2010 IEEE 3rd International Conference on Cloud Computing

Secure Virtual Machine Execution under an Untrusted Management OS

Chunxiao Li
Department of EE
Princeton University
chunxiao@princeton.edu

Anand Raghunathan
School of ECE
Purdue University
raghunathan@purdue.edu

Niraj K. Jha
Department of EE
Princeton University
jha@princeton.edu

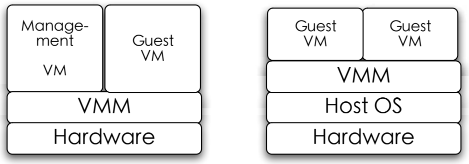
***Abstract*—Virtualization is a rapidly evolving technology that can be used to provide a range of benefits to computing systems, including improved resource utilization, software portability, and reliability. For security-critical applications, it is highly desirable to have a small trusted computing 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.**

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 run-time 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.

***Keywords*-virtual machine; trusted computing base; memory protection; cloud computing; computing as a service;**

I. 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. 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.



(a) Type I

(b) Type II

Figure 1. Types of virtualization architectures

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 [2]. In Type II virtualization architectures, such as VMware Player [3], 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.

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) [4] 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. 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

Acknowledgment: This work was supported in part by NSF grant No. CNS-0720110 and in part by NSF grant No. CNS-0914787.

978-0-7695-4130-3/10 $26.00 © 2010 IEEE
DOI 10.1109/CLOUD.2010.29

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 [5]). 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 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 (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 and the administrative interface of the management OS are exposed more often to careless or even malicious administrators.

In this paper, we mainly focus on Type I virtualization architectures, and take Xen as a prototype for demonstration – actually Amazon EC2 itself is a Xen-based infrastructure [4]. We show how the management OS can be removed from the TCB of the VM, thereby ensuring the data confidentiality and integrity of a VM execution environment even under an untrusted management OS.

II. Related work

The relationship between virtualization and security is a paradox [6], 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. A lot of research studies [7], [8], [9], [10] utilize virtualization to implement introspection from the secure domain to the target domain. Terra [11] 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 [12] and SP3 [13] rely on the underlying VMM (also called hypervisor) to separate processes from untrusted guest OSs.

On the other hand, the security of virtualization itself is a significant concern. In [1], security challenges in virtual environments are summarized.

As indicated in [14], 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.

In the Terra [11] architecture, the property "root secure" may incorrectly suggest that it has already achieved the goal of excluding the management OS out of 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, are actually implemented in the VMM, which itself is trusted in Terra's threat model. Our proposed architecture is different from Terra in that we exclude out of the TCB not only the administration command interface (as Terra did) but also most of the complex administrative functionalities themselves.

III. 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.

A. The Xen 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. The mechanism used for inter-VM communication is shared memory, which can be established through either the grant table or foreign mapping.

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. 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. Another role played by Dom0 in the Xen architecture is the task of VM management. 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.

B. Security Threats to DomU from Untrusted Dom0

We first describe a scenario for the security threats described in this subsection. 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 untrusted management domain, *i.e.*, Dom0 in Xen, is capable of undermining the confidentiality, integrity and the 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.

C. A Concrete Attack Example

In the simplest scenario, we encrypt some plaintext in a VM to ciphertext, and place the ciphertext on the hard disk, so that we are not worried even if the hard disk is lost or stolen. The keys are always located in system memory. During normal system execution, we do not worry much about memory safety. Under cold-boot attacks [15], 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 [15] to analyze a VM image file of size 128MB and were able to successfully identify all cryptographic keys in less than one second on a mainstream desktop, as described in Section V. 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.

IV. Methodology

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

A. Security Requirements Analysis

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 [16]. However, the management VM Dom0 is a complete OS and managed by the administrator. The client does not trust Dom0 because of the existence of the vulnerability window (between when a threat is identified and when security vendors release patches), the security holes of device drivers and careless or malicious system administrators.

The objective of our work is to ensure the confidentiality and integrity of a security-critical VM under an untrusted management VM. We do not consider hardware attacks, side-channel attacks and direct memory access (DMA) attacks. Hardware and side-channel attacks require physical access to the computers, which is quite challenging in the cloud computing scenario. Defending against the DMA attacks requires help from the input/output memory management unit (IOMMU) [17], whose controlling code should reside in the hypervisor.

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 the server.* The access control, input commands and results returned all require a secure interface between the client and the server. A technique that protects the confidentiality and integrity of communication is transport layer security (TLS) [18]. However, even if we use TLS, Dom0 can still extract the TLS cryptographic keys from the memory or virtual CPU (vCPU) registers, which is prevented by the secure run-time environment proposed in this paper.
- *A secure run-time environment.* This includes 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.

Among the above three aspects, a secure run-time 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 run-time 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 run-time environment. The keys used and the code executed cannot be well-protected unless a secure run-time environment is established. The attack described in Section III-C illustrated this clearly: even if AES encryption is used for secure storage, the keys may be extracted from an unprotected run-time environment. *Hence, in the rest of the paper, we will focus only on the design and implementation of a secure run-time environment.*

B. Design of a Secure Run-time Environment

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

- Memory access from Dom0 to DomU using foreign mapping is by default prohibited except for some specific cases listed below. Therefore, all 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 run-time state of DomU.

C. Details of the Secure Run-time Environment

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

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 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 [19].

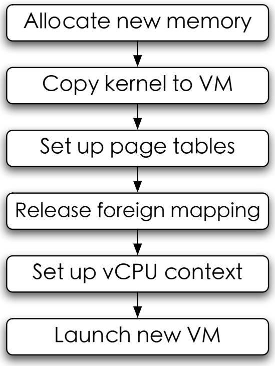


Figure 2. New VM building process

The main steps for building a new VM in the Xen architecture are shown in Fig. 2. 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.

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.

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

2) Domain Run-time: 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 DomU memory content and vCPU context.

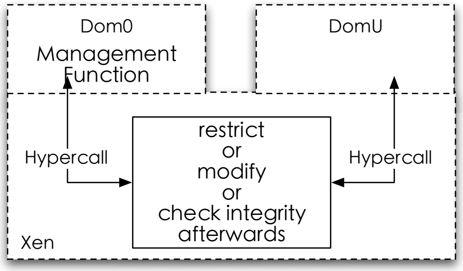


Figure 3. Intercepting the hypercalls between domains

The mechanisms that we use to secure the DomU runtime environment, as shown in Fig. 3, 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 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.
- 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.

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. 4 and 5, 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 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 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.

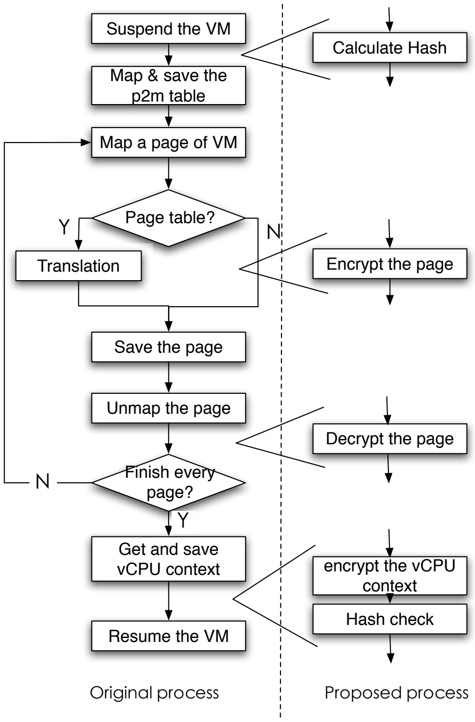


Figure 4. Domain save process in the original and proposed virtualization architecture

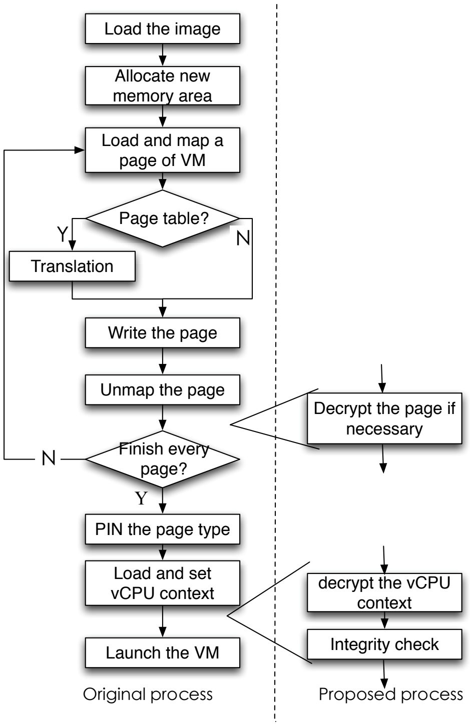


Figure 5. Domain restore process in the original and proposed virtualization architecture

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. 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. 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 (1) vCPU context privacy and integrity, (2) VM memory privacy, (3) VM memory integrity, and (4) vCPU and memory freshness.

- *vCPU context privacy and integrity.* We add the encryption/decryption of vCPU context and a hash check. 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. 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.
- *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. Hence, in the new domain, the machine addresses in all the page-table pages are changed to new ones. And 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." 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. 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.

V. 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.

We used a PC equipped with a 2.53GHZ dual-core Intel CPU and 2GB RAM, and employed Ubuntu Linux 8.04 and Xen 3.2.2 for the virtualization system. We varied the memory size of DomU from 32MB to 256MB.

There are two parts to the performance measurement experiments. The first part measures the execution time of 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 a DomU with 64MB, 128MB and 256MB memory sizes.

Table I shows that the domain build time has an overhead of 1.7× to 2.3×, the domain save time an overhead of 1.3× to 1.5×, and the domain restore time an overhead of 1.7× to 1.9×. 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.

The second part measures the performance of benchmarks run in DomU of both the original and modified systems. 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 256MB memory.

The benchmarks used are as follows. Nbench is a port of the BYTEmark benchmark to Linux/Unix platforms, and is CPU-intensive. Using the PostgreSQL database, we next exercised the open-source database benchmark (OSDB). Two results are presented for multi-user information retrieval (IR) and on-line transaction processing (OLTP) workloads, both in tuples per second. Dbench is a file system benchmark derived from NetBench. We used Dbench to measure the throughput experienced by a single client.

From Table II, we can see that the overhead incurred by the proposed enhancements to the Xen system is very small.

VI. 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 run-time execution environment. We implemented the secure run-time 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%.

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.

References

[1] T. Garfinkel and M. Rosenblum, "When virtual is harder than real: Security challenges in virtual machine based computing environments," in *Proc. Conf. Hot Topics in Operating Systems*, June 2005, pp. 20–25.

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

[3] "VMware Player,http://www.vmware.com/products/player."

[4] "EC2, http://www.redhat.com/f/pdf/rhel/EC2_Ref_Arch_V1.pdf."

[5] R. Caceres, C. Carter, C. Narayanaswami, and M. T. Raghunath, "Reincarnating PCs with portable SoulPads," in *Proc. ACM/USENIX MobiSys*, 2005, pp. 65–78.

[6] M. Price and A. Partners, "The paradox of security in virtual environments," *Computer*, vol. 41, no. 11, pp. 22–28, 2008.

Table I
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

Table II
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
Overhead	0.27%	0.16%	0.17%	0.07%	1.06%	1.05%

[7] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction," in *Proc. ACM Conf. Computer and Communications Security*, Oct. 2007, pp. 128–138.

[8] B. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization." in *Proc. IEEE Symp. Security and Privacy*, May 2008, pp. 233–247.

[9] N. L. Petroni, Jr. and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proc. ACM Conf. Computer and Communications Security*, Oct. 2007, pp. 109–115.

[10] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing," in *Proc. Int. Symp. Recent Advances in Intrusion Detection*, Sep. 2008, pp. 1–20.

[11] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," in *Proc. ACM Symp. Operating Systems Principles*, Oct. 2003, pp. 193–206.

[12] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems," in *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, Mar. 2008, pp. 2–13.

[13] J. Yang and K. G. Shin, "Using hypervisor to provide data secrecy for user applications on a per-page basis," in *Proc. ACM Int. Conf. Virtual Execution Environments*, Mar. 2008, pp. 71–80.

[14] P. Karger and D. Safford, "I/O for virtual machine monitors: Security and performance issues," *IEEE Security & Privacy*, vol. 6, no. 5, pp. 16–23, Sept.-Oct. 2008.

[15] 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," in *Proc. Usenix Security Symp.*, July 2008, pp. 45–60.

[16] "Trusted Platform Module (TPM) specifications, https://www.trustedcomputinggroup.org/specs/TPM/."

[17] 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," in *Proc. Ottawa Linux Symp.*, 2006.

[18] "The Transport Layer Security (TLS) Protocol Version 1.2, http://tools.ietf.org/html/rfc5246."

[19] D. Chisnall, *The Definitive Guide to the Xen Hypervisor*. Prentice Hall, 2008.