A Trusted Virtual Machine in an Untrusted Management Environment

Chunxiao Li, *Student Member*, *IEEE*, Anand Raghunathan, *Fellow*, *IEEE*, and Niraj K. Jha, *Fellow*, *IEEE*

Abstract—Virtualization is a rapidly evolving technology that can be used to provide a range of benefits to computing systems, including improved resource utilization, software portability, and reliability. Virtualization also has the potential to enhance security by providing isolated execution environments for different applications that require different levels of security. For security-critical applications, it is highly desirable to have a small trusted computing base (TCB), since it minimizes the surface of attacks that could jeopardize the security of the entire system. In traditional virtualization architectures, the TCB for an application includes not only the hardware and the virtual machine monitor (VMM), but also the whole management operating system (OS) that contains the device drivers and virtual machine (VM) management functionality. For many applications, it is not acceptable to trust this management OS, due to its large code base and abundance of vulnerabilities. For example, consider the "computing-as-a-service" scenario where remote users execute a guest OS and applications inside a VM on a remote computing platform. It would be preferable for many users to utilize such a computing service without being forced to trust the management OS on the remote platform. In this paper, we address the problem of providing a secure execution environment on a virtualized computing platform under the assumption of an untrusted management OS. We propose a secure virtualization architecture that provides a secure runtime environment, network interface, and secondary storage for a guest VM. The proposed architecture significantly reduces the TCB of security-critical guest VMs, leading to improved security in an untrusted management environment. We have implemented a prototype of the proposed approach using the Xen virtualization system, and demonstrated how it can be used to facilitate secure remote computing services. We evaluate the performance penalties incurred by the proposed architecture, and demonstrate that the penalties are minimal.

Index Terms—Virtual machine, trusted computing base, memory protection, cloud computing, computing-as-a-service

1 Introduction

Virtualization is an emerging technology that abstracts the physical resources of a computing platform into many separate logical resources or computing environments. Each of the separated virtual computing environments is called a virtual machine (VM). The virtualization environment allows users to create, copy, save, read, modify, share, migrate and roll back the execution state of VMs [1], which trims administrative overhead and makes system administration and management easier. However, the easier management also gives rise to security concerns. If the management environment is compromised, all the VMs can be easily copied and modified. Several attacks that are only available through hardware access in traditional computing systems, e.g., the cold-boot attack [2], can be implemented in software on a virtualized computing platform, as shown later. Furthermore, attacks from the management environment (e.g., due to exploits of its vulnerabilities) can easily bypass the security mechanisms present in guest VMs due to the higher privilege level of the management OS.

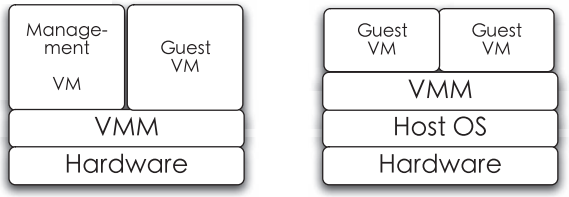
- *C. Li and N.K. Jha are with the Department of Electrical Engineering, Princeton University, E-Quad. Olden St., Princeton, NJ 08544. E-mail: chunxiao@princeton.edu, jha@ee.princeton.edu.*
- *A. Raghunathan is with the School of Electrical and Computer Engineering, Purdue University, 465 Northwestern Avenue, West Lafayette, IN 47907. E-mail: raghunathan@purdue.edu.*

Manuscript received 18 Nov. 2010; revised 6 Apr. 2011; accepted 4 May 2011; published online 14 June 2011.
For information on obtaining reprints of this article, please send e-mail to: tsc@computer.org, and reference IEEECS Log Number TSCSI-2010-11-0140. Digital Object Identifier no. 10.1109/TSC.2011.30.

There are two basic types of virtualization architectures, as shown in Fig. 1. In Type I virtualization architectures, the virtual machine monitor (VMM) is just above the hardware and intercepts all the communications between the VMs and the hardware. There is a *management VM* on top of the VMM, which manages other guest VMs, and is responsible for most communications with the hardware. A popular instance of this type of virtualization architecture is the Xen system [3]. In Type II virtualization architectures, such as VMware Player [4], the VMM runs as an application within the host operating system (OS). The host OS is responsible for providing I/O drivers and managing the guest VMs.

The VM execution environment includes the VMM and the management VM in the Type I architecture, and the VMM and host OS in the Type II architecture. However, from a security point of view, both architectures raise the question "How can the VM trust its execution environment, which may be either malicious, or susceptible to vulnerability exploits?" We elucidate this concern by describing two concrete application scenarios where it arises.

- Computing-as-a-service and cloud computing have gained increasing popularity in recent years. Services like Amazon.com's Elastic Computing Cloud (EC2) [5] use virtualization technology to provide clients with scalable computing capacities at low cost. An image containing the applications, libraries, data, and associated configuration settings is built as a VM and executed on the service provider's data centers. This is an attractive proposition for clients



(a) Type I (b) Type II

Fig. 1. Types of virtualization architectures.

who do not wish to incur the overheads of maintaining and operating their own computing facilities, and only pay for the computing resources that they actually consume. The problem is how can they trust the VM execution environment and be sure that the private data stored there are safe enough.

- The ubiquitous computing community has proposed the concept of storing the "working environment" of a user on a portable storage device so that any computer available to the user can be "personalized" to provide the exact same look and feel as the user's personal computer (e.g., the SoulPad system developed at IBM [6]). Virtualization can enable this concept by storing an OS image together with applications and data as a VM on a portable storage device. The user does not have to bring a computer everywhere, instead, his VM can be imported to the virtualization environment provided by his collaborators or a third-party computing company. In such a scenario, how can the user be assured of the privacy of data in his VM if he wants to execute it on an untrusted computer?

Generally, in order to ensure the trustworthiness of a software system, we first determine the trusted computing base (TCB) of that system. Then, we check the integrity of its TCB and decide whether to trust it. In Type I virtualization architectures, the TCB of a VM is the hardware, VMM, and the management VM. In Type II virtualization architectures, the TCB of a VM is the hardware, VMM, and the host OS. In a virtualization-based architecture, while the hardware is inevitably in the TCB and the VMM has a relatively small code base and is thus easy to verify, a full-fledged OS—the OS in the management VM or the host OS—cannot be trusted because in our threat model, the VMM and the management OS are generally controlled by the same party, e.g., the cloud computing service company. The reasons we choose to trust the VMM but not trust the OS are that 1) the sizes of the source code base of a VMM and an OS are very different, 2) the known and unknown vulnerabilities and numerous potentially malicious applications running within the management OS make it less trustworthy, and 3) the administrative interface of the management OS is exposed more often to careless or even malicious administrators.

In this paper, we mainly focus on Type I virtualization architectures, and use Xen as a prototype for demonstration—actually, Amazon EC2 itself is a Xen-based infrastructure [5]. We show how the management OS can be removed from the TCB of the VM, thereby ensuring data confidentiality and integrity of a VM execution environment even under an untrusted management OS. A preliminary version of this paper was presented at [7].

2 Related Work

The relationship between virtualization and security is a paradox [8], which naturally divides the related research in this field into two groups: virtualization for security and the security of virtualization itself.

On the one hand, virtualization can be utilized to enhance security. Several research studies [9], [10], [11], [12] utilize virtualization to implement introspection from the secure domain to the target domain. Lares [10] is an architecture that enables active monitoring from an isolated secure VM. A VMM-based "out-of-the-box" semantic view reconstruction approach [9] is designed to bridge the semantic gap introduced by VM introspection. Petroni and Hicks [11] and Riley et al. [12] present virtualization-based approaches to protect kernel integrity, and VMWall [13] is a virtualization-based firewall. Terra [14] is another virtualization architecture that allows many VMs that have different security requirements to run independently without the threat of interference from each other. Overshadow [15] and SP3 [16] rely on the underlying VMM (also called hypervisor) to separate processes from untrusted guest OSs. The proposed methodology and Overshadow use similar techniques to change the views of memory pages to different parties. The differences are as follows: 1) They have different goals. Overshadow's goal is to protect the application data from the untrusted OS, while our approach is targeted at protecting guest domains from compromised management domains. 2) Overshadow focuses on memory page protection, while our approach focuses on runtime environment protection. The proposed methodology protects not only the memory pages, but also the virtual CPU (vCPU) state. Protection is provided during domain build, domain save, domain restore, domain shutdown, and domain live migration. 3) The proposed key management model deals with a three-entity scenario (VMM, DomU, and remote user) with another untrusted, but very powerful, entity (management OS), which is more complex than the case in Overshadow that uses one key for all applications.

On the other hand, the security of virtualization itself is a significant concern. In [1], security challenges in virtual environments are summarized as scaling, transience, software lifecycle, diversity, mobility, identity, and data lifetime. Virtual machine-based rootkits (VMBRs) [17] represent an approach to exploiting the virtualization layer for security attacks. sHype [18] is a virtualization security architecture that enables control of the virtual environment resources with a system-wide mandatory access control policy.

As indicated in [19], all the research studies that utilize virtualization to enhance system security are based on one assumption—that it provides stronger isolation between guest OSs than the isolation between processes provided by current OSs. The theoretical foundation of this belief is that the hypervisor layer is smaller than the traditional OS, and is thus easy to verify and has a higher potential to be vulnerability-free. However, in a management VM, normally the whole OS is included in the TCB of the virtualization system, which severely undermines the

foundation of smaller VMMs and stronger isolation. Our work attempts to solve the fundamental security challenge in virtualization posed by the fact that the TCB of a guest VM is too large. We propose a secure virtualization architecture that removes the management OS from the TCB of a VM and demonstrate how it can be deployed by prototyping it in the Xen virtualization system.

In the Terra [14] architecture, the "root secure" property that says that "even the platform administrator cannot break the basic privacy" may incorrectly suggest that it has already achieved the goal of excluding the management OS from the TCB. Actually, in this architecture, the management OS is just a simple command interface and all the administrative work, e.g., to create, save, restore, and shut down a VM, is actually implemented in the VMM, which itself is trusted in Terra's threat model. Because these functions are complex and need access to the memory of guest VMs, we believe they should be left in the management OS so that the VMM can be made as small as possible—as done by the Xen architecture. Our proposed architecture is different from Terra in that we exclude from the TCB not only the administration command interface (as Terra did) but also most of the complex administrative functionalities themselves. Also, the proposed methodology in [20] has a similar goal as ours, but it not only includes the whole management OS kernel, but also introduces a new domain, DomB, inside the TCB.

3 Motivation

In this section, after a brief introduction to the Xen architecture, we describe the potential threats to a VM in an untrusted management environment, followed by a concrete example of a successful attack that exploits the management OS. These attacks are easy to implement and difficult to defend against in existing virtualization systems, forming the motivation for our research on secure VM execution under an untrusted management OS.

3.1 The Xen Architecture

Xen is an example of a Type I virtualization architecture. The Xen hypervisor sits between the OS and the hardware. The hypervisor, OS kernel, and user applications are three software layers in a Xen virtualization system. While separation of VMs is needed for security reasons, inter-VM communication is also crucial for some functions. In the x86 architecture, privilege levels are defined as four rings from ring 0 (the highest privilege level) to ring 3 (the lowest privilege level). In the traditional architecture, the OS kernel runs in ring 0 and the user applications run in ring 3. In the Xen architecture, the hypervisor runs in ring 0, VM kernels run in ring 1, and the user applications run in ring 3. Just like the system calls, which transfer control from ring 3 to ring 0 in the traditional architecture, Xen uses hypercalls to communicate from the VM kernel layer to the hypervisor layer. The system design of Xen follows the philosophy of separation of policy and mechanisms. The Xen hypervisor implements mechanisms, but the policy is designed in the VM. The mechanism used for inter-VM communication is shared memory, which can be established through either the grant table or foreign mapping.

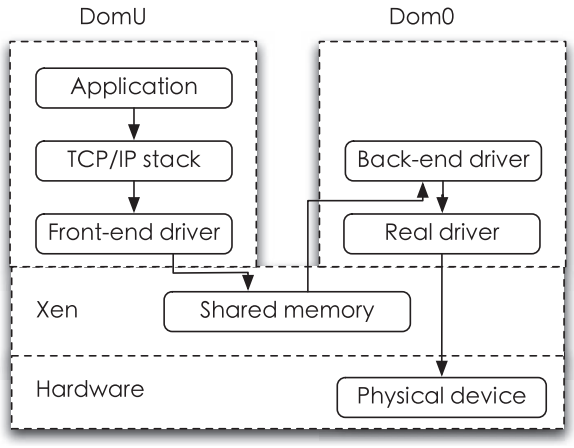


Fig. 2. The split device driver model.

For each memory page that a VM wants to share with another, a grant table entry is established. The entry includes information on which domain the permissions of memory access are granted to and what these permissions are. If another domain wants to access this memory page, it makes a hypercall and the hypervisor looks up the grant table to make a decision regarding whether sharing is allowed.

The management OS (Dom0 in the Xen architecture) can directly map memory pages from other domains into its own address space, which is called foreign mapping. This mapping can only be made by Dom0. During several management operations, such as domain building, saving, and restoring, this mapping mechanism is used. Among all the VMs above the hypervisor layer in the Xen architecture, Dom0 has the highest privilege level. Some of the hypercalls, e.g., foreign mapping, can only be made by Dom0. Since Dom0 is considered to be untrusted in the proposed mechanism, we need to clarify the role that Dom0 plays, as well as evaluate the damage that can result if Dom0 is malicious.

The most important task performed by Dom0 is to handle hardware devices. The device drivers are normally located in Dom0. In Fig. 2, a split driver model of the network card is shown. The packet in the unprivileged domain, which is called DomU, goes from the application to the front end driver in DomU. Using the shared-memory mechanisms mentioned earlier, the packet is shared with Dom0, in which the back-end driver and real device driver reside. Dom0 finally transmits the data packet to the physical device in the hardware layer. Another role played by Dom0 in the Xen architecture is the task of VM management. Dom0 is responsible for creating, copying, saving, reading, modifying, sharing, migrating, and rolling back the execution state of VMs. However, in an untrusted Dom0, these tasks must be supervised in order to ensure the integrity and confidentiality of the guest VM, which is the main objective of our research.

3.2 Protection of Xen Hypervisor from Untrusted Dom0

In the proposed methodology, the hypervisor is in the TCB and Dom0 is not, which means that even a compromised Dom0 cannot tamper with the hypervisor. This is a strong assumption, which has several requirements:

- Memory protection requirements: The runtime memory of the hypervisor cannot be compromised by Dom0. Currently, without additional protection of hardware, any direct memory access (DMA) capable device can access any arbitrary physical memory location. Dom0 has all device drivers and is able to set up DMA and have access to Xen's memory address space. Defending against DMA attacks requires hardware protection from the AMD input/output memory management unit (IOMMU) [21] or Intel VT-d [22] technology, whose controlling code should reside in the hypervisor. Researchers have shown how to launch new attacks on the hypervisor memory even when employing hardware protection techniques [23]. However, these attacks exploit vulnerabilities of the hardware, which should be fixed and do not affect our approach.
- Image protection requirements: The kernel image of the hypervisor cannot be compromised by Dom0. The Xen hypervisor kernel image is stored in the hard disk, which is controlled by Dom0. Dom0 can alter this image and reboot to load the compromised hypervisor. The solution is to utilize the trusted computing technology [24] to perform a measured and verified launch of the hypervisor. Note that the Trusted Platform Module (TPM) should be controlled by the hypervisor, not Dom0.

3.3 Security Threats to DomU from Untrusted Dom0

We first describe a scenario for the security threats described in this section. Suppose a client is running a guest VM on the remote virtualized computing platform provided by a cloud computing company. The computation in the VM is security-critical, and involves confidential data of an enterprise and/or personal sensitive information. The client needs to use the service provided by the cloud computing company, but is, however, reluctant to trust the management domain, which has full privilege to access all data in the guest VM. The untrusted management domain, i.e., Dom0 in Xen, is capable of undermining the confidentiality, integrity, and availability of a DomU, as described next.

- Confidentiality: Dom0 may access any memory page of DomU and read its contents. Also, Dom0 contains the device drivers for I/O devices, such as the network card and hard disk, which endangers the privacy of the data transmitted through the network and the data stored on the hard disk.
- Integrity: For the same reason, Dom0 may access any memory page of DomU and change its contents, as well as modify the data transmitted through the network and the data stored on the hard disk.
- Availability: Dom0 has the privilege to start and shut down the other domains and, thus, controls the availability of all guest VMs and the applications that execute within them.

3.4 A Concrete Attack Example

Encryption is an effective way to keep data secret. As long as we make the encryption keys strong enough and store them securely, the secrecy of data can be ensured. In the simplest scenario, we encrypt some plaintext in a VM to ciphertext and place the ciphertext on the hard disk. Hence, we are not worried even if the hard disk is lost or stolen. The keys are always located in system memory. During the normal system execution, we do not worry much about memory safety. Under cold-boot attacks [2], attackers, who can physically extract the memory chip from the computer, can extract its contents. In a virtualized computing environment, however, the memory contents are saved to an image file by a simple "domain save" command in Dom0. This file can be exploited to find the keys used for data encryption without physical access to the system.

In our implementation, the user in DomU creates an encrypted disk using the `dm-crypt` tools in Linux. While DomU is executing, the administrator in Dom0 saves the state of the VM to a memory image file. The file should contain the keys used by the encryption algorithm used in DomU. Although we do not know the exact location of these keys, there are well-known techniques that can be used to narrow down the possible locations. We used the algorithm described in [2] to analyze a VM image file of size 128 MB and were able to successfully identify all cryptographic keys in less than 1 second on a mainstream desktop, as described in Section 5. Once cryptographic keys are leaked, all the ciphertext in the encrypted disk can be decrypted, unknown to DomU.

The original cold-boot attack requires physical access to the memory chip before the contents of the chip decay. These attacks are effective, but difficult to implement because of the physical access requirement and time restriction. However, in the virtualization scenario, it becomes so easy that any adversary in control of Dom0 can launch a successful attack.

Even if the administrator in Dom0 is not malicious, some malicious software installed in Dom0, which has root privileges, or a hacker who exploits some vulnerabilities in the management OS, or even someone who, by chance, has the image file of the memory contents of DomU, can easily break into DomU and extract all the secret information from the encrypted disk. After all, even though not many effective ways to attack the hypervisor layer have been discovered to date, attacks targeted at OSs are numerous.

From the perspective of a remote user, using the techniques proposed in this paper, he can employ Trusted Computing techniques [24] to verify the VMM and does not have to trust the administrator and the management OS. The proposed architecture also suggests a secure network interface, secure runtime environment, and secure secondary storage, combining all of which we build a secure VM execution environment for a remote user.

4 Methodology

In this section, we define the threat model, analyze the security requirements, outline the design of the proposed secure virtualization architecture and then present the relevant details.

4.1 Threat Model

In this paper, we consider the scenario of a client executing a security-critical VM on the remote virtualized computing

environment provided by a cloud computing company. We assume that the small hypervisor layer is verified and its integrity is assured using Trusted Computing techniques [24]. However, the management VM Dom0 is a complete OS and managed by the administrator.

The security threats may come from several parties:

- Attackers from outside of the cloud computing environment. An attacker may exploit the vulnerability of an OS to compromise Dom0, and control root or other privileged access rights.
- Attackers who are clients of the cloud computing environment. Several bugs have been discovered that allow an unprivileged domain (DomU) to gain control of Dom0 [23]. Therefore, a client that runs a DomU in the cloud computing environment can control Dom0 and break into another client's VM.
- Attackers from inside the cloud computing environment. Although the hypervisor layer can be verified from hardware, Dom0 is controlled by the system administrator. A careless or malicious administrator may leak or change sensitive information of the client.

A compromised Dom0 can control the network I/O and secondary storage, but have no access to the hypervisor memory address space. We do not consider hardware attacks and side-channel attacks. Hardware and side-channel attacks require physical access to the computers, which is quite challenging in the cloud computing scenario.

4.2 Security Requirements Analysis

Based on the threat model discussed above, we can summarize the reasons that a client does not trust Dom0 as follows:

- The existence of the vulnerability window for an OS (the time between when a threat is identified and when security vendors release patches).
- The security holes of device drivers located in Dom0, which are often written with no regard to security.
- A careless or malicious system administrator.

The objective of our work is to ensure the confidentiality and integrity of a security-critical VM under an untrusted management VM. To obtain a secure execution environment for a remote computing VM under an untrusted Dom0, DomU should have:

- *A secure network interface between the client and server.* The access control, input commands, and results returned all require a secure interface between the client and server. In the scenario of a mobile user bringing his VM to an untrusted virtualized computing platform, the keyboard and display interfaces also need to be included in the TCB, which is not the case in the cloud computing scenario. In the Xen architecture, any plaintext transferred from DomU to the outside world will be exposed to the potentially malicious Dom0. A technique that protects the confidentiality and integrity of communication is transport layer security (TLS) [25]. However, even if we use TLS, Dom0 can still extract the TLS cryptographic keys from the memory or vCPU registers, which is prevented by the secure runtime environment proposed in this paper.
- *A secure runtime environment.* This includes the secure vCPU state and secure memory, ensuring both confidentiality and integrity. Dom0 must not be allowed to access the sensitive information in the vCPU registers and the memory of the security-critical VM. However, the management of these resources by Dom0 is also necessary. *The mechanism for Dom0 to manage the domains without knowing their contents is the focus and main contribution of this paper.*
- *A secure secondary storage.* Sensitive data need to be stored in secondary storage, e.g., a hard disk, which is provided by the remote computing platform. Another option for storage is the network file system (NFS) [26], allowing the VM to access files over a network. However, all the device drivers of the hard disk or network device are in Dom0 and untrusted. Thus, ensuring confidentiality and integrity of the sensitive data stored in the guest VM is essential.

  If the client uses NFS as secondary storage, the secure network interface is sufficient to realize secure storage. If the hard disk is used for secondary storage, since it is controlled by Dom0, the confidentiality of the data may be compromised if they are not encrypted, and the integrity of the data may also be compromised by Dom0. Therefore, a disk encryption technique, such as `dm-crypt` in Linux and BitLocker in Windows, is necessary. The issue here for a normal full disk encryption is that the OS must be decrypted before the OS can boot, meaning the key has to be available before there is a user interface to ask for a password. In the virtualization scenario, the hypervisor can act as the authority for this preboot authentication. We will discuss this in more detail in Section 4.7.

Among the above three aspects, a secure runtime environment is the most fundamental. On the one hand, there are already solutions, as mentioned before, for secure network interface and secure secondary storage. However, techniques to secure the vCPU state and memory used by the guest VM from the management VM have not been well-researched before. On the other hand, a secure runtime environment is the basis for all the mechanisms needed to make the network and storage secure: all the cryptographic algorithms and security protocols actually reside in the runtime environment. The keys used and the code executed cannot be well-protected unless a secure runtime environment is established. The attack described in Section 3.4 illustrated this clearly: even if AES encryption is used for secure storage, the keys may be extracted from an unprotected runtime environment. *Hence, in the rest of the paper, we will focus only on the design and implementation of a secure runtime environment.*

4.3 Design of a Secure Runtime Environment

In this section, we list a few key aspects involved in the design of a secure runtime environment. A detailed implementation is presented in the next section.

The runtime state of a VM consists of the contents of the memory and vCPU registers. These contents should not be

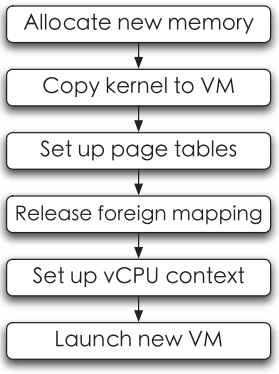


Fig. 3. New VM building process.

leaked to the untrusted Dom0. However, the normal management of VMs also requires Dom0 to allocate memory, read and write to the memory, and get and set the vCPU registers. This apparent conflict is solved in the following manner.

- Memory access from Dom0 to DomU using foreign mapping is by default prohibited except for some specific cases listed below. Therefore, the shared memory between DomU and Dom0 has to use the grant table method, in which DomU initiates the granting and Dom0 asks for access through hypercalls.
- During the execution of functions in which foreign mapping has to be used, the memory page mappings are monitored and controlled by the hypervisor layer. The hypervisor makes sure that it monitors every memory and vCPU access from Dom0 to DomU, and encrypts all the memory pages and vCPU registers if they involve any private information of DomU. Dom0 is provided with an encrypted view of memory pages and vCPU registers for the purpose of saving or restoring state. Thus, the contents of these pages and registers remain secret from Dom0.
- After the access of sensitive information in DomU (in the encrypted view) or the execution of some security-critical domain management tasks, the hypervisor checks the integrity of the runtime state of DomU. The restart or unpause command initiated by Dom0 is validated by the hypervisor only if these integrity checks are successfully completed and no security concerns are raised.

4.4 Details of the Secure Runtime Environment

In this section, we provide further details of how the proposed secure runtime environment is implemented.

4.4.1 Domain Building and Shutdown

In the Xen architecture, domain building is managed by Dom0. We mainly focus on the building of a paravirtualized VM, meaning that the OS in the guest VM must be modified to use hypercalls instead of privileged instructions. Due to the paravirtualization, the low-level interactions with the BIOS are not available in the Xen environment [27].

The main steps for building a new VM in the Xen architecture are shown in Fig. 3. Dom0 first loads the kernel and the ramdisk (optionally) from the secondary storage, which is the hard disk in our case. Then, the new memory area is allocated to the new VM by Dom0. After that, using foreign mapping, the kernel image is loaded into the new VM memory. Next, Dom0 sets up the initial page tables for the new VM. Finally, the new VM is launched after Dom0 releases all the foreign mappings of the new VM memory area and sets up the vCPU registers.

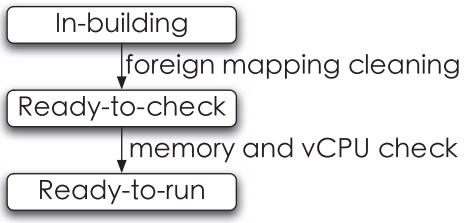


Fig. 4. State of new VM during building.

In this process, if Dom0 is malicious, it may

1. launch a denial-of-service (DoS) attack by refusing to load the kernel, allocate the memory, or start the VM,
2. maliciously modify the kernel image of the new VM to insert rootkits or other external code,
3. set up wrong initial page tables to undermine the integrity of the new VM execution environment,
4. set up the wrong vCPU context (registers) configuration to undermine the integrity of the new VM execution environment, and
5. refuse to release the foreign mappings of the new VM memory area so that it can read that memory area later when the VM is running.

We are interested in defending against the attacks that compromise privacy and integrity. Obviously, 2, 3, and 4 may undermine the integrity of the VM execution environment and 5 may undermine the privacy of the environment. Our solution is to perform the foreign mapping cleaning and integrity check just before launching the new VM in the hypervisor layer.

Foreign mapping cleaning is performed by the hypervisor layer. It checks the page tables of Dom0 and makes sure that none of the new VM memory pages are now mapped to Dom0. It can be implemented either by going through all the page table pages of Dom0, or more efficiently, as we implemented, it can be realized by recording the pages that are currently mapped by Dom0. Thus, if Dom0 refuses to release some of the mappings, the list is not empty.

Integrity check is performed for the new VM kernel and the vCPU context. The remote user is responsible for providing a hash of the correct image of the VM kernel right before the VM starts. Also, the vCPU context is checked for integrity. Dom0 is supposed to use a hypercall for checking the ready-to-launch VM. Upon receiving the hypercall, the hypervisor layer performs the kernel and vCPU integrity check.

To defend against the time-of-check-to-time-of-use (TOCTTOU) attack, which means that Dom0 may do the integrity check first and then modify the kernel image, the hypervisor layer marks the state of the new VM after each hypercall of foreign mapping cleaning and integrity check. As shown in Fig. 4, after the hypercall of foreign

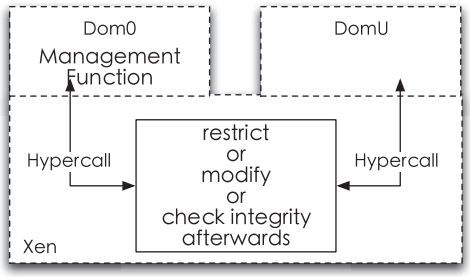


Fig. 5. Intercepting the hypercalls between domains.

mapping cleaning, the state of the VM is changed to "ready-to-check," and then after the integrity check, the state of the VM is changed to "ready-to-run." In the "ready-to-check" and "ready-to-run" states, no further foreign mappings are allowed in order to avoid malicious modification. Also, in the "ready-to-run" state, the vCPU context cannot be set again using the vcpu_set_context hypercall. Hence, the integrity of the vCPU context is ensured.

During domain shutdown, the hypervisor layer needs to make sure that all the memory pages are cleared before they are reallocated to a new domain.

4.4.2 Domain Runtime

During domain execution, the untrusted management domain, Dom0, uses the mechanism of hypercalls to communicate with DomU. Given the secure network interface and secure secondary storage discussed earlier, we now focus on those hypercalls that are potentially harmful to the confidentiality and integrity of the DomU memory content and vCPU context.

The mechanisms that we use to secure the DomU runtime environment, as shown in Fig. 5, are to intercept the hypercalls made from Dom0 to DomU and

1. restrict the use of some hypercalls, in a certain time window or in the whole life cycle of the protected DomU,
2. provide a different, but usable, result for some hypercalls, e.g., provide an encrypted view of a memory page when Dom0 uses foreign mapping hypercalls to access it,
3. check the integrity of the DomU state after some security-critical hypercalls, and
4. design some new hypercalls for security reasons.

We categorize the hypercalls that are used by Dom0 for the management of DomU into three groups:

- Hypercalls that are harmful to the privacy and integrity of DomU, but not necessary for DomU management. Some hypercalls that can access the memory of DomU are related to the functions of IOMMU and debugging. These functions are not necessary in our scenario of a remote computing environment with untrusted management domain. These hypercalls should be prohibited. However, in domains that do not need a highly trusted environment, these functions can still be used. We depend on the hypervisor layer, which is small and easy to verify and thus trusted, to monitor and verify the use of these hypercalls.

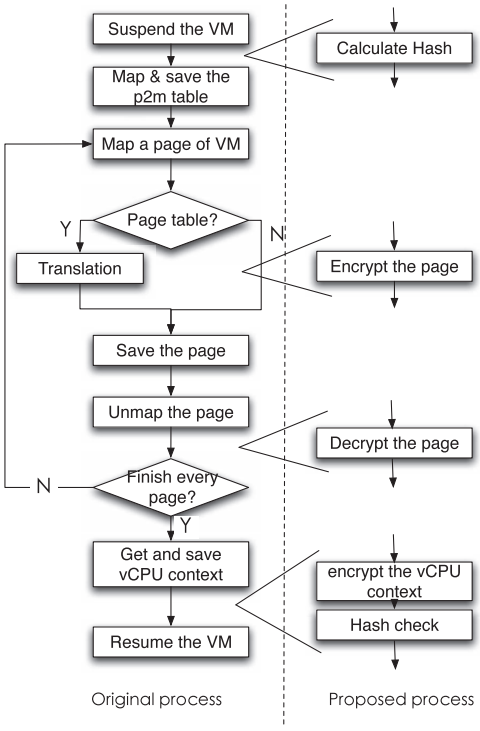


Fig. 6. Domain save process in the original and proposed virtualization architecture.

- Hypercalls that are not harmful to the privacy and integrity of DomU. Some hypercalls are just for management use and are not related to read or write to the memory area or vCPU registers of DomU. These hypercalls can be left unmodified.
- Hypercalls that are harmful to the privacy and integrity of DomU, but necessary for its management. Some hypercalls, such as those for foreign mapping and getting or setting vCPU context, can harm DomU. However, we cannot simply restrict the use of these hypercalls because they are also necessary for the normal management of DomU, e.g., to save or restore the state of DomU.

Note that during the normal runtime of the guest VM, Dom0 does not need to have access to the memory or vCPU registers of DomU, which means that we can easily block these hypercalls for security reasons without interfering with the guest VM.

For the third group, we discuss in detail the hypercalls related to the memory and vCPU. Dom0 mainly uses these hypercalls during the domain save or restore operations, which are illustrated in Figs. 6 and 7, respectively.

First, we define the machine address and physical address. Machine address is the real host memory address, which can be understood by the physical processor. Physical address is given for each VM. The guest VM runs in an illusory contiguous physical address space, which is most likely not contiguous in the machine address space. There is a physical-to-machine (p2m) mapping table stored in each VM, and a machine-to-physical (m2p) mapping table stored in the hypervisor layer.

Another mechanism the Xen architecture uses for the separation of VMs is the management of page tables. All the

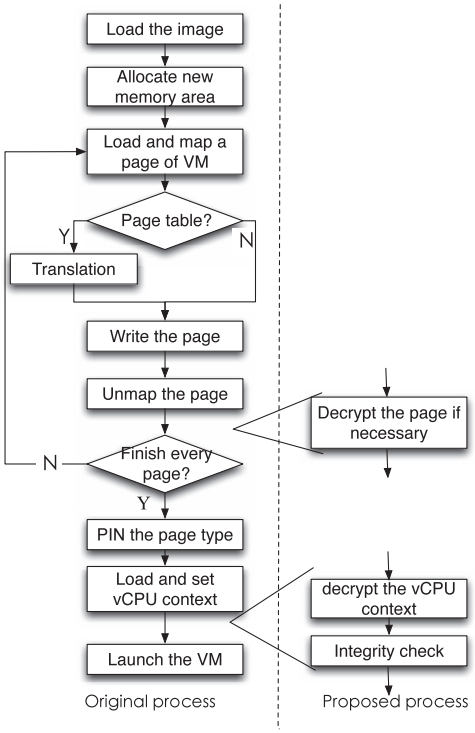


Fig. 7. Domain restore process in the original and proposed virtualization architecture.

page tables are managed by the hypervisor layer instead of the VM itself (including Dom0). In order to map or unmap a page (either a domestic mapping from its own domain or a foreign mapping from DomU to Dom0), a hypercall "page-table update" has to be made. The hypervisor layer is responsible for the security check of whether this update is legal. For example, a foreign mapping initiated from DomU is not allowed.

In the domain save process, Dom0 suspends the VM, and maps the p2m table into its own memory space. An image file to store the memory and vCPU state of DomU is then created. After saving (writing) the p2m table in this image file, Dom0 repeatedly maps and saves each of the memory pages of the VM. For the special type of "page-table page," Dom0 uses the m2p table to translate the machine address to the physical address and then saves the translated page. Dom0 unmaps every memory page after saving its contents in the image file, then makes a hypercall to get the vCPU context and saves it in the same image file. Finally, the original VM resumes execution.

In the domain restore process, Dom0 is responsible for loading the image file and allocating the new memory area (through the use of memory allocation hypercalls). Then, Dom0 maps each page of the newly allocated memory, reads the contents of the image file, and writes every page back to memory. If the page is a "page-table page," a translation of the physical address to the machine address is also needed. Dom0 unmaps the page, and assigns the page type to "page-table page," using the PIN hypercall. After loading and setting the vCPU context, Dom0 is now ready to launch the new VM.

In the proposed process of domain save and restore, we insert some new functions into the original process for the protection of

- *vCPU context privacy and integrity.* We add the encryption/decryption of vCPU context and a hash check. Dom0 has the privilege to get/set the contents of the vCPU registers in domain building and domain save/restore. We prohibit the use of these hypercalls while DomU is running. In addition, during domain save, once Dom0 makes the hypercall to get the vCPU context, the hypercall is intercepted. The contents of the vCPU registers are first encrypted and a keyed-hash is calculated, both of which are included in the result of the hypercall. Hence, Dom0 can only see the encrypted view of the vCPU registers and any malicious modification of the context can be detected. In the same way, the vCPU context is decrypted and an integrity check scheduled during domain restore.
- *VM memory privacy.* Under an untrusted Dom0 scenario, we cannot let the private information in VM memory leak into Dom0. However, in some important domain management operations, such as domain save and restore, foreign mapping is necessary for acquiring and restoring the VM state. We solve this problem by intercepting the "page-table update" hypercall. After Dom0 asks for a page-table update through a hypercall, the hypervisor layer checks whether it is a mapping or unmapping operation. For a mapping hypercall, which maps a foreign VM page, this page is encrypted first so that Dom0 only sees the encrypted view of that page. Dom0 can then save this encrypted view to the file image, without knowing the actual contents. All the memory mappings/unmappings are managed by the hypervisor instead of Dom0, so that Dom0 cannot cheat by using aliasing or indirections. If the hypercall is for unmapping a foreign VM page, the hypervisor updates the page table in Dom0 and then decrypts the memory page. Because these hypercalls are prohibited when DomU is running, normal processing in DomU is not affected.
- *VM memory integrity.* Integrity protection of the VM memory is based on the simple fact that the memory view of a VM should be unchanged from the time just before domain save to the time just after domain restore. We use the hypervisor layer to calculate a hash of all memory pages just before domain save and perform integrity check just after domain restore.

There are some further issues involving hash calculation and checking that deserve attention. The memory views in save and restore are not exactly identical—they are simply "functionally the same": Dom0 may allocate different regions of memory to the newly restored DomU (the original memory may be dynamically allocated to other domains/applications). Hence, in the new domain, the machine addresses in all the page-table pages are changed to new ones, which are different from the

TABLE 1
Performance Measurement for Domain Build, Save, and Restore

Time (s)	64M-ori	64M-mod	128M-ori	128M-mod	256M-ori	256M-mod
Domain build time (s)	0.210	0.347	0.220	0.402	0.225	0.527
Domain save time (s)	1.976	2.612	3.743	5.182	7.353	10.774
Domain restore time (s)	1.580	2.742	2.929	5.282	5.680	10.537

ones in the old domain. Thus, a simple hash calculation and check mechanism does not work. The solution is the *use of the physical address instead of the machine address during hashing*. We translate the machine address to physical address for every "page-table page." Whatever the machine address changes to, the physical address, which is relative and in an illusory contiguous address space, never changes. After the translation, the integrity check mechanism works correctly.

- *vCPU and memory freshness.* To ensure the freshness of the vCPU context and memory content and prevent a replay attack, version information can be added to the hash. After domain save, the version information is transmitted back to the remote user. Once the remote user wants to start or restore a VM, the version information needs to be specified and finally verified by the VMM. To avoid a mix-and-match attack, which uses memory pages from an old snapshot and vCPU registers from another, the version information of both need to be the same.

4.5 Domain Shutdown

During domain shutdown, the hypervisor layer needs to make sure that all the memory pages are cleared before they are reallocated to a new domain.

4.6 Live Migration

In this section, we analyze the applicability of the proposed methodology to the live migration of VMs.

Live VM migration allows the administrators to separate hardware and software considerations. For example, if a physical machine needs to be removed or upgraded in the cloud computing infrastructure, a live migration of VMs to another physical machine helps minimize the downtime of the VM.

Xen uses an approach of precopy migration [28]. During memory migration from host A to host B, the main steps are as follows: 1) Iterative precopy: all memory pages are transferred from A to B in the first iteration. The following iterations copy only those pages modified during and after the first transfer. 2) Stop-and-copy: host A is suspended and redirects its network traffic to B. The vCPU state and any remaining inconsistent memory pages are then transferred, to make sure the copies of VM at both A and B are consistent. 3) The VM on B is started.

The proposed methodology can be applied to live migration using the following steps:

1. In the iterative precopy step, memory pages are encrypted before Dom0 on host A maps the page to its own address space.
2. A hash for each transferred page is calculated (instead of a hash of the whole memory image shown in Fig. 6).
3. The memory page is decrypted on host B and integrity is checked.
4. In the stop-and-copy step, the vCPU state is also encrypted and hashed before being transferred.

The main difference in the application of the proposed methodology between the normal domain save/restore and live migration is integrity protection. Because the memory pages keep changing during live migration, a hash calculation per memory page, rather than a hash of whole memory image, is needed. These hash results, along with the encrypted memory pages, are transferred to host B.

We did not implement our methodology for live migration because we believe live migration and domain save/restore are not fundamentally different. Moreover, because of the use of the same procedure for encryption/decryption/hashing on the memory pages, the downtime in live migration, which occurs in the stop-and-copy step, should bear the same overhead (around $2\times$, as shown in Table 1).

4.7 Key Management

In the protection mechanisms discussed before, several encryption and keyed-hash functions are used. The keys used in the proposed protection mechanisms include:

1. keys used in secure network communication protocols [TLS, virtual private network (VPN), etc.] between DomU and the remote user,
2. keys used in the encryption of secure secondary storage,
3. keys used in vCPU register encryption and keyed-hash, and
4. keys used in memory page encryption and keyed-hash.

These keys cannot be simply stored permanently in plaintext because the secondary storage is not trusted, nor can these keys be protected using the password, because the network interface to transmit passwords is insecure under an untrusted Dom0, which controls all the network hardware and device drivers.

A possible approach to protect all the above keys would be the use of a key file including all these keys, which itself is protected by a master key—both by encryption and keyed-hash. Hence, all the key management systems include one secret—the master key, three entities—DomU, hypervisor, and remote user, two potential attackers—Dom0 and an attacker in the normal network threat model who exploits the untrusted network, and finally, two channels—an untrusted network from the remote user

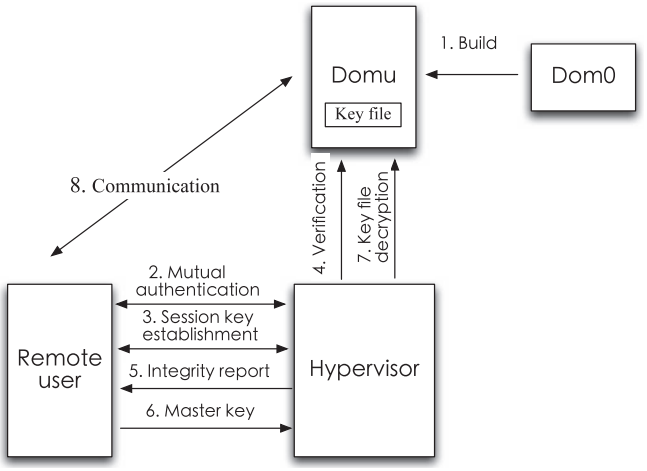


Fig. 8. Proposed protocol for key establishment.

and host machine and a trusted channel from the hypervisor to DomU (using hypercalls).

Unlike the traditional key management models, this new three-entity scenario has the following requirements and assumptions:

Requirement 1: The master key cannot be stored in the permanent storage of the host machine. As we mentioned earlier, all the I/O devices and their drivers are controlled by Dom0. Any material stored in them in the plaintext format is not secure.

Requirement 2: The master key should be a session key rather than a long-term key. In the remote computing scenario, Dom0 may have several saved images of DomU. Hence, "forward secrecy" is important, which means an incidental leak of the master key should not be allowed to be exploited to unseal the previously encrypted images of DomU. Furthermore, it is more secure and reasonable to give remote users the option to update master keys as they wish.

Requirement 3: It is easier for a compromised Dom0 to undermine the whole computing environment than for a normal outside attacker. Hence, traditional attacks, like the man-in-the-middle attack (actually Dom0 acts here as the relay between DomU and the remote user) or the replay attack, should be strongly prevented.

Requirement 4: Dom0 also has the ability to build a new domain, which is beyond the ability of an attacker in the normal network threat model. Hence, an attack consisting of building a domain to impersonate DomU and getting the master key from the remote user should be prevented.

Assumption 1. *In order to authenticate the host machine and remote user, we need an assumption that the public key of the hypervisor (the cloud provider) is known by the remote user and the integrity of the hypervisor can be verified by the remote user.*

Assumption 2. *The private key of the hypervisor can be stored in hardware that is not under the control of Dom0. The TPM chip and its "sealed storage" [24] is a good candidate for this purpose, which may be controlled only by the hypervisor.*

Note that though a shared secret is needed between the remote user and running VM, it cannot be simply stored in the VM image provided by the remote user because Dom0 has access to this information.

Our key establishment goal is to establish trust from the public key of the VMM to a secure sharing of various keys (including the SSH keys, storage keys, and vCPU register and memory page keys) among DomU, hypervisor, and remote user, but protected from Dom0. To achieve this goal, the following steps, as shown in Fig. 8, are proposed.

Step 1: Dom0 first builds DomU. The key file is now encrypted by the master key and is thus safe. However, Dom0 may launch an attack based on modification of the kernel of DomU or the key file, which can be prevented in Steps 4 and 5.

Steps 2 and 3: Using the public key of the hypervisor, a public-key cryptography algorithm (such as RSA) can be used for mutual authentication and session key establishment between the hypervisor and remote user.

Step 4: To prevent Dom0 from maliciously modifying the DomU kernel, the hypervisor should check the integrity of DomU—hash the kernel memory and the key file.

Step 5: The hypervisor reports the hash of the built DomU, the hash of the keyfile, and the hash of itself to the remote user for attestation. The communication should be protected from the session key established before.

Step 6: After the verification of the correctness of the hash sent by the hypervisor, the remote user now trusts the hypervisor and then sends the master key, protected from the session key established before.

Step 7: The hypervisor decrypts the keyfile using the master key transmitted from the remote user. However, the decrypted content of the key file cannot be stored in the secondary storage. All the information should only be kept in the memory, which is protected by the proposed mechanism of vCPU registers and VM memory protection.

Step 8: Now using the content from the key file, DomU and the remote user finally share the keys and can communicate with each other securely, protected from the untrusted Dom0. Also, the remote users are free to update any keys in the key file as they wish, as long as the hash of the key file is verified in the next session.

5 Experimental Results

We implemented the proposed secure virtualization architecture in the Xen virtualization system and evaluated its performance penalties through execution-specific domain operations as well as several benchmarks. The proposed protection mechanism restricts memory and vCPU access from Dom0 using foreign mapping, which is normally used only in domain build, domain save, and domain restore operations. Hence, we can expect a very small overhead during the normal execution of DomU.

We used a PC equipped with a 2.53 GHz dual-core Intel CPU and 2 GB RAM, and employed Ubuntu Linux 8.04 and Xen 3.2.2 for the virtualization system. We varied the memory size of DomU from 64 to 256 MB.

5.1 Performance Measurement for Domain Build, Save, and Restore

There are two parts to the performance measurement experiments. The first part measures the execution time of

TABLE 2
Performance Measurement for Other Benchmarks

Benchmark	Nbench (memory index)	Nbench (integer index)	Nbench (floating point index)	OSDB-IR (tup/s)	OSDB-OLTP (tup/s)	Dbench (MB/s)
Original	19.115	18.330	33.466	297.75	324.45	299.53
Modified	19.064	18.301	33.410	297.53	321.02	296.38
Penalty	0.27%	0.16%	0.17%	0.07%	1.06%	1.05%

domain build, domain save, and domain restore in both the original Xen system and the modified system with the proposed memory and vCPU protection. We measured the time required for building, saving, and restoring DomU with 64, 128, and 256 MB memory sizes. The overhead of the modified Xen system is mainly due to the following factors:

1. verification of memory access from Dom0 to DomU,
2. encryption of every memory page during domain save,
3. verification and decryption after Dom0 unmaps the memory pages of DomU, and
4. integrity check after domain build, save, and restore.

Table 1 shows that the domain build time has an overhead of $1.7\times$ to $2.3\times$, the domain save time an overhead of $1.3\times$ to $1.5\times$, and the domain restore time an overhead of $1.7\times$ to $1.9\times$. The overhead may seem significant, however, note that domain build only occurs once in the whole life cycle of DomU and domain save/restore occur only when Dom0 needs to back up the state of DomU. These events may have a frequency of once an hour or several hours, even once a day. Hence, we believe that the overall overhead for the proposed protection mechanism is quite acceptable.

5.2 Performance Measurement for Normal Execution of DomU

The second part measures the performance of benchmarks run in DomU of both the original and modified systems. Cryptographic overhead is incurred only when Dom0 tries to access the memory page of DomU. Hence, we can expect the overhead to be very small during the normal execution of DomU. We ran several benchmarks in both the original Xen system and the proposed secure virtualization system to quantify these overheads. All the measurements were taken for a DomU with 256 MB memory.

The benchmarks used are as follows: Nbench [29] is a port of the BYTEmark benchmark [30] to Linux/Unix platforms, and is CPU-intensive. It is designed to expose the capabilities of a computing platform's CPU, FPU, and memory system. Using the PostgreSQL database, we next exercised the open-source database benchmark (OSDB) [31]. Two results are presented for multiuser information retrieval (IR) and online transaction processing (OLTP) workloads, both in tuples per second. Dbench [32] is a file system benchmark to measure the throughput experienced by a single client.

From Table 2, we can see that the penalty incurred by the proposed enhancements to the Xen system is very small. We believe that such a small penalty is quite acceptable in light of the fact that in the presence of the proposed protection mechanisms, DomU is now run in a more secure runtime execution environment.

6 Conclusion

In this paper, we proposed a virtualization architecture to ensure a secure VM execution environment under an untrusted management OS. The mechanism includes a secure network interface, secure secondary storage, and most importantly, a secure runtime execution environment. We implemented the secure runtime environment in the Xen virtualization system. Using the proposed mechanism, DomU is protected from the untrusted management domain Dom0, while Dom0 can still carry out the normal domain administrative tasks, such as domain build, domain save, and domain restore. Performance evaluation shows that the overhead is mainly due to domain build, save, and restore operations, which occur only once or at a very low frequency during the whole life cycle of DomU. The execution of DomU remains almost the same in terms of performance, with a slowdown of at most 1.06 percent.

We believe that using the proposed secure virtualization architecture, even under an untrusted management OS, a trusted computing environment can be created for a VM which needs a high security level, with very small performance penalties.

Acknowledgments

This work was supported in part by US National Science Foundation (NSF) Grant No. CNS-0720110 and in part by NSF Grant No. CNS-0914787.

References

[1] T. Garfinkel and M. Rosenblum, "When Virtual Is Harder Than Real: Security Challenges in Virtual Machine Based Computing Environments," *Proc. Conf. Hot Topics in Operating Systems*, pp. 20-25, June 2005.

[2] J. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum, E. Felten, and E. Foundation, "Lest We Remember: Cold Boot Attacks on Encryption Keys," *Proc. Usenix Security Symp.*, pp. 45-60, July 2008.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," *Proc. ACM Symp. Operating Systems Principles*, no. 5, pp. 164-177, Oct. 2003.

[4] VMware Player, http://www.vmware.com/products/player, 2012.

[5] EC2, http://www.redhat.com/f/pdf/rhel/EC2_Ref_Arch_V1.pdf, 2012.

[6] R. Caceres, C. Carter, C. Narayanaswami, and M.T. Raghunath, "Reincarnating PCs with Portable SoulPads," *Proc. ACM MobiSys*, pp. 65-78, 2005.

[7] C. Li, A. Raghunathan, and N.K. Jha, "Secure Virtual Machine Execution under an Untrusted Management OS," *Proc. Int'l Conf. Cloud Computing*, pp. 172-180, July 2010.

[8] M. Price and A. Partners, "The Paradox of Security in Virtual Environments," *Computer*, vol. 41, no. 11, pp. 22-28, Nov. 2008.
[9] X. Jiang, X. Wang, and D. Xu, "Stealthy Malware Detection through VMM-Based 'Out-of-the-Box' Semantic View Reconstruction," *Proc. ACM Conf. Computer and Comm. Security*, pp. 128-138, Oct. 2007.
[10] B. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An Architecture for Secure Active Monitoring Using Virtualization," *Proc. IEEE Symp. Security and Privacy*, pp. 233-247, May 2008.
[11] N.L. Petroni Jr. and M. Hicks, "Automated Detection of Persistent Kernel Control-Flow attacks," *Proc. ACM Conf. Computer and Comm. Security*, pp. 109-115, Oct. 2007.
[12] R. Riley, X. Jiang, and D. Xu, "Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing," *Proc. Int'l Symp. Recent Advances in Intrusion Detection*, pp. 1-20, Sept. 2008.
[13] A. Srivastava and J. Giffin, "Tamper-Resistant, Application-Aware Blocking of Malicious Network Connections," *Proc. Int'l Symp. Recent Advances in Intrusion Detection*, pp. 39-58, Sept. 2008.
[14] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A Virtual Machine-Based Platform for Trusted Computing," *Proc. ACM Symp. Operating Systems Principles*, pp. 193-206, Oct. 2003.
[15] X. Chen, T. Garfinkel, E.C. Lewis, P. Subrahmanyam, C.A. Waldspurger, D. Boneh, J. Dwoskin, and D.R. Ports, "Over-Shadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 2-13, Mar. 2008.
[16] J. Yang and K.G. Shin, "Using Hypervisor to Provide Data Secrecy for User Applications on a Per-Page Basis," *Proc. ACM Int'l Conf. Virtual Execution Environments*, pp. 71-80, Mar. 2008.
[17] S. King and P. Chen, "SubvVirt: Implementing Malware with Virtual Machines," *Proc. IEEE Symp. Security and Privacy*, pp. 314-327, May 2006.
[18] R. Sailer, E. Valdez, T. Jaeger, R. Perez, L. van Doorn, J. Griffin, and S. Berger, "sHype: Secure Hypervisor Approach to Trusted Virtualized Systems," IBM Research Report RC23511, 2005.
[19] P. Karger and D. Safford, "I/O for Virtual Machine Monitors: Security and Performance Issues," *IEEE Security and Privacy*, vol. 6, no. 5, pp. 16-23, Sept./Oct. 2008.
[20] D. Murray, G. Milos, and S. Hand, "Improving Xen Security through Disaggregation," *Proc. ACM Int'l Conf. Virtual Execution Environments*, pp. 151-160, Mar. 2008.
[21] M. Ben-Yehuda, J. Mason, O. Krieger, J. Xenidis, L. van Doorn, A. Mallick, J. Nakajima, and E. Wahlig, "Utilizing IOMMUs for Virtualization in Linux and Xen," *Proc. Ottawa Linux Symp.*, 2006.
[22] Intel VT-D, http://www.intel.com/technology/virtualization/technology.htm, 2012.
[23] Xen Owning Trilogy, http://www.invisiblethingslab.com/itl/Resources.html, 2012.
[24] Trusted Platform Module (TPM) Specifications, https://www.trustedcomputinggroup.org/specs/TPM/, 2012.
[25] The Transport Layer Security (TLS) Protocol Version 1.2, http://tools.ietf.org/html/rfc5246, 2012.
[26] NFS, http://tools.ietf.org/html/rfc3530, 2012.
[27] D. Chisnall, *The Definitive Guide to the Xen Hypervisor*. Prentice Hall, 2008.
[28] C. Clark, K. Fraser, S. Hand, J.G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live Migration of Virtual Machines," *Proc. Symp. Networked Systems Design and Implementation*, pp. 273-286, May 2005.
[29] Nbench, http://www.tux.org/mayer/linux/bmark.html, 2012.
[30] BYTEmark, http://www.tux.org/mayer/linux/byte/bdoc.pdf, 2012.
[31] OSDB, http://osdb.sourceforge.net, 2012.
[32] Dbench, http://dbench.samba.org, 2012.

Chunxiao Li received the BEng degree in electrical engineering from Tsinghua University, Beijing, China, in 2007 and the MA degree in electrical engineering from Princeton University, New Jersey in 2009. He is currently working toward the PhD degree in the Electrical Engineering Department at Princeton University. His research interests include embedded systems and computer security. He is a student member of the IEEE.

Anand Raghunathan received the BTech degree in electrical and electronics engineering from the Indian Institute of Technology, Madras, India, in 1992, and the MA and PhD degrees in electrical engineering from Princeton University, New Jersey, in 1994 and 1997, respectively. He is currently a professor in the School of Electrical and Computer Engineering at Purdue University, West Lafayette, Indiana, and serves as vice-chair of the Tutorials & Education Group at the IEEE Computer Society's Test Technology Technical Council. He has been a member of the technical program and organizing committees of several leading conferences and workshops. He has coauthored a book and holds or has filed for 24 US patents in the areas of advanced system-on-chip architectures, design methodologies, and VLSI CAD. He has received six best paper awards and three best paper nominations at leading conferences. He was chosen by MIT's Technology Review among the TR35 in 2006, for his work on "making mobile secure." He was a recipient of the IEEE Meritorious Service Award (2001) and the Outstanding Service Award (2004), and was elected a Golden Core Member of the IEEE Computer Society in 2001, in recognition of his contributions. He is a fellow of the IEEE.

Niraj K. Jha received the BTech degree in electronics and electrical communication engineering from the Indian Institute of Technology, Kharagpur, India, in 1981, the MS degree in electrical engineering from the State University of New York (SUNY) at Stony Brook in 1982 and the PhD degree in electrical engineering from the University of Illinois, Urbana, in 1985. He is a professor of electrical engineering at Princeton University. He served as the editor-in-chief of *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* and as an associate editor of *IEEE Transactions on Computers*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, and *IEEE Transactions on Circuits and Systems I and II*, and currently serves as an associate editor of the *Journal of Low Power Electronics* and the *Journal of Nanotechnology*. He has published more than 370 papers, of which 14 have received various awards and six have received best paper award nominations. He is also a coauthor/coeditor of five books, 12 book chapters, and 14 patents. His research interests include FinFETs, IC power/thermal analysis and optimization, computer-aided design of integrated circuits and systems, digital system testing, computer architecture, and computer security. He is a fellow of the IEEE and the ACM.