This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2015.2469657, IEEE Transactions on Cloud Computing

Secure Outsourcing of Virtual Appliance

Yubin Xia, Yutao Liu, Haibing Guan, Yunji Chen, Tianshi Chen, Binyu Zang, Haibo Chen

Abstract—Computation outsourcing using virtual appliance is getting prevalent in cloud computing. However, with both hardware and software being controlled by potentially curious or even malicious cloud operators, it is no surprise to see frequent reports of security accidents, like data leakages or abuses. This paper proposes Kite, a hardware-software framework that guards the security of tenant's virtual machine (VM), in which the outsourced computation is encapsulated. Kite only trusts the processor and makes no security assumption on external memory, devices, or hypervisor. Unlike prior hardware-based approaches, Kite retains transparency with existing VM and requires few changes to the (untrusted) hypervisor by introducing VM-Shim mechanism. Each VM-Shim instance runs in between its VM and the hypervisor, which only transfers necessary information designated by the VM to the hypervisor and external environments. Kite also considers the high-level semantic of interaction between VM and hypervisor to defend against attacks through legitimate operations or interfaces. We have implemented a prototype of Kite's secure processor in a QEMU-based full-system emulator and its software components on real machine. Evaluation shows that the performance overhead of Kite ranges from 0.5%-14.0% on simulated platform and 0.4%-7.3% on real hardware.

Index Terms—Computer Architecture, Virtual Machine, Secure Processor, Cloud Computing, Security, Computation Outsourcing

1 Introduction

The convenience, low price, and effectiveness of cloud computing makes computation outsourcing more and more popular. One common way of computation outsourcing is encapsulating code and data in virtual machines (VMs), aka. *virtual appliance* (VA) [2], and deploying the VMs on IaaS (Infrastructure as a Service) cloud, such as Amazon's EC2. However, currently, outsourcing computation means losing all of the control. A malicious cloud provider could steal or tamper with tenants' data, pirate or abuse tenants' valuable algorithm, or even deliver wrong computing results. For example, an IC design company would like to outsource the process of chip validation, which involves a huge amount of computation, to the cloud. By doing so, the company is facing three possible security issues. First, both the data of chip design and algorithm of validation are top secrets, which might be stolen from the cloud. Second, the validation process should keep intact. A malicious cloud provider may tamper with the process to generate a wrong result. Third, the company may also want to host the validation on the cloud and offer it as a service to other chip designers, and charge them by the times of using. In this case, a malicious cloud provider could conceal the times of using or even abuse tenant's service while the tenant has no way to prohibit.

Currently, the mainstream multi-tenant cloud providers have very limited assurance on protecting the security of tenant's data or code. For example, the user agreement of Amazon's Web Services (AWS) explicitly states that tenants are responsible to secure their contents [3]. Similarly, Microsoft's Azure cloud platform requires tenants to encrypt their data before putting into the cloud [4]. However, encrypted data will be ultimately decrypted in the cloud when being processed. Thus, it is no surprise that a recent survey over 500 chief executives and IT managers shows that they are reluctant to move their business to cloud due to "fear about security threats and loss of control of data and systems" [5]. Worse even, the threats are not groundless but real. In 2010, Google fired their employees for "breaking internal privacy policies" and causing "a massive breach of privacy" in Gtalk and Google Voice [6]. Surprisingly, the privacy breach lasted for several months before being detected.

There are two main reasons for such limited security assurance. First, the hardware and software stack in multi-tenant cloud, which usually adopts off-the-shelf hardware and virtualized infrastructures, is notoriously large and complex, which raises the possibility of security compromises on the virtualized stack. For example, among hundreds of reported security vulnerabilities in hostless virtualized infrastructures such as Xen and VMware, more than a third of them cause a privilege escalation and thus allow the adversaries to gain the control of a whole machine [7]. Second, though typical cloud vendors do place some physical (e.g., surveillance cameras and extra security personnel) and software access control to cloud operators, it is hard to strictly limit the behavior of cloud operators as software and hardware maintenance (e.g., memory or device replacement) of a cloud platform has become a

- *Yubin Xia, Yutao Liu, Binyu Zang and Haibo Chen are with the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai, China. E-mail: {xiayubin, ytliu.cc, byzang, haibochen}@sjtu.edu.cn*
- *Haibing Guan is with the Department of Computer Science, Shanghai Jiao Tong University, Shanghai, China. E-mail: hbguan@sjtu.edu.cn*
- *Yunji Chen and Tianshi Chen are with the Institute of Computing Technology, Chinese Academy of Sciences, China. E-mail: {cyj, chentianshi}@ict.ac.cn*

This article is an extension of an earlier version appearing in Proceedings of 2013 International Symposium on High Performance Computer Architecture (HPCA'13) [1]. Corresponding author: Haibo Chen.

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2015.2469657, IEEE Transactions on Cloud Computing

daily work [8], [9]. Those curious or even malicious cloud operators who can easily gain the full control of the cloud may unrestrictedly inspect or tamper with tenants' code or data, by either physical [10], [11], [12] or software attacks [7]. Furthermore, it is possible to recover data from residues of off-power memory [10], [12]. This situation will be even worse if the replaced memory has become non-volatile (e.g., phase-change memory [13]).

With the whole hardware and software stack of a multi-tenant cloud being controlled by cloud operators, tenants will likely be forced to assume a strong adversary model that trusts only a small part of the cloud. Existing software-based approaches such as CloudVisor [14] remove the hypervisor from the trusted computing base (TCB), but cannot guard against physical attacks like bus snooping and cold-boot attack[10]. Similar to CloudVisor, hardware proposals including SecureMMU [15], H-SVM [16] and HyperWall [17] leverage architectural support to enhance the memory management units to isolate a VM's memory from the hypervisor [1]. However, they require changes to OS and cannot defend against physical attacks as well.

In order to further reduce the TCB size and protect computation outsourcing even against physical attacks, in this paper we propose a secure framework, named Kite, to provide the tenants full control over the outsourced virtual appliance. "Full control" here has three aspects: *privacy*, *integrity* and *control of execution*. To ensure privacy and integrity, neither the data nor the executing code could be leaked to or tampered with by anyone other than the tenants themselves. Control of execution means that the tenants should be able to determine the way of execution of the outsourced computation, e.g., execution quota, time period constraint, revocation, etc.

Kite leverages both software and hardware to provide strong and transparent VM protection in a multi-tenant cloud. It adopts "secure tamper-resistant processor" on the host platform, while assuming all other hardware components (device and main memory) as untrusted to defend against physical attacks. Unlike previous secure processor based approaches which mainly focus on application-level protection and require a non-trivial change of OS [18], [19], [20], [21], or applications [22], or both, Kite aims at protecting the entire VM without any modification to the guest OS.

There are two major challenges to the design of Kite. First, although the secure processor can protect the privacy and integrity of code and data on the chip, it is not expressive enough to capture and handle complex high-level semantics between the processor and VMs, which is known as *semantic gap*. For example, the processor has no idea on which data should be transferred from guest VM to the hypervisor for a particular scenario of *trap-and-emulate*. Simply adding the related processing logic into the processor will make it unacceptable complex, while requiring guest OS to explicitly share interactive data with the hypervisor leads to non-trivial modifications to both software.

Second, in a virtualization environment, even if a secure processor is used, a malicious hypervisor can still issue attacks through legal cloud operations or hypervisor interfaces, like rollback attack [23], [24] or Iago attack [25]. For example, an attacker may launch a brute-force attack to guess the login password of a VM. Even if the guest OS has restriction on the number of failed trials (e.g., blocking for a while after three times, or erasing all data after ten times), the attacker can still infinitely rollback the VM to an initial state after each trial to clear the counter inside the VM and bypass the restriction. Another example is that a malicious hypervisor may manipulate the entropy sources (e.g., device events) to weaken the randomness of random numbers used by guest VM, which can be used to further break the TLS (Transport Layer Security) protection used by the VM [23].

To bridge the semantic gap between a VM and the secure processor while keeping the transparency of protection, Kite takes a novel approach that lets the secure processor provide security-enhancing mechanisms, while leaving the handling of most virtualization-specific semantics in a small piece of software, named VM-Shim[2]. Kite provides both hardware support for running VM-Shim and a specification of interactive data for communication between the hypervisor and VM-Shim. The implementation of VM-Shim software can be various as long as it follows the specification. It also has small code size thus is amenable for formal verification. The VM-Shim software could be implemented by the processor manufacturer, a third party open source organization, or even the tenants themselves.

To protect the execution of guest VMs and defend against rollback attack and Iago-like attacks, we investigate all the VM operations and the semantic of virtualization interface, analyze the attack surface and possible exploits, and propose several defending technologies. We introduce *secure logging* mechanism to make VM operations auditable, and *sealed policies* mechanism by leveraging condition servers to enforce tenant-defined policies to control the execution of VM. We also show possible Iago-like attacks and defense solutions.

To demonstrate the applicability of Kite, we have implemented a prototype in a QEMU-based full-system emulation environment with the Xen VMM (Virtual Machine Monitor). The VM-Shim consists of around 1,200 LOCs (Line-Of-Code) and requires modification around 430 LOCs to the VMM, which shows that the VM-Shim can be easily implemented with modest code size. The performance measurement shows that the overhead is small.

In summary, this paper makes following contributions:

- The first hardware-software framework that achieves full-control of outsourced computation by tenants

1. While the virtualization stack contains both a management VM, zero or more driver VMs and the hypervisor, other than specially mentioned, we uniformly call them all together as the hypervisor for presentation clarity in this paper.

2. Shim is common software mechanism for application adaptability in computing [26], we use the name VM-Shim as it adapts a VM to Kite without the requirement to change the guest OS.

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2015.2469657, IEEE Transactions on Cloud Computing

through *transparently* protecting guest VMs against an untrusted hypervisor and even physical attacks.

- The VM-Shim mechanism that provides a scalable and transparent approach to protecting VMs on commercial off-the-shelf virtualization stack.
- The mechanism that offers cloud tenants the full control over the execution of outsourced computation.
- A prototype implementation and evaluation in both a QEMU-based full-system emulation environment and real hardware platform with the Xen VMM, which is demonstrated with low performance overhead.

2 Background and Related Work

2.1 Securing Computation Outsourcing

Previous research on securing computation outsourcing usually focused on proving the correctness of the computation, by using technologies such as computation verification [27], [28]. Such methods usually focused on specific domains, e.g., biometric comparison [29], encryption algorithm [30], algebraic computation [31], [32], etc. Fully homomorphic encryption scheme is another hot topic in this area [33]. However, the performance overhead is still the most significant limitation. There are also some other research focused on securely outsourcing data storage without losing control [34], [35], [36]. In this paper, we use virtual appliance as the way to outsource *general* computation, and ensure the security of such computation by protecting its privacy, integrity and the control of execution.

2.2 Virtualization & VM Protection

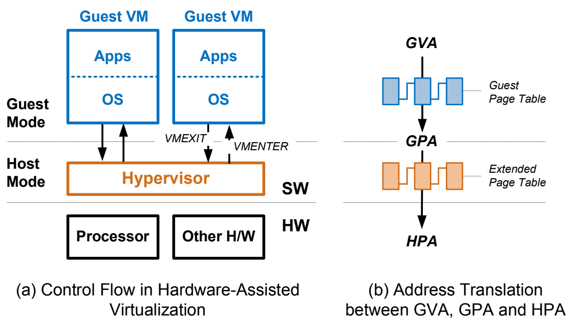


Fig. 1: Hardware-Assisted Virtualization

Hardware-assisted virtualization [37] has now been a standard feature in desktop and server platforms. For example, x86 processor virtualization introduces a "host mode" to run hypervisor and a "guest mode" to run VMs. When a VM executes a privileged operation, it gets trapped from guest mode to host mode, which is called a "VMEXIT". The hypervisor then handles the VMEXIT according to different exit reasons, e.g., I/O operations, privilege instructions execution. Then it resumes the trapped VM by issuing VMENTER instruction. Figure 1-a shows the process. To encapsulate a VMs CPU context, there is also an in-memory VM control structure (VMCS) for each virtual CPU, which encapsulates the CPU context for both the VMs (VM context) and the hypervisor (hypervisor context). The VMCS is saved by processor during VMEXIT and is used by the hypervisor to handle the VMEXIT and resume a VM's execution. Figure 1-b shows the address translation process in virtualized platforms. Guest application uses guest virtual address (GVA), which is translated to guest physical address (GPA)[3] through a VM's page table. GPA is further translated to host physical address (HPA) through extended page table (EPT) maintained by the hypervisor. GPA is a continuous memory space from a VM's perspective, but can be mapped to discontinuous HPA space.

The commercial success of virtualization and multi-tenant cloud and the lack of security guarantees for tenant's data have generated considerable interests to the research community to improve the cloud trustworthiness and security. On the software side, NoHype [38] advocates space-partitioning cores, memory and devices to a VM, detaching the virtualization layer during a VM's normal execution time. This reduces the attack surfaces for a VM as the VM is physically isolated from other VMs as well as the management VM for most of the time. Compared to NoHype, Kite assumes a stronger adversary model that further considers physical attacks, while NoHype only considers software attacks and cannot guard against sophisticated attacks such as inspecting a VM disk, bus snooping and memory freezing. Further, Kite still retains most functionalities in a commercial hypervisor like time-multiplexing resources, which are currently absent in NoHype. CloudVisor [14] separates the security protection from resource management and leverages a tiny nested hypervisor to encrypt and check the integrity of VMs. Excalibur [39] enables sealed-policy data abstraction by leveraging nested hypervisor to check the running context before data being used. However, these systems do not defend against physical attacks and still require means to secure the nested hypervisor, which may suffer from single point of security failure.

On the hardware side, H-SVM [16] and HyperWall [17] also separate the management of memory resources from the security protection, but without the need of a nested hypervisor. Instead, H-SVM uses microcode programs in hardware to enforce memory protection. HyperWall introduces CIP (Confidentiality and Integrity Protection) tables to do memory isolation. However, they require non-trivial changes to the guest OS as well as the hypervisor. Both systems store plain text data in memory and thus are vulnerable to physical attacks. Finally, major cloud maintenance operations like VM snapshot/restore are disabled since the hypervisor cannot access any protected memory. Live VM migration must be done with the assistant of guest OS, which needs further modification of the guest OS.

Similar to prior H-SVM and HyperWall, Kite also assumes an untrusted hypervisor. However, Kite does not trust the external memory or devices and uses memory encryption and integrity verification to protect off-chip data. Further, Kite captures complex VM interactions and

3. In this paper, we denote the guest physical address as the pseudo physical address seen by the guest VM, and the host physical address as the real physical address in a machine.

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2015.2469657, IEEE Transactions on Cloud Computing

data exchanging in a VM-Shim to retain OS transparency. Finally, Kite is designed to still support existing cloud maintenance operations (section 6.1).

	OS Transparent	**TCB Size**	**Phys. Attk.**	**Cloud Func.**
HyperWall	No	CPU + Mem + IOMMU	No	Part
H-SVM	No	CPU + Mem + IOMMU	No	Full
CloudVisor	Yes	HW + CloudVisor (5.5K LOC)	No	Full
Kite	**Yes**	**CPU + VM-Shim (1.2K loc)**	**Yes**	**Full**

TABLE 1: Comparison of Related Systems

Similar to Kite, Overshadow [40] also leverages the concept of shim [26], but mainly uses it to adapt the semantics of system calls between a hypervisor and guest OSes, which is much more complex than the VM-Shim in Kite. Further, the shim mechanism in Overshadow is OS-specific and requires completely different shim implementations for different OSes. In contrast, the VM-Shim mechanism in Kite is portable among different OSes and is much simpler.

2.3 Secure Processor

Secure processor has been extensively studied during the last decade [18], [19], [20], [41], [42], [43], [44], [45], [46], [47], [22], [21]. For example, Bastion [22] and SecureME [21] also leverage secure processor to protect against hardware attacks. However, they both need to trust the hypervisor, while Kite only trust the processor chip. Further, unlike Kite, they break application/OS transparency in requiring either non-trivial OS modification [21] or refactoring applications into modules for protection. In contrast, Kite retains OS and application transparency by leveraging the VM-Shim mechanism. For secure processor designs, we adapt two major techniques: AISE (Address Independent Seed Encryption) based data encryption and Bonsai Merkle Tree (BMT) [45] to secure off-chip data. Briefly, AISE and BMT are used to ensure privacy and integrity accordingly. More details could be found in [45].

2.4 Threat Model

In Kite, neither the virtualization software stack nor physical environment is trusted, as well as the interface between the hypervisor and VMs, resulting in a strong adversary model and a minimized TCB. We also consider a malicious cloud operator who is able to control the execution of a VM, including booting, rebooting, suspending and resuming at any time. However, there are three kinds of attacks that are not considered in this work. First, an adversary may still have the opportunities to subvert a VM by exploiting the security vulnerabilities inside a VM. How to harden the VM itself is out of the scope of this paper. Second, Kite provides no guarantee on the availability of deployed virtual appliances, which means that a tampered or abused hypervisor might be able to provide degraded or wrong services to a tenant VM, or even refuse to serve tenant's VMs. Third, we do not try to prevent side-channel attacks in the cloud [48], [49], which are usually hard to deploy and have very limited bandwidth to leak information. However, Kite ensures that an adversary controlling a subverted VM cannot further break other VMs through the tampered hypervisor or even abused hardware.

3 Design Overview

Figure 2 shows an overview of Kite. Kite contains three parts: the first part (section 4) leverages traditional secure processor technology and extends it to support virtualization environment. Different VMs are isolated with each other since they use different encryption keys. Kite uses AISE for encryption, BMT for integrity check, and introduces VM-Table for multiplexing. Tagged-cache and EPT protection are proposed to protect the security of memory mapping within and between VMs.

The second part of Kite (section 5) is a hardware-software mechanism named VM-Shim, which is introduced to retain OS transparency and only reveals necessary information of VM to the hypervisor. VM-Shim is further composed of two aspects: 1) hardware support to enable control interposition between the hypervisor and VMs, and 2) a specification of interactive data between the hypervisor and VMs, which contains CPU context, I/O data and auxiliary information. The hypervisor and VM-Shim instance use the specification to communicate with each other. The software implementation of VM-Shim only depends on the specification. Meanwhile, the hypervisor does not need to trust the software of VM-Shim since each VM-Shim instance runs with the VM and is isolated from each other.

The third part of Kite (section 6) aims to ensure the control of VM execution. In order to defend against control-flow attacks such as VM *rollback attack* and service abusing, while retaining normal VM operations, Kite uses *secure logging* to allow the tenant to distinguish between benign and malicious operations to detect rollback attack. It also introduces *sealed-policies*, which are tenant-defined code in virtual appliances to update and check execution conditions, to precisely control the execution of VMs or even revoke them from the cloud.

The TCB of Kite contains only the secure processor and the VM-Shim. The processor which can be easily verified by its public key through a well-known certificate authority. The VM-Shim can be verified by its source code, or be provided by the tenants. It is the tenants' responsibility to update their VM-Shim implementation by recreating and redeploying the entire VM images including the new VM-Shim. The tenants don't need to trust any software component from the cloud provider. It also maintains backward compatibility to guest OS and requires minor change to the hypervisor.

4 Architectural Support

4.1 Secure Processor Integration

AISE and BMT for Memory Protection: Kite adopts AISE and BMT to protect memory due to their low overhead (as mentioned in section 2.3). One difference from

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2015.2469657, IEEE Transactions on Cloud Computing

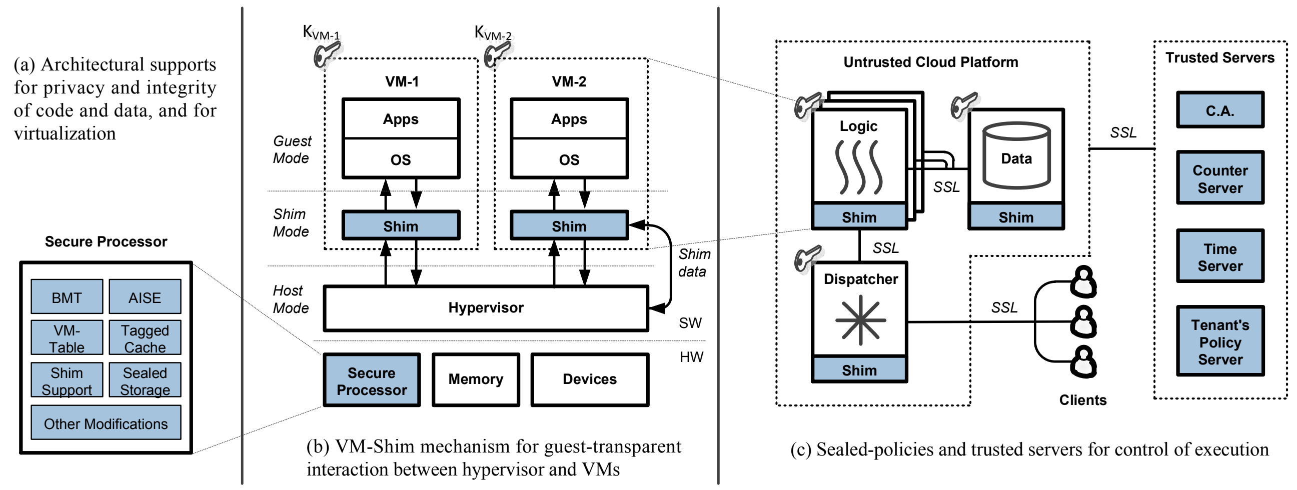


Fig. 2: Overview of Kite architecture

New Instruction	Environment	Instruction Semantic
vector_install, addr1, addr2	Hypervisor	Install *vm_key* (addr1) and *vm_vector* (addr2). Return VMID. Update hash in NVR-1
vector_uninstall, VMID	Hypervisor	Remove the *vm_vector* indexed by VMID from the VM-Table
vector_dump, VMID, addr	Hypervisor	Encrypt the *vm_vector* indexed by VMID and store it to memory. Update hash in NVR-1
ept_st, addr, val	Hypervisor	Update data in EPT memory. Invalid cache only if an GPA_2_HPA mapping is modified or deleted
nvr1_get, val, addr	Hypervisor	Use the *val* as a nonce to XOR with the NVR-1, sign the result with PK_{cpu} and write it to *addr*
VMENTER (modified)	Hypervisor	Resume VM-Shim instead of the VM
VMEXIT (modified)	Guest VM	Transfer control to VM-Shim instead of the hypervisor
raw_st, addr, val	Shim/Guest	Store data into memory without encryption
raw_ld, enc_on, addr	Shim/Guest	Load data without integrity check. Use enc_on to control encryption engine on or off
rand_gen, reg	Shim/Guest	Generate a random number and put it into the *reg*
shim_to_host	Shim	Trigger VMEXIT and switch to host mode
shim_to_guest	Shim	Switch to guest mode and resume VM

TABLE 2: New Instructions in Kite

Key	Context	Protection
K_{vm}	per VM	Encrypt VM memory and disk image
K_{mem}	per Chip	Encrypt CPU reserved memory
SK_{cpu}/PK_{cpu}	per Chip	Private/public key pair of a CPU
SK_{app}/PK_{app}	per VM	Private/public key pair of a VM

TABLE 3: Keys involved in Kite

traditional AISE and BMT is that in Kite the processor uses GPA to index counters and hash values, instead of using HPA, since the memory of a VM is not physically continuous. Although a malicious hypervisor has the control over mapping from GPA to HPA, it still cannot tamper with guest's data, counter or hash because the root of the BMT is securely protected, as shown in figure 3. We add a per-core guest-TLB (g-TLB) tagged with VMID to assist address translation of hashes and counters from GPA to HPA, to avoid affecting the main TLB.

CPU Context Protection: CPU context is also properly protected when execution transfers from a VM to the hypervisor or other VMs. The processor encrypts the context data and leverages hash to protect its integrity, thus the hypervisor cannot access or tamper with VM's context. One exception is the *virtual interrupt vector* field, which is used to deliver interrupt from device to VM. For other fields, VM-Shim will offer minimal necessary fields of CPU context to the hypervisor, according to the semantics of different VMEXIT reasons, which will be detailed in section 5.2.1.

Tagged-cache for Inter-VM Cache Isolation: Since data is not encrypted inside on-chip cache, it might be vulnerable to inter-VM remapping attacks. A malicious VM can map some physical memory of a victim VM (with the help of a malicious hypervisor) and access the data with a cache hit, thus bypasses the encryption engine. In Kite, each VM is assigned a unique VMID, and each cache line is tagged with its owner's VMID. This ensures that a VM can only access cacheline with its own tag. VMID is the index of a VM in VM-Table, as stated in section 4.2.

EPT Protection for Secure Address Mapping: The extended page table (EPT) of VMs are fully controlled by the hypervisor for memory management. However, a malicious hypervisor may issue *intra-VM remapping attack* by changing GPA to HPA address mapping without invalidating cache. Thus, if the mapped data is in cache, no integrity check will occur and the memory integrity of the VM is violated. Kite addresses this issue by mandating that all the EPTs are stored within a specific memory region named *EPT memory*. A new instruction, *ept_st*, is introduced as the only way to modify *EPT memory*, which triggers cache invalidation when GPA to HPA mapping is changed. We observed that hypervisor usually updates the entire EPT table in a batch, before flushing TLB

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2015.2469657, IEEE Transactions on Cloud Computing

to enable the new mapping. To avoid unnecessary cache invalidation, e.g., during VM launching and destroying, Kite delays invalidating cache when updating TLB. When the hypervisor is modifying EPT using *ept_st*, the processor sets a bit flag to indicate that EPT has been updated. Each time there's a TLB miss or TLB flush, the processor invalidates cache if the flag is set, and then clears the flag. Nevertheless, EPT updates are rare during VM execution.

4.2 VM-Table for Multiplexing

Each VM has an entry in VM-Table that contains information necessary for AISE and BMT engines:

- $VMID$ is a unique identifier for each VM, which is the index of VM slot in the VM-Table. The range of VMID is large enough for the number of running VMs.
- K_{vm} is the encryption key for that VM during runtime.
- *vm_vector*, which includes following items:
  - H_{mem} is the root hash of the VM's BMT.
  - H_{ctx} is the hash of the CPU's context of the VM.
  - $Addr_{cnt}$ and $Addr_{BMT}$ are starting addresses (GPA) of counters and BMT in memory.
  - $Addr_{shim}$ is the starting address (GPA) of VM-Shim.

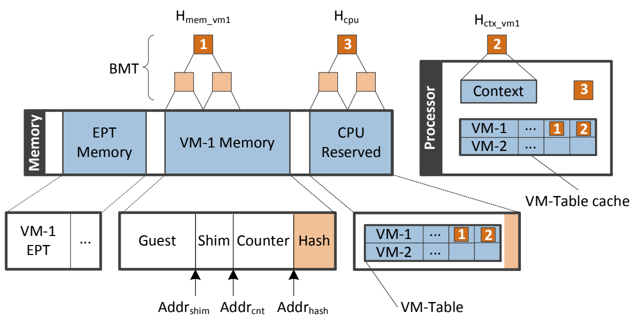


Fig. 3: The root hash of guest BMT is stored in VM-Table. VM-Table is in CPU-reserved memory region, whose root hash is stored on-chip.

In our design, the VM-Table is stored in a CPU-reserved portion of physical memory, which is also protected by AISE & BMT. This portion of memory is accessible only to the secure processor itself. A separated key, K_{mem}, is used for encryption and BMT, which is generated randomly when the processor is powered on, and is securely stored inside the processor. Since the VM-Table contains root hashes of VMs' BMT, it further ensures the integrity of VMs' memory space, as shown in figure 3. By using memory to store the VM-Table, we can save expensive on-chip storage. Meanwhile, the number of VMs running concurrently can be proportional to the memory size. We also introduce on-chip cache for VM-Table entries to optimize the performance. Three new instructions are introduced to operate on the VM-Table, i.e., *vector_install*, *vector_uninstall* and *vector_dump*, as listed in table 2. More details of the VM operations will be described in section 6.1.

4.3 Summary of Architectural Extension

The architectural support of Kite includes the integration of secure processor with existing hardware virtualization mechanism, as well as the VM-Shim support. This section presents the former part, while the VM-Shim part will be described in next section.

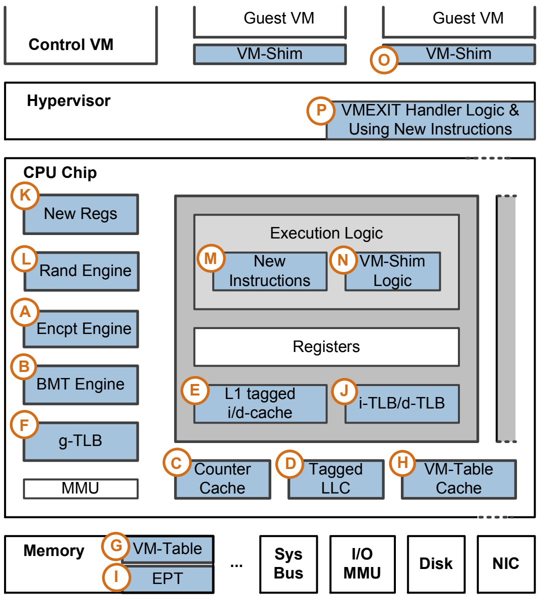


Fig. 4: Hardware/software Modifications for Kite

Figure 4 shows the hardware and software components of Kite. The secure processor substrate includes AISE encryption engine Ⓐ and BMT engine Ⓑ. Counter data is accessed through *split counter cache* Ⓒ, while hash data shares cache with ordinary data. Each cache line Ⓓ Ⓔ is tagged with VMID of its owner VM, and data in a cache line can be accessed only by its owner. Since both counters and hashes are indexed by GPA, a g-TLB Ⓕ is added to optimize address translation from GPA to HPA.

A VM-Table is introduced to store protection information of currently running VMs. Each entry contains $\{VMID, K_{vm}, H_{mem}, H_{ctx}, Addr_{cnt}, Addr_{BMT}, Addr_{shim}\}$. The VM-Table is stored in a CPU protected memory region Ⓖ, and most recent used entries are cached on chip Ⓗ. Three new instructions, *vector_install*, *vector_dump* and *vector_uninstall* are introduced to operate the VM-Table. The EPT memory Ⓘ is used to store EPT, and can only be modified by *ept_st* instruction, which triggers invalidation of cache when address mapping is changed to defend against intra-VM remapping attack, as stated in section 4.1. The invalidation is delayed to TLB update Ⓙ for optimization.

Registers Ⓚ of Kite include two non-volatile registers (NVR). NVR_0 is used for generating LPID to ensure that LPID is unique for each page, even after system rebooting. NVR_1 is used for logging that gets updated every time a VM is booted/resumed or a snapshot is made, triggered by *vector_install* and *vector_dump*, respectively. Kite also adds an engine for random number generation Ⓛ, which

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2015.2469657, IEEE Transactions on Cloud Computing

is used by the new instruction *rand_gen*. All of the new instructions Ⓜ are listed in table 2.

In order to support the VM-Shim mechanism, the logic of mode switching Ⓝ is changed to enable VM-Shim running in-between the hypervisor and a VM. In addition, two new instructions are added to switch mode from VM-Shim to host or guest, by *shim_to_host* and *shim_to_guest*, respectively. VM-Shim Ⓞ is responsible to exchange data between the hypervisor and VMs, by using two new instructions: *raw_ld* and *raw_st*. Meanwhile, the hypervisor Ⓠ needs to be modified to access guest's data through the interface provided by VM-Shim. Thus the guest OS can remain unchanged.

5 VM-SHIM MECHANISM

5.1 VM-Shim Mode

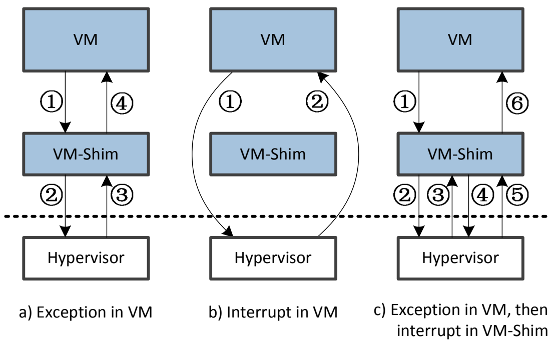


Fig. 5: VM-Shim Interposition

A VM-Shim has its own running context. It shares the same K_{vm} and BMT with the corresponding VM and can access the CPU context and all the memory of the VM. It also reserves a memory region that the VM cannot access. A VM-Shim has no permission to read or write memory of other VMs or VM-Shims. In order to enable data exchanging between VMs and the hypervisor, Kite provides two new instructions: *raw_ld* and *raw_st*, as described in table 2.

A VM-Shim interposes the control transition between a VM and the hypervisor. Adding a "man-in-the-middle" also introduces the reentry issue. For example, if a hardware interrupt occurs when a VM-Shim is running, the processor will enter the same VM-Shim again. One way to handle this is disabling hardware interrupt when a VM-Shim is running. However, as a VM-Shim runs in the context of a VM, it should not be granted with the privilege to turn on/off CPU interrupt. Otherwise, a malicious VM can easily freeze the whole system by disabling all interrupts.

Kite solves this problem by dividing events causing a VM trap into two cases: synchronous events caused by exception, and asynchronous events caused by interrupt. Figure 5-a shows the handling process of exception-caused VMEXIT:

- ①: The processor transfers control from a VM to its VM-Shim.
- ②: VM-Shim prepares the data needed by the hypervisor according to the semantics of different VMEXIT reasons, and transfers control to the hypervisor through *shim_to_host*.
- ③: the hypervisor handles the VMEXIT, exchanges data with VM through VM-Shim's memory region if needed, and issues VMENTER.
- ④: the VM-Shim copies data from the hypervisor (if any) to the VM's memory space and CPU context. Finally it resumes the VM's through *shim_to_guest*. More details on data interaction between a VM and hypervisor is in section 5.2.

The second case is interrupt-caused VMEXIT. Since it is an async event, in order to prevent the re-entry problem, the processor skips VM-Shim, as shown in figure 5-b. Since the VMEXIT is not caused by a VM, the hypervisor does not need the guest's information. However, it may still deliver a virtual interrupt to a VM by setting flags on guest's *virtual interrupt vector*, which is a part of the VM's CPU context. As we mentioned in section 4.1, the *virtual interrupt vector* is not protected, thus the hypervisor can modify it directly. A malicious hypervisor could not issue attack through *virtual interrupt vector*, since the semantic of the field is limited.

Once an interrupt-caused VMEXIT occurs when VM-Shim is running, as shown in figure 5-c, the processor will save the context of VM-Shim (step ②) and transfer control to the hypervisor. When hypervisor finishes handling VMEXIT, it resumes the VM-Shim from where it is interrupted (step ③), then the VM-Shim continues execution as normal. The steps ④, ⑤ and ⑥ in figure 5-c are similar as steps ②, ③ and ④ in figure 5-a, correspondingly. Meanwhile, the VM-Shim itself will not trigger exceptions itself. First, the VM-Shim avoids to use any privilege instructions that would cause VMEXIT. Second, the hypervisor must pin the memory used by the VM-Shim to avoid page fault. Double fault will be treated as fatal error.

5.2 Interactive Data Specification

When a VMEXIT occurs, the VM needs to provide the hypervisor minimal yet sufficient information to correctly handle the VMEXIT. When the hypervisor finishes, it sends the results back to the VM. There are different types of VMEXIT, each type requires different sets of interactive data. The VM-Shim specification defines the data sets for both the VM-Shim and the hypervisor as an interface. A VM-Shim logically divides its memory into two portions: a protected (encrypted) memory area and an unprotected (plain) memory area that assists the interactive data. It uses *ld_raw* and *st_raw* instructions to transfer data between the two memory regions. The interactive data contains three parts: CPU context, I/O data and auxiliary data.

5.2.1 CPU Context

Data in CPU context, including registers in VMCS and general registers, cannot be accessed by the hypervisor

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2015.2469657, IEEE Transactions on Cloud Computing

directly. Instead, VM-Shim is responsible to exchange data between the hypervisor and VM. For different types of VMEXIT, VM-Shim only allows necessary context fields to be accessed by the hypervisor.

5.2.2 Disk I/O

Data on disks is encrypted with the same way as that in memory. Hence, no encryption or decryption is required during disk reading or writing. However, to protect against replay attack, VM-Shim needs to do integrity checking and maintain the merkle hash tree to ensure the integrity of disk I/O data. The hashes are stored in the same VM image file with VM data. During VM booting, VM-Shim is required to cooperate with the hypervisor to fetch all the hash value in non-leaf nodes and keeps them in memory. During DMA of disk read, the disk copies both the data and counter into guest's memory. The VM-Shim then uses *ld_raw* with decryption enabled, and checks the loaded disk data by calculating its hash value.

5.2.3 Network I/O

Network I/O is handled differently from disk I/O as Kite should not send an encrypted version of data to the communicating peer (e.g., a web client), which usually does not have the key to decrypt the data. As typical security-sensitive applications usually do application-level encryption like SSL, Kite, like other similar systems [14], [40], does not protect data sent out through network.

VM-Shim interposes network I/O to exchange the data. When data is read from a NIC device, it is first copied to a shadow buffer in VM-Shim. The VM-Shim then loads the data using the *raw_ld* instruction to the VM's buffer based on the I/O request. When data is written to a NIC device, VM-Shim uses the *raw_st* instruction to send data in plain-text.

Note that even for direct assignment or single-root I/O virtualization (SR-IOV) NIC devices, current virtualization hardware can still trap the I/O operations into the hypervisor. In Kite, a guest OS can also support SR-IOV by developing a NIC driver for optimization. More specifically, the driver of guest OS needs to be modified to use raw_st and raw_ld instructions to exchange both meta-data (e.g., I/O command) and raw data between the processor and the device. Thus the shadow buffer is not needed and the I/O performance can be improved.

5.2.4 Auxiliary Information

There are cases where the data to be exchanged are not present in the VM. For example, when a hypervisor needs its VM's page table entries to do address translation, the VM's page table entries might not be present. This requires cooperation among the VM-Shim, the VM and the hypervisor to handle such cases. In the followings, we will use a relative complex instruction from x86 (e.g., *rep ins io-port mem-addr*) as an example to show how the VM-Shim handles it.

The instruction mentioned above repetitively load data from disk to memory, with the repetition number being indicated in the *%ecx* register, the disk I/O port being specified in *io-port* and the starting memory address (in the VM) in *mem-addr*. In most commercial hypervisors, the I/O instruction will cause a trap to the hypervisor, which gets the virtual address of the instruction pointer (IP). Then the hypervisor needs to translate address of both IP and the target memory address from GVA to GPA by walking the VM's page table. However, it is possible that the target memory region starting from *mem-addr* might not be aligned and might cross multiple pages. In this case, the hypervisor needs to inject a page fault to the VM to let the VM fill the translation.

The VM-Shim interposes the above process and exchange the data with the hypervisor. VM-Shim avoids the need of guest page table walking for decoding the I/O instruction by fetching the opcode in the VM context during the trap. On interposing the VM trap, VM-Shim proactively translates the *mem-addr* from GVA to GPA by walking the VM's page table. It then puts the plain-text version of addresses to memory using *st_raw*. If a translation cannot be done due to the absence of page table entries, VM-Shim just puts an invalid entry. When the hypervisor starts to execute, it fetches the addresses VM-Shim puts. If necessary address translation is absent, the hypervisor will again inject a page fault to the VM, which will resolve the fault and retry the I/O instruction. The retrying will again trap to VM-Shim first, which can now do the translation and put the obtained address translation to make the hypervisor be able to emulate the I/O instruction.

Similarly, the VM-Shim exchanges a VM's data to the hypervisor and external environment according to the context. It completely eliminates the need for the hypervisor to access a guest VM's memory and also makes it easy to reason about each data interaction.

6 Control of VM Execution

Controlling the execution of an outsourced computation is a goal orthogonal to privacy and integrity of the computation. In Kite, we address this problem by leveraging two technologies. The first is *secure logging* supported by the architecture, which aims to defend against rollback attacks. It logs all of the VM operations in a safe way and thus allows the tenant to distinguish between legitimate VM operations and rollback attacks. The second is *sealed policies* within a VM, which is used to prevent service abusing. Such policies are implemented as a portion of code by the tenants themselves that checks specific conditions during the execution of the service, e.g., maintaining and checking a counter to control the times of running, checking current date to ensure not exceeding expiration period, etc. The services will run only if all the checks pass. A tenant can also leverage sealed policies to enable virtual appliance revocation.

6.1 VM Life Cycle

This subsection introduces the life cycle of a VM as well as all the legitimate operations. As mentioned in section 4,

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2015.2469657, IEEE Transactions on Cloud Computing

the life cycle of a VM depends heavily on the *vm_vector*. Kite offers new instructions to insert or remove a *vm_vector* into or from the VM-Table, or dump a encrypted *vm_vector* to memory for future use, as figure 6 shows. All the VM operations, including bootup, shutdown, snapshot, restore and migration, will leverage these new instructions.

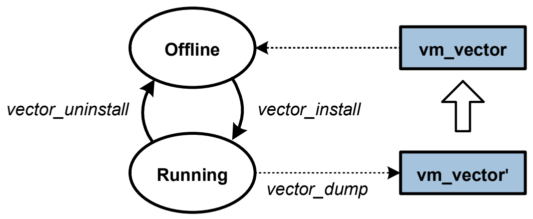


Fig. 6: VM life cycle. Vm_vectors are encrypted by K_{vm}.

VM Deployment: Tenant deploys applications as virtual appliances on untrusted cloud. When deploying, the owner of the VM needs to offer following components, which are generated offline by tenants using our provided tool:

- A disk image of the VM, which is encrypted as section 5.2.2 described. The metadata of the disk includes the starting address of the counter zone and hash zone.
- An initial memory image of the VM. The image contains logic of VM-Shim and is formatted by generating counters and BMT and encrypting the data part using K_{vm}. The memory image also contains the root hash of BMT of the disk in the VM-Shim. A wrong formatted or tampered image will be denied by a secure processor.
- A context of the processor, which contains the program counter pointing to the first instruction to be executed.
- A *vm_vector*, which is encrypted by K_{vm}. The vector summarizes the initial VM memory image and is used by the processor to verify the image. It also contains the hash of CPU context, as shown in figure 7.
- K_{vm}, which is encrypted by the SK_{cpu} of host chip.

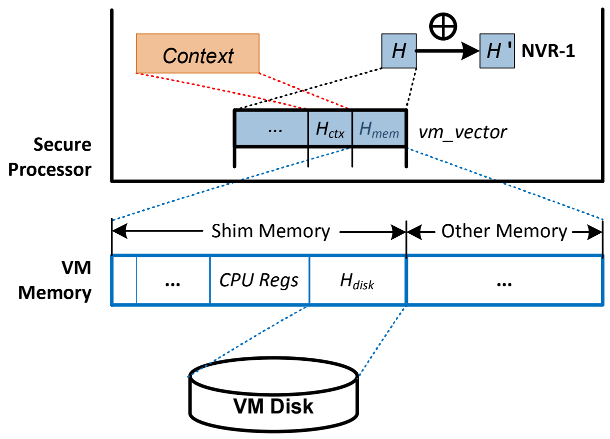


Fig. 7: The vm_vector contains the hash of CPU context, and the root hash of memory that further contains the root hash of the disk image

VM Bootup: The process of VM booting includes following steps:

1) The hypervisor allocates memory pages for the VM, initializes its page tables, and loads the VM's initial memory image into the allocated pages.
2) The hypervisor invokes *vector_install* instruction and passes the encrypted K_{vm} and *vm_vector* as arguments.
3) The secure processor allocates a slot in the VM-Table, and decrypts the K_{vm} and *vm_vector* into the slot. It then returns the slot index as VMID to the hypervisor.
4) The hypervisor then issues VMENTER with the VMID to start the VM. The CPU context is also provided as the parameter of VMENTER.

Now, all the essential information for booting the VM are now ready. Since the BMT root hash of the disk image is already in the memory of VM-Shim, each disk read can be checked to ensure disk data's integrity.

VM Shutdown: When a VM is shutting down, the guest OS does not have to zero all the memory pages, since they are encrypted already. The VM-Shim will then enter a function to reset its status, and prepare to go to the initialization code at the next VMENTER. After that, the hypervisor will execute *vector_dump* to get the *vm_vector*, and dump the memory including VM-Shim. Finally, it needs to execute *vector_uninstall* to revoke the slot of VM-Table. Next time, when the hypervisor boots the guest VM, it has to use the *vm_vector*, the memory dump, the corresponding disk image. In order to tolerate abnormal shutdown, such as those caused by power outage, the cloud provider needs to maintain at least one available set of the disk, vector and memory image.

VM Snapshot: When a hypervisor takes a snapshot of a VM, the process is similar with VM shutting down. It first saves the VM's current CPU context, memory data and disk data, all in cipher-text, then issues *vector_dump* to get the VM's *vm_vector*, which is encrypted by the K_{vm}. The *vm_vector* is later used to restore the snapshot by using *vector_install*, similar as booting a VM.

VM Restore: The VM restoring process is the same as the process of VM booting from the hardware's perspective. The hypervisor also needs to prepare the disk, memory image, CPU context and the corresponding *vm_vector*.

VM Migration: VM migration is similar to a combination of snapshot and restore. The target machine has already got the K_{vm} of the VM, which is encrypted using the chips' SK_{cpu} in advance. Thus, it can execute *vector_install* to install the encrypted *vm_vector*. The key distribution is done offline by the VM owner so that no key exchange is needed. A target machine is trusted if and only if it has the encrypted K_{vm}. The VM migration is done by the hypervisor without any involvement of VM-Shim.

6.2 Secure Logging

Secure logging is introduced to defend against *rollback attacks*. First, we define a *running epoch* of a VM as a period that starts from booting or resuming and ends with

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2015.2469657, IEEE Transactions on Cloud Computing

shutdown or suspending, as shown in Figure 8. The underlying secure mechanism assures following two properties:

- Starting a VM from the middle of an epoch is forbidden.
- All the beginning and end of each epoch are safely logged.

By defining and logging the running epoch of VM, the rollback process is auditable. A tenant can analyze the log to detect rollback attack, or to constrain the behavior of cloud providers to prevent rollback attack.

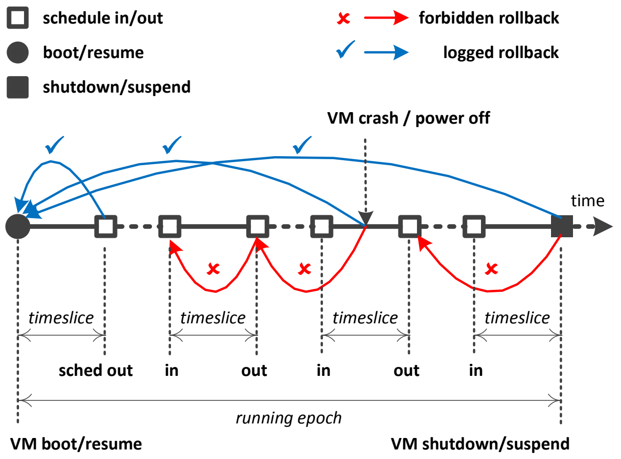


Fig. 8: Running Epoch of VM

The first property is ensured by the protection of processor context, as described in section 4.1. The hypervisor cannot change the program counter of a context, thus cannot change the control flow of a running VM. Otherwise, the processor will generate a warning if the hash of context doesn't match, and stop working.

To make the log tamper resistant, we leverage the secure processor to protect its integrity. Each VM operation will involve either *vector_install* or *vector_dump*. Each instruction will accumulate the hash of the *vm_vector* involved to the NVR-1 register, as shown in figure 7. The hypervisor is responsible to log all of the operations and *vm_vector*s involved. Thus, the tenant can recalculate the hash of the *vm_vector*s to check the integrity of the log.

Because a VM can be migrated or cloned to different physical machines, the log may be distributed on multiple machines. A malicious cloud operator may hide some log from tenant, which may contain evidence of rollback attacks. In order to keep the integrity of the entire log set, we need to record all the physical machines that have ever hosted the VM. Fortunately, the tenant has already had the set of hosts since at the deploying phase, the tenant needs to prepare one encrypted K_{vm} for each processor with its public key.

6.3 Sealing Policies in VM

In order to control the execution even after being outsourced, a tenant needs to seal some policies into the VM. Before a service beginning to run, the VM will first check to ensure that all the conditions specified by the policies are met. A condition could be a period, a counter, a simple flag, or any other ones defined by the tenant. The conditions cannot depend on the host machines, which are not trusted. Thus, we introduce the trusted *condition servers*, which are either deployed publicly such as CAs (Certificate Authority) or deployed by the tenants themselves in a safe place. The deployment is illustrated previously in figure 2-c.

There are two types of condition servers, one is *read-only* and the other is *read-write*. A read-only server is used to offer some information, such as the current time, as a condition. A read-write server could be used to save the status for each VM, and update the status according to the execution dynamically, e.g., a counter or a flag. In order to use the read-write server, a VM needs to register itself before making any request. Each VM has its own pair of public key and private key for attestation and identification. During the registration phase, a VM identifies itself with its public key, and asks for a condition. The server will allocate corresponding resource for the VM and accept requests from it.

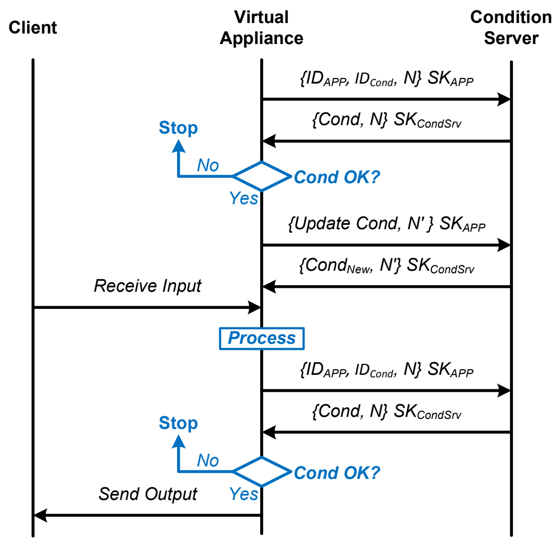


Fig. 9: Attestation communication protocol with condition servers

The process of condition check is shown in figure 9. As the figure shows, there are two checks for each processing, one is before the service starting, the other is before sending results to clients. The second check could ensure that no output will be delivered to clients if the condition is not met. However, a malicious cloud may issue timing channel attack by measuring the execution time of the service to infer some results. The first check could mitigate such threat by refusing to execute in the beginning.

Each check request contains the application ID, condition ID and a nonce generated by the new instruction *rand_gen*. The request is signed by the private key of the application. The condition server will then reply with the current condition and nonce in order to prevent replay attack. The reply is also signed by the server. The VM will update the condition and will not run the service unless the update is

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2015.2469657, IEEE Transactions on Cloud Computing

confirmed.

The sealing policies mechanism relies on the integrity of VM's control flow, which is ensured by the architectural support and secure logging. It also assumes that the condition servers are trusted and available. This mechanism can be used to implement following scenarios:

- **Lease-based execution**: The tenant seals a policy in the VM to check the current time, and continue the execution only if the time is before some threshold.
- **Counter-based execution**: The VM increase a counter for each execution, and check the counter to ensure it's less than a preset threshold. The counter is saved on a trusted counter server.
- **VM revocation**: The tenant adopts a condition server at a trusted place. The VM is sealed with a policy that check a flag on the condition server every time it runs a service. The tenant can easily revoke the VM as long as the flag is cleared on the condition server.

7 Application: Outsourcing Chip Validation Algorithm

This section revisits the case of outsourcing computation of IC chip validation algorithm mentioned in the first section and see how Kite can retain full control to the tenant. We assume that the validation algorithm is encapsulated in VAs that are deployed on Kite. The tenant also offers the validation as a service to other clients, and charge them by the times of service running. We also assume that a trusted counter server is available, and the VMs has sealed a policy that increases the counter every time the service is used.

A malicious operator may try to steal the algorithm and chip design data from the VM, or modify the algorithm to deliver a misleading result. In Kite, a hypervisor is able to access all of a VM's memory data. However, it can only get cypher-text and will cause BMT checking error if it tampers with the data. Meanwhile, since the data in memory is encrypted, the system can also defeat physical attacks that access memory directly, e.g., sniffing system bus or even using offline methods such as cold-boot attack. It is noted that even if an attacker has an emulator of the processor and run the whole VM image on it, the VM's data is still safe as long as the private key of processor is not obtained.

A malicious hypervisor may swap a VM's data from memory to the disk, tamper with the data, and swap back to memory. This attack can be defeated since the BMT protection is in the space of guest physical address, which means that a data block is protected by the BMT no matter it is in memory or disk.

One possible attack is inter-VM remapping attack, which can be made by a hypervisor with a collusive VM. Since data is decrypted in cache, a malicious hypervisor may map a VM's memory page to a collusive VM, and then expect a cache-hit when the bad VM accesses the page and thus bypasses the protection mechanisms. This attack will also fail since the cache is tagged with each VM's VMID. Thus, a VM cannot access another VM's data in cache.

The security of VM-Shim can also be ensured. The VM-Shim resides in the same protected context with its VM but is isolated from the VM: a VM-Shim has separated address space but is granted with access to arbitrary memory in its corresponding VM; oppositely, a VM cannot access the memory space of its VM-Shim. Each VM-Shim instance only does very little work and is usually quite small (in the scale of one to two thousands LOCs). Hence, it is relatively easy to formally verify its correctness. There is exactly one VM-Shim for each VM and each VM-Shim is protected by the processor. Hence, a security breach of a single VM-Shim cannot affect the security of other VMs.

Meanwhile, the VM-Shim is also non-bypassable. Each time there is a VM exception, the processor will immediately transfer control to the entry address of VM-Shim that is saved in the VM-Table. The memory of VM-Shim is also protected by the encryption and BMT. Meanwhile, the VM-Shim is trusted by each guest OS in our threat model. Any behavior of the VM-Shim is considered as following the tenant's intention.

Another possible attack could be issued by a malicious service client, who could bribe the cloud operator to hide the actual usage amount of the service and thus pay less. Such attack could be prevented by the sealed policy, which updates a counter saved on trusted a counter server every time it runs. The counter could not be tampered with, and could be retrieved by the tenant as a proof of actual usage.

8 Security Analysis

Besides the attacks listed in last section, a malicious hypervisor may also issue attacks by manipulating the return value of services it provides to guest VMs, similar as the Iago attack [25] from malicious OS to applications.

8.1 Defending Against Iago-like Attack

Unlike the complex interface between OS and application, the interface between hypervisor and guest VM is much simpler, which greatly reduce the attacking surface. However, by analyzing the semantic of all the interfaces and considering the usage of returned value, we still find several possible Iago-like attacks. Kite can well address all of these potential threats.

Memory-mapping attack: In Iago attack, a malicious OS may map different virtual memory pages to the same physical memory page. If one of the virtual pages belongs to application's stack, then modifications to another virtual page will also change the stack, which might further cause violation of control flow integrity. Similarly, in virtualized environment, a malicious hypervisor may map two guest physical pages to the same host physical page, and issue similar attack. Kite defends this attack by letting the user verify the integrity of initial memory image, which is protected by the initial *vm_vector*. As mentioned in section 4, AISE uses a different counter for each memory page, thus even if two pages have identical data, they are different after being encrypted on the memory, which prevents a malicious hypervisor from mapping them to the same

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2015.2469657, IEEE Transactions on Cloud Computing

physical memory page. Similarly, a malicious hypervisor may remap different pages of the same VM during runtime. If the pages are in cache, then the BMT check is bypassed and the integrity of VM's memory is violated. Kite defeats this attack by invalidating cache at the time of remapping (EPT modification), as stated in Section 4.1.

Random number generating: OS typically seeds its entropy pool from devices, such as keyboard and mouse input, disk I/O, network I/O, etc. In a virtualized environment, all of the devices are virtualized by the hypervisor, which should not be trusted as entropy sources. In Kite, we add a random number generator to the secure processor and provide an instruction *rand_gen* to serve the guest OS secure random numbers.

Timing info: A VM usually relies on hypervisor to get time information. Thus, a malicious hypervisor may return faked time value to the guest. If the guest uses time info to enforce some security policies, it may be fooled and its security may be violated. Kite solves this problem by leveraging trusted time server, as shown in figure 2, which accepts a nonce and returns a signed value containing the current time with the nonce. The guest VM can check the nonce and signature to ensure the freshness and validity of the time value.

Other attack surface: Most of the interface between VM and hypervisor is at hardware level, including memory, device, processor states, etc., which exposes a small attack surface to attackers. On one hand, in most VMEXITs the hypervisor just requires a few pieces of data, e.g., contents of control registers. Thus, the VM data exported through VMEXITs is very limited. On the other hand, the value returned by the hypervisor as a service is also at low-level and has little semantic, which is far from enough to mount a security attack. However, we still cannot prove that the Iago-like attacks are 100% eliminated on Kite. We plan to adopt interface-verification technologies such as InkTag [50] to Kite to strengthen the defense against Iago attack.

8.2 Other Security Issues

As discussed in section 2.4, Kite does not ensure the availability of a VM. A malicious hypervisor may slow down the execution of a VM by limiting its resource, or even stop it by not scheduling it at all. Meanwhile, it can also give a wrong result when processing VMEXIT. However, Kite ensures that these attacks on availability cannot get tenant's private data or tamper with the execution of a VM.

We currently adopted a fail-stop model in the paper. Before halting, the abnormal behavior will be logged, and the hash of log will be saved in a non-volatile register in the processor for later auditing.

Currently, we support multi-core chips, but not multi-chip processors. The challenge is that the data encryption mechanism used in Kite does not suit data exchanging among chips. SENSS [51] utilizes the Cipher Block Chaining mode of the advanced encryption standard (CBC-AES) for encryption/decryption of shared bus between chips. Supporting multi-chip processors and multi-processor will be our future work.

Encrypting all VM's memory prohibits content-based memory sharing and deduplication among VMs, as well as some VM introspection, since the hypervisor can only see encrypted version of VM's data.

9 Performance Evaluation

We implemented a working prototype by modifying QEMU full-system emulator to validate the applicability of Kite. A VM-Shim is implemented to support unmodified Linux and Windows VM, which can run both on a real machine and QEMU. In a real machine, it runs in the host mode together with Xen to simulate the control transitions among the hypervisor, VM and VM-Shim. A user-level agent consisting of 200 LOCs is implemented in the management tools of Xen to assist the fetching and storing of counters and hashes for Kite. The VM-Shim currently consists of around 1,200 LOCs. We also change around 230 LOCs in Xen to secure VM booting and cooperate with the data exchange mechanisms in VM-Shim.

9.1 Performance Evaluation on Simulator

As there is currently no cycle-accurate full-system simulator that can run a virtualized platform, we use QEMU as a full-system simulator to collect traces and dinero-IV [52] to do trace-replay. These benchmarks are run with reference input set. Each benchmark is simulated for 1-billion instructions inside the VM, after skipping 10-billion instructions. As the number of benchmarks is too large for exposure, we only report a set including astar, bzip2, gcc, lbm, libquantum, mcf, milc, sjeng, sphinx3, and h264ref, similar as prior work [21].

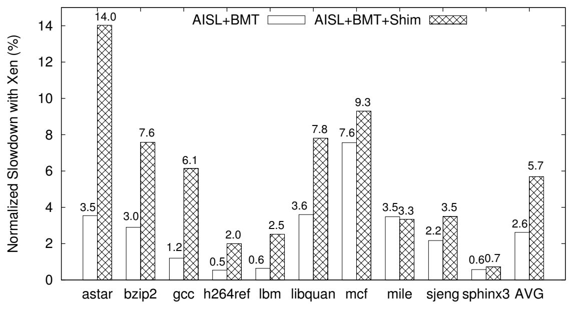
We model an in-order processor with split caches for data and counter. Specifically, the processor is modeled as single-core as the evaluated benchmarks are single-threaded. The last level data cache is 8MB in size, 8-way set-associative and has a counter cache with 64KB and 8-way set-associative. All caches uses the LRU replacement policy and each block is with 64 bytes. The main memory size is 512MB with an access latency of 350 cycles. The encryption engine uses AES with a latency of 80 cycles. HMAC based on SHA-1 is used for MAC computation, with an 80-cycle latency. The latency of each non-memory instruction is counted as one cycle. The encryption seed contains a 64-bit per-page LPID and a 7-bit per-block counter. A total of 64 counters and 1 LPID are co-located within one chunk, which corresponds to a 4KB memory page. The default hash size is 128 bits. The simulated machine runs Xen-4.0.1 as the hypervisor, Debian-6 and Windows XP-SP2 as the OSes for the VMs.

Figure 10a shows the performance overhead caused by Kite. The average overhead is about 5.7%, in which 2.6% is due to AISE and BMT, and 3.1% is due to the VM-Shim.

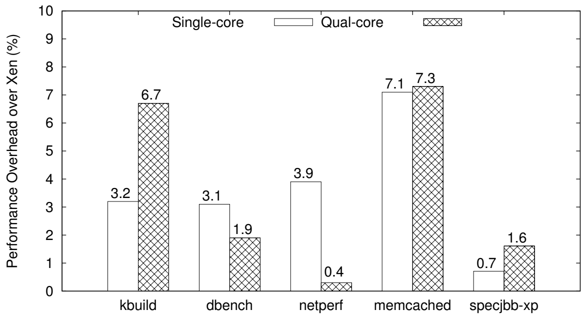
It should be noted that in our evaluation, we assume that the memory bandwidth is not the bottleneck even under the

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2015.2469657, IEEE Transactions on Cloud Computing



(a) The overall performance overhead and the encryption overhead of Kite on SPECINT-2006.



(b) The overhead of VM-Shim on real machine

Fig. 10: Performance Overhead on both Simulated and Real Machine

pressure of MAC access. However, this assumption may not be practical enough as the number of CPU cores increases, which could cause longer memory access latency due to contention. The evaluation and optimization for multi-core platform will be needed in the future.

9.2 Performance Evaluation on Real-machine

To evaluate the performance overhead of VM-Shim, we run several benchmarks on a real machine without a secure processor substrate. The real machine has an AMD quad-core CPU with 4GB memory and a 100Mb NIC and 320GB disk, on which we compare the performance of Linux and Windows VMs runs upon VM-Shim against vanilla Xen-4.0.1. Each VM is configured with one or more virtual CPUs, 1GB memory, a 20GB virtual disk and a virtual NIC. The VMs run unmodified Debian-Linux with kernel version 2.6.31 and Windows XP with SP2, both are with x86-64 versions.

We use a set of application benchmarks for Linux VMs, including Linux kernel build (kbuild), dbench-3.0.4 for disk I/O, netperf for network I/O and memcached-1.4.5 for memory. We also used SPECjbb-2005 to evaluate the server side performance of Java runtime environment in the Windows VM. We further evaluate the performance and scalability of VM-Shim by running all the benchmarks and applications with multiple cores.

Figure 10b shows the performance of VM-Shim on a single-core and a quad-core machine. The performance overhead for Kbuild is rather low, because there are very few VM traps. The overhead of disk I/O (shown in dbench) is also small because the disk and memory use the same encryption mechanism and same key to encrypt/decrypt, thus the hypervisor can do the copy between the memory and disk directly. The network performance overhead is relatively high, since VM-Shim needs to interpose and reveal data during package sending and receiving. On the quad-core machine, the netperf is bounded by network bandwidth, thus the performance degradation is less than the one on single-core machine. The performance overhead for SPECjbb is quite low, because it rarely interacts with the hypervisor and VM-Shim.

9.3 Storage Overhead

For each 4KB memory page, there is a 64-bit LPID and 64 7-bit counters needs to be saved in memory. For example, on a machine with 4GB main memory, 64MB memory is needed for the counters. The MAC of the 64MB counters occupies 128 bits per cache line (64B), which takes another 16MB memory (0.39% overhead) for the counters. The whole BMT is then about 21MB. Meanwhile, the MAC of the data is also 128 bits per 64 bytes, which takes 1GB memory. Thus, the total memory overhead is 21.55%. This memory overhead will be smaller if using 64-bit or 32-bit MAC. The disk also needs 1.56% storage for seeds. The overhead of hash is even smaller, since the hash tree costs 128 bits per 512KB. The total storage overhead of disk is 1.61%.

10 Conclusion

This paper considered a strong adversary model for multi-tenant cloud and proposed a hardware-software framework called Kite that enables the user to fully control the data, code and execution of the outsourced virtual appliance in the presence of untrusted hypervisor and even physical attacks. Kite carefully considered design and implementation issues with commercial off-the-shelf virtualization stack and extended existing processor virtualization with memory encryption and integrity checking as well as the VM-Shim mechanism to transparently secure control transitions and data interaction. It also regarded the high-level semantic of interaction between the VM and hypervisor to defend against attacks like rollback attack and Iago-like attacks. Performance evaluation shows that the performance overhead is small.

Acknowledgments

This work is supported in part by National Natural Science Foundation of China (No. 61303011), Doctoral Fund of Ministry of Education of China (No. 20130073120040), the Program for New Century Excellent Talents in University of Ministry of Education of China (No.ZXZY037003), Shanghai Science and Technology Development Funds

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2015.2469657, IEEE Transactions on Cloud Computing

(No. 12QA1401700), Shanghai Committee of Science and Technology Funds (No. 14511100900), a foundation for the Author of National Excellent Doctoral Dissertation of PR China (No.TS0220103006), Zhangjiang Hi-Tech program (No. 201501-YP-B108-012), and Singapore NRF (CREATE E2S2).

REFERENCES

[1] Y. Xia, Y. Liu, and H. Chen, "Architecture support for guest-transparent vm protection from untrusted hypervisor and physical attacks." in *HPCA*, 2013, pp. 246–257.

[2] C. P. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, M. S. Lam, and M. Rosenblum, "Virtual appliances for deploying and maintaining software." in *LISA*, vol. 3, 2003, pp. 181–194.

[3] Amazon Inc., "Amazon web service customer agreement," http://aws.amazon.com/agreement/, 2011.

[4] Microsoft Inc., "Microsoft online services privacy statement," http://www.microsoft.com/online/legal/?langid=en-us&docid=7, 2011.

[5] CircleID Reporter, "Survey: Cloud computing 'no hype', but fear of security and control slowing adoption." http://www.circleid.com/posts/20090226_cloud_computing_hype_security, 2009.

[6] TechSpot News, "Google fired employees for breaching user privacy," http://www.techspot.com/news/40280-google-fired-employees-for-breaching-user-privacy.html, 2010.

[7] "Common vulnerabilities and exposures," http://cve.mitre.org/.

[8] B. Schroeder, E. Pinheiro, and W. D. Weber, "Dram errors in the wild: A large-scale field study," in *SIGMETRICS*, 2009.

[9] "Failure rates in google data centers," http://www.datacenterknowledge.com/archives/2008/05/30/failure-rates-in-google-data-centers/, 2008.

[10] J. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum, and E. Felten, "Lest we remember: cold-boot attacks on encryption keys," *Communications of the ACM*, vol. 52, no. 5, pp. 91–98, 2009.

[11] Simply Security, "Physical attacks threaten internet security too," http://www.simplysecurity.com/2011/08/23/physical-attacks-threaten-internet-security-too/, 2011.

[12] J. Valamehr, T. Sherwood, A. Putnam, D. Shumow, M. Chase, S. Kamara, and V. Vaikuntanathan, "Inspection resistant memory: Architectural support for security from physical examination," in *Proc. ISCA*, 2012.

[13] S. Lai, "Current status of the phase change memory and its future," in *Electron Devices Meeting, 2003. IEDM'03 Technical Digest. IEEE International*. IEEE, 2003, pp. 10–1.

[14] F. Zhang, J. Chen, H. Chen, and B. Zang, "CloudVisor : Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization," in *Proc. SOSP*, 2011, pp. 203–216.

[15] S. Jin and J. Huh, "Secure MMU: Architectural Support for Memory Isolation among Virtual Machines," in *HotDep'11*, 2011.

[16] S. Jin, J. Ahn, S. Cha, and J. Huh, "Architectural Support for Secure Virtualization under a Vulnerable Hypervisor," in *MICRO*, 2011.

[17] J. Szefer and R. Lee, "Architectural support for hypervisor-secure virtualization," in *Proceedings of ASPLOS*, 2012.

[18] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," in *Proc. ASPLOS*, 2000, pp. 168–177.

[19] D. Lie, C. Thekkath, and M. Horowitz, "Implementing an untrusted operating system on trusted hardware," in *Proc. SOSP*, 2003.

[20] G. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas, "AEGIS: architecture for tamper-evident and tamper-resistant processing," in *Proc. Supercomputing*, 2003.

[21] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic, "SecureME: A Hardware-Software Approach to Full System Security," in *ICS*, 2011.

[22] D. Champagne and R. Lee, "Scalable architectural support for trusted software," in *Proc. HPCA*, 2010, pp. 1–12.

[23] T. Ristenpart and S. Yilek, "When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography." in *NDSS*, 2010.

[24] Y. Xia, Y. Liu, H. Chen, and B. Zang, "Defending against vm rollback attack," in *International Workshop on Dependability of Clouds Data Centers and Virtual Machine Technology*, 2012.

[25] S. Checkoway and H. Shacham, "Iago attacks: why the system call api is a bad untrusted rpc interface," in *ASPLOS*, 2013.

[26] Wikipedia, "Shim," http://en.wikipedia.org/wiki/Shim_%28computing.

[27] P. Golle and I. Mironov, "Uncheatable distributed computations," in *Topics in CryptologyCT-RSA 2001*. Springer, 2001, pp. 425–440.

[28] D. Szajda, B. Lawson, and J. Owen, "Hardening functions for large scale distributed computations," in *Proceedings of IEEE Symposium on Security and Privacy*. IEEE, 2003, pp. 216–224.

[29] M. Blanton, Y. Zhang, and K. B. Frikken, "Secure and verifiable outsourcing of large-scale biometric computations," *Transactions on Information and System Security (TISSEC)*, vol. 16, no. 3, Nov. 2013.

[30] M. Green, S. Hohenberger, and B. Waters, "Outsourcing the decryption of ABE ciphertexts," in *Proceedings of the 20th USENIX conference on Security (SEC'11)*. USENIX Association, Aug. 2011.

[31] M. J. Atallah, K. N. Pantazopoulos, J. R. Rice, and E. E. Spafford, "Secure outsourcing of scientific computations," *Advances in Computers*, vol. 54, pp. 215–272, 2002.

[32] M. J. Atallah and J. Li, "Secure outsourcing of sequence comparisons," *International Journal of Information Security*, vol. 4, no. 4, pp. 277–287, Mar. 2005.

[33] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford University, 2009.

[34] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Provable data possession at untrusted stores," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 598–609.

[35] A. Juels and B. S. Kaliski Jr, "Pors: Proofs of retrievability for large files," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 584–597.

[36] R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina, "Controlling data in the cloud: outsourcing computation without outsourcing control," in *CCSW*, 2009, pp. 85–90.

[37] I. Intel, "Intel64 and ia-32 architectures software developers manual volume 3: System programming guide," 2014.

[38] E. Keller, J. Szefer, J. Rexford, and R. Lee, "NoHype: virtualized cloud infrastructure without the virtualization," in *Proc. ISCA*, 2010.

[39] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu, "Policy-sealed data: A new abstraction for building trusted cloud services," in *Usenix Security*, 2012.

[40] X. Chen, T. Garfinkel, E. Lewis, P. Subrahmanyam, C. Waldspurger, D. Boneh, J. Dwoskin, and D. Ports, "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems," in *Proc. ASPLOS*. ACM, 2008, pp. 2–13.

[41] G. Suh, D. Clarke, B. Gasend, M. van Dijk, and S. Devadas, "Efficient memory integrity verification and encryption for secure processors," in *Proc. MICRO*, 2003, pp. 339–350.

[42] R. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang, "Architecture for protecting critical secrets in microprocessors," in *Proc. ISCA*, 2005, pp. 2–13.

[43] B. Rogers, M. Prvulovic, and Y. Solihin, "Efficient data protection for distributed shared memory multiprocessors," in *PACT*, 2006.

[44] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving Cost, Performance, and Security of Memory Encryption and Authentication," in *Proc. ISCA*, 2006, pp. 179–190.

[45] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, "Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly," in *MICRO*, 2007.

[46] J. Dwoskin and R. Lee, "Hardware-rooted trust for secure key management and transient trust," in *Proc. CCS*. ACM, 2007.

[47] S. Chhabra, B. Rogers, and Y. Solihin, "SHIELDSTRAP: Making secure processors truly secure," in *Proc. ICCD*, 2009, pp. 289–296.

[48] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proc. CCS*. ACM, 2009, pp. 199–212.

[49] J. Shi, X. Song, H. Chen, and B. Zang, "Limiting cache-based side-channel in multi-tenant cloud using dynamic page," in *Proc. HotDep*, 2011.

[50] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: secure applications on an untrusted operating system," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 265–278, 2013.

[51] Y. Zhang, L. Gao, J. Yang, X. Zhang, and R. Gupta, "SENSS: Security Enhancement to Symmetric Shared Memory Multiprocessors," in *Proc. HPCA*, 2005, pp. 352–362.

[52] J. Edler and M. Hill, "Dinero iv trace-driven uniprocessor cache simulator," http://pages.cs.wisc.edu/ markhill/DineroIV/, 1998.

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2015.2469657, IEEE Transactions on Cloud Computing

Yubin Xia received the deploma degree in software school, Fudan University, Shanghai, China, in 2004, and the Ph.D. degree in computer science and technology from Peking University, Beijing, China, in 2010. He's now an assistent professor in Shanghai Jiao Tong University since September 2012. His research interests include computer architecture, operatig system, virtualization and security.

Yutao Liu received the deploma degree in software school, Fudan University, Shanghai, China, in 2012, and he's now pursuing his the Ph.D. degree in Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai, China. His research interests include the area of computer architecture, system security, virtualization.

Haibing Guan obtained his Ph.D. degree from Tongji University in 1999 and worked in Shanghai Jiao Tong University since 2002. He's now the Vice Dean of School of Electronic, Information and Electronic Engineering, Shanghai Jiao Tong University. His major research interests include computer system, Cloud Computing.

Yunji Chen received the deploma degree in University of Science and Technology of China in 2002, and the Ph.D. degree in Institute of Computing Technology, Chinese Academy of Sciences, in 2007. He's now a professor leading the NOVEL group to build non-traditional computer, especially neural network computer.

Tianshi Chen received the deploma and Ph.D. degree from University of Science and Technology of China (USTC) in 2005 and 2010, respectively. He's currently an associate professor at State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, China. His research interests lie in computer architecture and computational intelligence.

Binyu Zang received the deploma degree in Computer Science in Fudan University, in 1999. He's now the Dean of School of Software of Shanghai Jiao Tong University, Shanghai, China. His research interests include compiler, parallel processing, computer architecture, operating systems and virtualiztation.

Haibo Chen received the deploma degree and Ph.D. degree in Fudan University, Shanghai, China, in 2004 and 2009, respectively. He's now a professor leading the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai, China. His research interests include the area of virtualization, computer architecture, system security, distributed computing, etc.

2168-7161 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.