11f shows part of the hypercall automaton for the save command, which saves the state of a VM to a disk. The hypercall automaton in this figure is used for a helper program invoked by the command and supports only fully virtualized VMs. The command issues several hypercalls, invokes the helper, and then issues hypercalls for VM destruction. The last step is the same as the destroy command. The helper issues three memory_op hypercalls to obtain the memory size of the VM, the size of physical memory actually allocated to the VM, and the list of physical memory pages. Then, it saves memory contents with page types obtained by issuing many mmu_update and domctl [getpageframeinfo3] hypercalls. The state of CPUs is saved by the same number of domctl [getvcpuinfo] hypercalls as virtual CPUs. In addition, timestamp counters are saved by issuing the domctl [gettscinfo] hypercall. The state specific to a fully virtualized VM is saved by the hvm_op [get_param] and domctl [gethvmcontext] hypercalls.

Security evaluation

We examined that a VM could be booted with a virtual disk only when we used the corresponding disk encryption key. When we registered a correct key to the hypervisor, the VM could be booted normally. However, using an incorrect key, the VM could not read the boot loader from its disk because disk data was not decrypted correctly. This means that malicious cloud operators cannot boot users' VMs.

After a VM is booted correctly, we examined that UVBond could detect the execution of illegal commands by using VM descriptors and hypercall automata. We used six management commands of the xl tool: pause, unpause, mem-set, shutdown, destroy, and save. We executed these commands to a target VM with the correct VM descriptor and hypercall automata. As a result, we confirmed that these commands could be normally executed. As illegal execution, we executed these commands to a different VM that did not correspond to the specified VM descriptor. In addition, we executed a different command that did not match the specified hypercall automaton to the correct VM. In either case, we could detect the illegal execution of management commands.

Next, we examined that a VM could be resumed only with a correct disk encryption key. When we registered a correct key to the hypervisor, the VM could be restarted normally. However, using an incorrect key, the resumed VM stopped soon because its CPU state could not be restored correctly. This means that malicious cloud operators cannot resume users' VMs with a malicious disk and its encryption key.

Similarly, we examined that VM migration could succeed only with a transferred disk encryption key. When the source host transferred a correct key to the destination, a migrated VM was restarted correctly. However, when we discarded the transferred key and registered an incorrect key to the hypervisor at the destination host, we could not access a migrated VM due to the same reason as the failure on VM resumption.

In addition, we examined that a VM could be introspected only by its user. For this experiment, we have developed a simple command for introspecting the kernel version stored in VM's memory using the libxc library in Xen. Specifically, this VMI command translates the virtual address of the linux_banner variable in the kernel into a physical address and obtains the string of the kernel version. It first checks that the specified VM is fully virtualized using the domctl [getdomaininfo] hypercall. If so, it obtains the value of the CR3 register, which points to the page directory, using the domctl [gethvmcontext_partial] hypercall. Then the command walks the page tables by mapping VM's memory using multiple mmu_update hypercalls. Finally, it accesses the memory specified by the obtained physical address. For this command, we have also created a hypercall automaton. When we specified both the correct VM descriptor and hypercall automaton, this command could show the kernel version. With either an incorrect VM descriptor or an incorrect hypercall automaton, however, it failed to be executed.

We also examined that the virtual serial console of a VM could be used by its user. The console command in the xl tool does not use an identifiable sequence of hypercalls because it just accesses the virtual serial device in the device emulator running in Dom0. Fortunately, SCCrypt [18] registers a key for encrypting a virtual serial console to the hypervisor using a new hypercall. Using this hypercall, we could create a hypercall automaton for the management command used for SCCrypt. Since it was not easy to port SCCrypt, we have added only the hypercall for SCCrypt to the hypervisor. When we specified both the correct VM descriptor and hypercall automaton, we could register an encryption key for SCCrypt successfully. However, the hypercall denied that registration if we used either an incorrect VM descriptor or an incorrect hypercall automaton.

Performance of hypercalls

To clarify the overhead per hypercall execution, we have developed two simple commands executing only one hypercall, domctl [pausedomain] or domctl [unpausedomain]. For these commands, we created hypercall automata, each of which includes only two states and one transition. Then, we executed these commands in Dom0 and measured the execution time. Figure 12 shows the execution time of the commands in UVBond and vanilla Xen. For both commands, the overhead was about

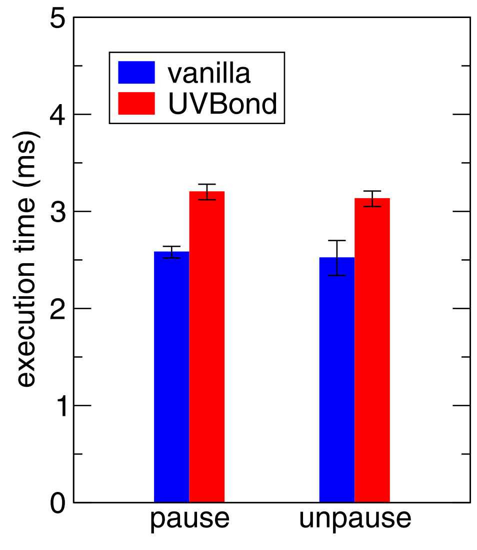

Fig. 12 The execution time of only one hypercall

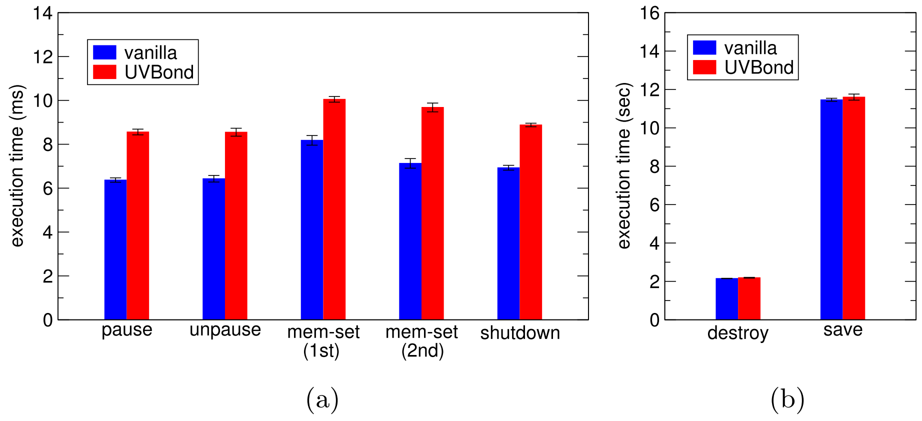

Fig. 13 The execution time of management commands (local). **a** Short-time. **b** Long-time

0.6 ms. This mainly came from the registration of a hypercall automaton, including encryption and encoding. The registered hypercall automaton was checked when a hypercall was issued, but the execution time of a hypercall was almost not affected.

Performance of management commands

To examine the execution overhead of real commands, we executed the above six management commands of the xl tool. For the mem-set command, we measured the time of the first and second execution because the hypercall sequences for them were different. Figure 13 shows the execution time when we executed management commands by invoking the xl tool from the management server in Dom0. For short-time commands, the overhead of UVBond was 1.9–2.6 ms and became larger than that for the two simple commands used in the previous subsection. Since the registration of a hypercall automaton was done only once for each command, the root cause of this overhead was that the size of a hypercall automaton became larger and the number of issued hypercalls increased. For long-time commands, the overhead of UVBond was negligible because the registration time of a hypercall automaton and the execution time of hypercalls were relatively short, compared with the total execution time of the commands.

To examine the overhead when remote users executed management commands to their VMs, we remotely executed the above six management commands. In this experiment, a remote client sent a management command with a VM descriptor and a hypercall automaton to the management server. Figure 14 shows the remote execution time. The execution time increased by 37–40 ms in any commands due to network communication. The overhead of UVBond was 1.6 ms at maximum. Compared with the total execution time of the commands, this was relatively small. The root cause of this overhead was the same as that of the above local command execution.

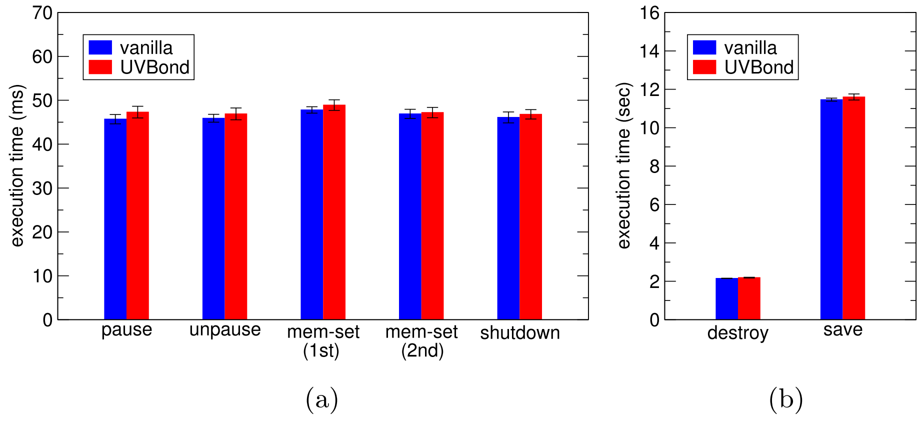

Fig. 14 The execution time of management commands (remote). **a** Short-time. **b** Long-time

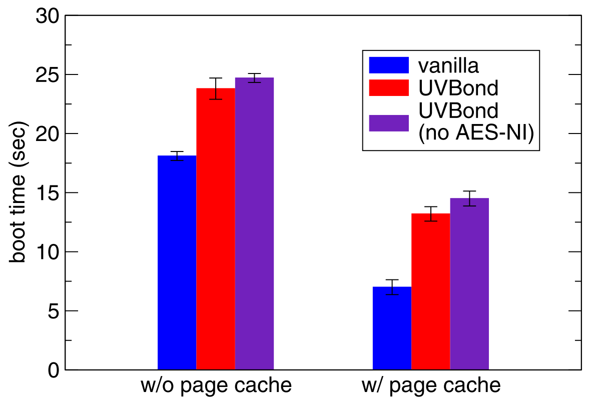

Fig. 15 The boot time of a VM

Performance of VM boot

We examined the boot time of a VM to confirm the overhead of UVBond. For comparison, we used UVBond with AES-NI disabled. We measured the time from when we executed the create command for creating a VM until the system was booted in the VM. Since the page cache in Dom0 affected the disk performance of the VM largely, we measured the time both when we cleared the page cache before booting the VM and when we booted the VM again with the page cache kept.

As shown in Fig. 15, the boot time in UVBond was 6 seconds longer than that in vanilla Xen without depending on the page cache. This is because the booting process of the operating system conflicted with the decryption of the virtual disk. Figure 16 shows the CPU utilization of Dom0 and the VM during the boot. The VM read 258 MB of data from the disk, while it wrote only 5 MB of data. Since read data was decrypted by the hypercall issued by Dom0, the CPU utilization of Dom0 increased by 16 percent point on average. Compared with when we disabled AES-NI, it was shown that AES-NI made the boot only 1 second faster.

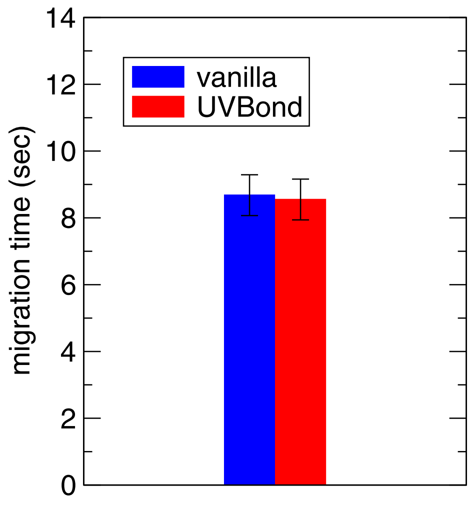

Fig. 17 The resumption time of a VM

Performance of VM resumption and migration

We examined the performance of resuming a VM suspended by the save command. We measured the time from when we executed the restore command until it was completed. As shown in Fig. 17, the resumption time in UVBond was almost the same as that in the traditional system. UVBond was slightly faster than the traditional system, but the variance was relatively large and therefore the difference was within the margin of error. This means that the overhead of the decryption of the CPU state was negligible, compared with a long resumption time.

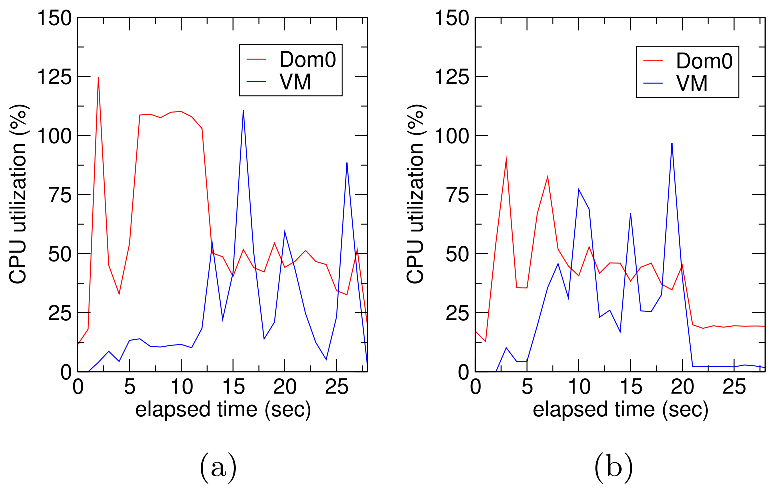

Fig. 16 The CPU utilization during VM boot. **a** UVBond. **b** vanilla

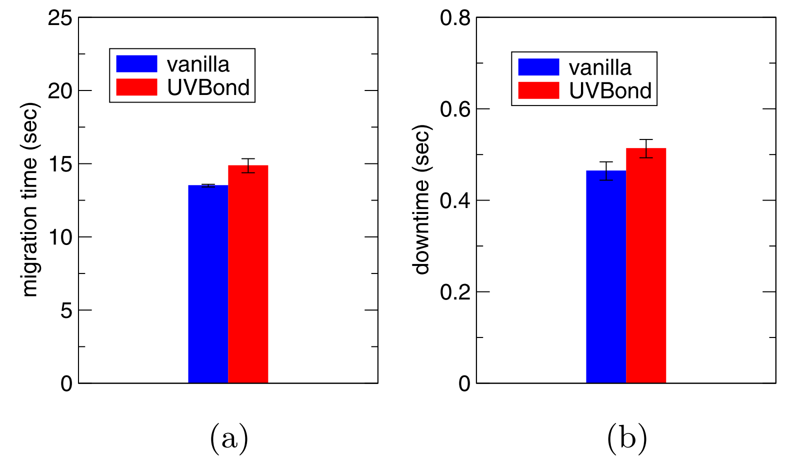

Fig. 18 Migration performance. **a** Migration time. **b** Downtime

Next, we examined the performance of migrating a VM using UVBond. We measured the migration time, which was the time from when we executed the migrate command until it was completed. As shown in Fig. 18a, the migration time in UVBond was 1.4 seconds longer than vanilla Xen because UVBond had to transfer the disk encryption key and the session key and encrypt and decrypt the CPU state. The encryption of the CPU state needed 0.2 seconds. Also, we measured the downtime, which was the time from when the VM was stopped at the source host until it was resumed at the destination host. As shown in Fig. 18b, the downtime in UVBond became only 50 ms longer.

Performance of disk I/O

We examined the disk I/O performance using the fio benchmark [46]. In addition to UVBond with or without AES-NI and vanilla Xen, we used the system using Linux dm-crypt [47] inside a VM. dm-crypt encrypts and decrypts disks in the operating system of a VM. For dm-crypt, we used AES-ECB as the encryption method, which was the same as that of UVBond. We measured read and write performance of sequential and random access.

Figure 19 shows the throughput and the latency when a VM used a virtual disk on a local disk. Compared with vanilla Xen, the throughput in UVBond degraded only by 3 to 10% and the latency increased only by 0.8 to 1.3 ms thanks to AES-NI. Unlike the experiment of VM boot, the performance did not degrade largely because the CPU utilization was not so high.

Without AES-NI, the throughput degradation was 45% and the latency increase was 6.5 ms at maximum. The throughput in UVBond was comparable to or even better than that in dm-crypt, while the latency was slightly longer.

Figure 20 shows the throughput and the latency when a VM used a virtual disk on NFS. This configuration is often used when a VM is migrated. The performance

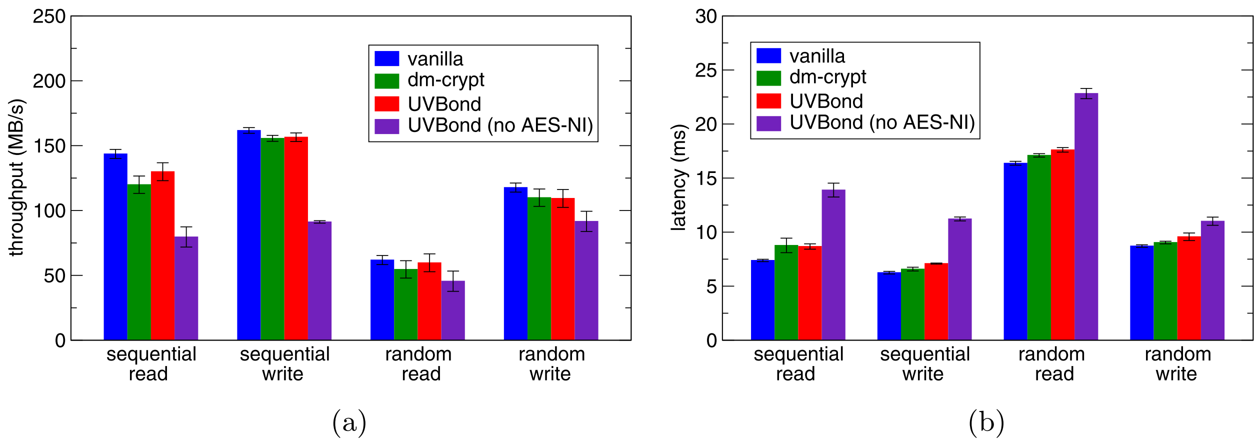

Fig. 19 The performance of file access (local disk). **a** Throughput. **b** Latency

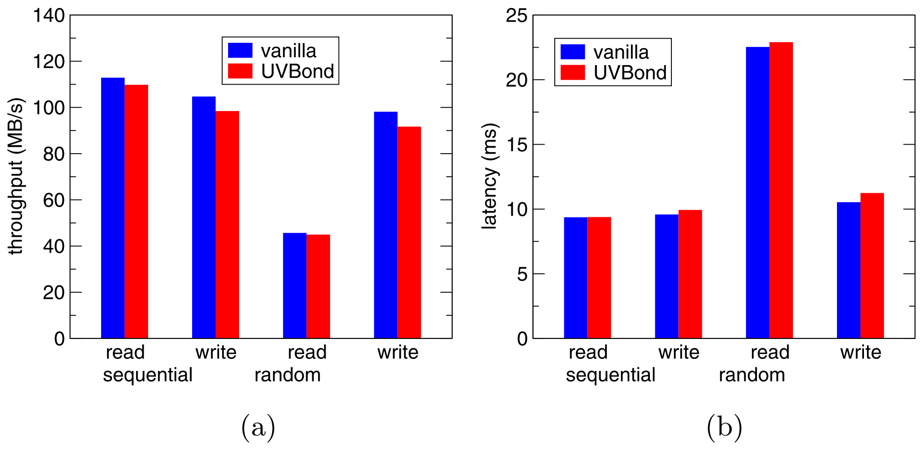

Fig. 20 The performance of file access (NFS). **a** Throughput. **b** Latency

degradation in UVBond was similar to that when using a local disk. The throughput degraded by 1.6 to 6.6%, while the latency increased by 0 to 0.7 ms.

Conclusion

This paper proposed UVBond for providing strong user binding to their VMs. UVBond enables only a user to boot his VM by decrypting its encrypted disk inside the trusted hypervisor. Then it issues a VM descriptor for securely identifying that VM. To bridge the semantic gap between high-level management commands and low-level hypercalls, UVBond uses hypercall automata and accepts only the sequences of hypercalls issued by user's commands. Using UVBond, untrusted cloud operators cannot execute arbitrary commands to user's VMs or redirect user's commands to their malicious VMs. We have implemented UVBond in Xen and created various hypercall automata. Then we confirmed that a VM descriptor and hypercall automata prevented insider attacks and that the overhead was negligible in remote VM management.

One of our future work is to extend hypercall automata so as to include access to XenStore in Xen. Since management commands can access XenStore without hypercalls, UVBond needs to restrict accepted commands more strictly. It is not difficult to incorporate XenStore access into hypercall automata because UVBond already monitors it. In addition, we need to include the parameters of hypercalls in hypercall automata. In the current implementation, generic hypercalls such as domctl and sysctl consider only the first parameter used for specifying a subdivided operation. To prevent the Iago attack [48], we need to extend our implementation so as to consider more parameters. Also, it is necessary for one hypercall automaton to accept hypercall sequences issued by a group of multiple processes. This is because several management commands invoke external programs.

Another direction is to apply UVBond to large cloud management systems such as OpenStack. To support UVBond in such systems, we have to extract used management commands and create their hypercall automata. If there are management commands executed across multiple compute nodes, we need to extend hypercall automata so as to accept a sequence of hypercalls issued by multiple hypervisors. Also, we need to modify the Web interface and API so as to send VM descriptors and hypercall automata as well as commands. In addition, it is interesting to apply UVBond to not only VMs but also containers. We would need to use system-call automata instead of hypercall automata because containers are managed with system calls.

Abbreviations

IaaS: Infrastructure-as-a-service; IDS: Intrusion detection systems; PIO: Programmable input and output; SD: Service domain; SMM: System management mode; SSC: Self-service cloud; TCB: Trusted computing base; VM: Virtual machines; VMI: Virtual machine introspection

Acknowledgements

Not applicable.

Authors' contributions

KI implemented the system and conducted experiments. KK conducted the research, designed the system, and wrote the manuscript. Both authors read and approved the manuscript.

Funding

The research results have been achieved by the "Resilient Edge Cloud Designed Network (19304)," the Commissioned Research of National Institute of Information and Communications Technology (NICT), Japan.

Availability of data and materials

The dataset supporting the conclusions of this article is available in the OSF repository at https://osf.io/dx9ve/.

Competing interests

The authors declare that they have no competing interests.

Received: 27 April 2019 Accepted: 8 January 2020
Published online: 17 January 2020

References

1. Santos N, Gummadi KP, Rodrigues R (2009) Towards Trusted Cloud Computing. In: Proceedings of Workshop on Hot Topics in Cloud Computing. USENIX Association, CA, USA
2. Li C, Raghunathan A, Jha NK (2010) Secure Virtual Machine Execution under an Untrusted Management OS. In: Proceedings of IEEE International Conference on Cloud Computing. pp 172–179. https://doi.org/10.1109/cloud.2010.29
3. Zhang F, Chen J, Chen H, Zang B (2011) CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In: Proceedings of ACM Symposium on Operating Systems Principles. pp 203–216. https://doi.org/10.1145/2043556.2043576
4. Butt S, Lagar-Cavilla HA, Srivastava A, Ganapathy V (2012) Self-service Cloud Computing. In: Proceedings of ACM Conference on Computer and Communications Security. pp 253–264. https://doi.org/10.1145/2382196.2382226
5. Li M, Zang W, Bai K, Yu M, Liu P (2013) MyCloud: Supporting User-configured Privacy Protection in Cloud Computing. In: Proceedings of the 29th Annual Computer Security Applications Conference. pp 59–68. https://doi.org/10.1145/2523649.2523680
6. Kourai K, Juda K (2016) Secure Offloading of Legacy IDSes Using Remote VM Introspection in Semi-trusted Clouds. In: Proceedings of IEEE International Conference on Cloud Computing. pp 43–50. https://doi.org/10.1109/cloud.2016.0016
7. Miyama S, Kourai K (2017) Secure IDS Offloading with Nested Virtualization and Deep VM Introspection. In: Proceedings of European Symposium on Research in Computer Security. pp 305–323. https://doi.org/10.1007/978-3-319-66399-9_17
8. Futagami S, Unoki T, Kourai K (2018) Secure Out-of-band Remote Management of Virtual Machines with Transparent Passthrough. In: Proceedings of the 2018 Annual Computer Security Applications Conference. pp 430–440. https://doi.org/10.1145/3274694.3274749
9. PricewaterhouseCoopers (2014) US Cybercrime: Rising Risks. Reduced Readiness: Key Findings from the 2014 US State of Cybercrime Survey. http://www.pwc.com/us/en/increasing-iteffectiveness/publications/assets/2014-us-state-of-cybercrime.pdf. Accessed 11 Jan 2020
10. CyberArk Software (2009) Trust, Security & Passwords Survey. https://www.cyberark.com/press/global-security-survey-findsinsider-snooping-rise/. Accessed 11 Jan 2020
11. Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A (2003) Xen and the Art of Virtualization. In: Proceedings of Symposium on Operating Systems Principles. pp 164–177. https://doi.org/10.1145/1165389.945462
12. Inokuchi K, Kourai K (2018) UVBond: Strong User Binding to VMs for Secure Remote Management in Semi-Trusted Clouds. In: Proceedings of IEEE/ACM International Conference on Utility and Cloud Computing. pp 213–222. https://doi.org/10.1109/UCC.2018.00030
13. TechSpot News (2010) Google Fired Employees for Breaching User Privacy. http://www.techspot.com/news/40280-google-fired-employees-for-breaching-user-privacy.html. Accessed 27 Apr 2019
14. Oracle Corporation (2019) Oracle Database 2 Day DBA. https://docs.oracle.com/en/database/oracle/oracle-database/19/admqs/administering-user-accounts-and-security.html. Accessed 22 Nov 2019
15. IBM Corporation (2018) IBM Domino 10.0.1 Documentation. https://www.ibm.com/support/knowledgecenter/SSKTMJ_10.0.1/admin/conf_restrictingadministratoraccess_t.html. Accessed 22 Nov 2019
16. Garfinkel T, Rosenblum M (2003) A Virtual Machine Introspection Based Architecture for Intrusion Detection. In: Proceedings of Network and Distributed Systems Security Symposium. Internet Society, VA, USA. pp 191–206
17. Egawa T, Nishimura N, Kourai K (2012) Dependable and Secure Remote Management in IaaS Clouds. In: Proceedings of International Conference on Cloud Computing Technology and Science. pp 411–418. https://doi.org/10.1109/cloudcom.2012.6427597
18. Kourai K, Kajiwara T (2015) Secure Out-of-band Remote Management Using Encrypted Virtual Serial Consoles in IaaS Clouds. In: Proceedings of International Conference on Trust, Security and Privacy in Computing and Communications. pp 443–450. https://doi.org/10.1109/Trustcom.2015.405
19. The OpenStack Project OpenStack Open Source Cloud Computing Software. https://www.openstack.org/. Accessed 27 Apr 2019
20. Butt S, Ganapathy V, Srivastava A (2014) On the Control Plane of a Self-service Cloud Platform. In: Proceedings of Symposium on Cloud Computing. https://doi.org/10.1145/2670979.2670989
21. Murray D, Milos G, Hand S (2008) Improving Xen Security through Disaggregation. In: Proceeedings of the 4th ACM International Conference on Virtual Execution Environments. pp 151–160. https://doi.org/10.1145/1346256.1346278
22. Colp P, Nanavati M, Zhu J, Aiello W, Coker G, Deegan T, Loscocco P, Warfield A (2011) Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor. In: Proceeedings of the 23rd ACM Symposium on Operating Systems Principles. pp 189–202. https://doi.org/10.1145/2043556.2043575
23. Li C, Raghunathan A, Jha NK (2012) A Trusted Virtual Machine in an Untrusted Management Environment. IEEE Trans Serv Comput 5(4):472–483. https://doi.org/10.1109/TSC.2011.30
24. Tadokoro H, Kourai K, Chiba S (2012) Preventing Information Leakage from Virtual Machines' Memory in IaaS Clouds. IPSJ Online Trans 5:156–166. https://doi.org/10.2197/ipsjtrans.5.156
25. Bleikertz S, Vogel C, Groß T, Mödersheim S (2015) Proactive Security Analysis of Changes in Virtualized Infrastructures. In: Proceedings of the 31st Annual Computer Security Applications Conference. pp 51–60. https://doi.org/10.1145/2818000.2818034
26. OpenStack Project OpenStack Docs: Welcome to Congress. https://docs.openstack.org/congress/. Accessed 22 Nov 2019
27. Li S, Koh J, Nieh J (2019) Protecting Cloud Virtual Machines from Hypervisor and Host Operating System Exploits. In: Proceedings of the 28th USENIX Security Symposium. USENIX Association, CA, USA. pp 1357–1374
28. Zhu M, Tu B, Wei W, Meng D (2017) HA-VMSI: A Lightweight Virtual Machine Isolation Approach with Commodity Hardware for ARM. In: Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. pp 242–256. https://doi.org/10.1145/3050748.3050767
29. Wu Y, Liu Y, Liu R, Chen H, Zang B, Guan H (2018) Comprehensive VM Protection Against Untrusted Hypervisor Through Retrofitted AMD Memory Encryption. In: Proceedings of 2018 IEEE International Symposium on High Performance Computer Architecture. pp 441–453. https://doi.org/10.1109/hpca.2018.00045
30. Baumann A, Peinado M, Hunt G (2014) Shielding Applications from an Untrusted Cloud with Haven. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. pp 267–283. https://doi.org/10.1145/2799647
31. Ben-Yehuda M, Day MD, Dubitzky Z, Factor M, Har'El N, Gordon A, Liguori A, Wasserman O, Yassour B-A (2010) The Turtles Project: Design and Implementation of Nested Virtualization. In: Proceedings of USENIX Symposium on Operating Systems Design and Implementation. USENIX Association, CA, USA. pp 423–436
32. Shinagawa T, Eiraku H, Tanimoto K, Omote K, Hasegawa S, Horie T, Hirano M, Kourai K, Oyama Y, Kawai E, Kono K, Chiba S, Shinjo Y, Kato K (2009) BitVisor: A Thin Hypervisor for Enforcing I/O Device Security. In: Proceedings of International Conference on Virtual Execution Environments. pp 121–130. https://doi.org/10.1145/1508293.1508311
33. Hofmeyr S, Forrest S, Somayaji A (1998) Intrusion Detection Using Sequences of System Calls. Comput Secur 6:151–180. IOS Press, https://doi.org/10.3233/JCS-980109
34. Rutkowska J, Wojtczuk R (2008) Preventing and Detecting Xen Hypervisor Subversions. Black Hat, USA
35. Wang J, Stavrou A, Ghosh A (2010) HyperCheck: A Hardware-assisted Integrity Monitor. In: Proceedings of International Symposium on Recent Advances in Intrusion Detection. pp 158–177. https://doi.org/10.1007/978-3-642-15512-3_9
36. Azab A, Ning P, Wang Z, Jiang X, Zhang X, Skalsky N (2010) HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity. In: Proceedings of ACM Conference on Computer and Communications Security. pp 38–49. https://doi.org/10.1145/1866307.1866313
37. McCune J, Parno B, Perrig A, Reiter M, Isozaki H (2008) Flicker: An Execution Infrastructure for TCB Minimization. In: Proceedings of European Conference on Computer Systems. pp 315–328. https://doi.org/10.1145/1352592.1352625
38. Deng L, Liu P, Xu J, Chen P, Zeng Q (2017) Dancing with Wolves: Towards Practical Event-driven VMM Monitoring. In: Proceedings of the 13th ACM

SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. pp 83–96. https://doi.org/10.1145/3050748.3050750
39. Sgandurra D, Lupu E (2016) Evolution of Attacks, Threat Models, and Solutions for Virtualized Systems. ACM Comput Surv 48(3). https://doi.org/10.1145/2856126
40. Patil R, Modi C (2019) An Exhaustive Survey on Security Concerns and Solutions at Different Components of Virtualization. ACM Comput Surv 52(1). https://doi.org/10.1145/3287306
41. Huang W, Ganjali A, Kim B, Oh S, Lie D (2015) The State of Public Infrastructure-as-a-Service Cloud Security. ACM Computing Surv 47(4). https://doi.org/10.1145/2767181
42. wolfSSL Inc. wolfSSL Embedded SSL/TLS Library. https://www.wolfssl.com/. Accessed 27 Apr 2019
43. Red Hat Inc. Kernel Based Virtual Machine. http://www.linux-kvm.org/. Accessed 22 Nov 2019
44. Merkle R (1980) Protocols for Public Key Cryptosystems. In: Proceedings of IEEE Symposium on Research in Security and Privacy. https://doi.org/10.1109/sp.1980.10006
45. Jones S, Arpaci-Dusseau A, Arpaci-Dusseau R (2006) Antfarm: Tracking Processes in a Virtual Machine Environment. In: Proceedings of USENIX Annual Technical Conference. USENIX Association, CA, USA
46. Axboe J fio: Flexible I/O Tester. https://github.com/axboe/fio. Accessed 27 Apr 2019
47. Brož M dm-crypt: Linux Kernel Device-mapper Crypto Target. https://gitlab.com/cryptsetup/cryptsetup/wikis/DMCrypt. Accessed 27 Apr 2019
48. Checkoway S, Shacham H (2013) Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In: Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems. pp 253–264. https://doi.org/10.1145/2451116.2451145

Publisher's Note