Architecture Support for Guest-Transparent VM Protection from Untrusted Hypervisor and Physical Attacks*

Yubin Xia, Yutao Liu, Haibo Chen

Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
{xiayubin, ytliu.cc, haibochen}@sjtu.edu.cn

Abstract

The privacy and integrity of tenant's data highly rely on the infrastructure of multi-tenant cloud being secure. 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 data leakages or abuses in cloud. Unfortunately, most prior solutions require intrusive changes to the cloud platform and none can protect a VM against adversaries controlling the physical machine.

This paper analyzes the challenges of transparent VM protection against sophisticated adversaries controlling the whole software and hardware stack. Based on the analysis, this paper proposes HyperCoffer, a hardware-software framework that guards the privacy and integrity of tenant's VMs. HyperCoffer only trusts the processor chip and makes no security assumption on external memory and devices. HyperCoffer extends existing processor virtualization with memory encryption and integrity checking to secure data communication with off-chip memory. Unlike prior hardware-based approaches, HyperCoffer retains transparency with existing virtual machines (i.e., operating systems) and requires very few changes to the (untrusted) hypervisor. HyperCoffer introduces a mechanism called VM-Shim that runs in-between a guest VM and the hypervisor. Each VM-Shim instance for a VM runs in a separate protected context and only declassifies necessary information designated by the VM to the hypervisor and external environments (e.g., through NICs). We have implemented a prototype of HyperCoffer in a QEMU-based full-system emulator and the VM-Shim mechanism in a real machine. Performance measurement using trace-based simulation and on a real hardware platform shows that the performance overhead is small (ranging from 0.6% to 13.9% on simulated platform and 0.3% to 6.8% on real hardware for the VM-Shim mechanism).

1. Introduction

A key premise underlying multi-tenant cloud is that users' data should be securely stored and processed. Unfortunately, many commercial clouds only provide limited security assurance on users' data [1, 2]. 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" [3].

There are two main reasons for such limited security assurance First, the hardware and software stack in multi-tenant cloud is notoriously large and complex, which raises the possibility of security compromises on the virtualized stack [4]. 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 daily work. In practice, Google reports thousands of machine failures every year [5]. Detecting subtle malicious actions from frequent maintenance operations in large cloud platforms is like finding a needle in a haystack. Those curious or even malicious cloud operators who can easily gain the full control of the cloud may unrestrictedly inspect or tamper with users' data, by either physical [6, 7, 8] or software attacks [4]. Furthermore, recovering data from residues of off-power memory [6, 8] has shown to be possible. This situation will be even worse if the replaced memory has become non-volatile (e.g., phase-change memory).

For these reasons, one report from Gartner states that one of the greatest challenges of cloud computing is "invisibly access unencrypted data in its facility" [9]. One recent survey on cloud security also states that physical attacks have serious implication on data security [7]. Worse even, the threats are not groundless but real. Google has recently fired employees for "breaking internal privacy policies" and causing "a massive breach of privacy" in Gtalk and Google Voice [10]. Surprisingly, the privacy breach lasted for several months before being detected.

With the whole hardware and software stack of a multi-tenant cloud being controlled by cloud operators, users will likely be forced to assume a strong adversary model that trusts only a small part of the cloud. Previous research shows that it is reasonable to assume a tamper-resistant CPU chip to be secure, while leaving external memory and devices as un-

*This work is supported by grants from Shanghai Science and Technology Development Funds (No. 12QA1401700), Foundation for the Author of National Excellent Doctoral Dissertation of PR China, and China National Natural Science Foundation (No. 61003002).

trusted [11, 12, 13, 14]. However, most prior secure processor proposals focus on application-level protection and require a non-trivial change of operating systems [11, 12, 13, 15], applications [14], or both. In contrast, software-based approaches such as CloudVisor [16] cannot guard against physical attacks like bus snooping and cold-boot attack[6, 8]. Similar to CloudVisor, hardware proposals including SecureMMU [17], H-SVM [18] and HyperWall [19] leverage architectural support to enhance the memory management units to isolate a VM's memory from the hypervisor[1]. However, they require changes to operating systems and cannot defend against physical attacks.

In this paper, we propose a hardware-software framework, named HyperCoffer, which provides strong and transparent VM-level protection in a multi-tenant cloud. HyperCoffer only trusts the processor chip, while assuming all other hardware components such as external memory and devices as untrusted. Unlike previous approaches, HyperCoffer can protect against both software and physical attacks at VM level.

However, enforcing security policies inside secure processor is difficult due to the *semantic gap* between a VM and secure processor. This is because processors are not expressive enough to capture and handle complex high-level semantics inside a VM, including the data interaction between VM and hypervisor as well as external environments. Further, relying purely on hardware-based protection limits the scalability in supporting a virtually infinite number of VMs on a CPU core, due to the restricted functionality and limited on-chip storage. To this end, HyperCoffer 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]. HyperCoffer 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. One way of VM-Shim deployment is to make the code publicly open-sourced for all clients to verify its harmlessness.

To demonstrate the applicability of HyperCoffer, we have implemented a prototype in a QEMU-based full-system emulation environment with the Xen VMM. The VM-Shim consists of around 1,100 LOCs and requires modification around 380 LOCs to the hypervisor and VM management tools, which shows that the VM-Shim can be easily implemented with modest code size. The performance measurement also shows that the overhead is small.

[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 [20], we use the name VM-Shim as it adapts a VM to HyperCoffer without the requirement to change the guest OS.

In summary, this paper makes the following contributions:

- The first hardware-software framework that *transparently* protects guest virtual machines against an untrusted hypervisor and even physical attacks.
- The VM-Shim mechanism that provides a scalable and transparent approach to protecting an arbitrary number of VMs on commercial off-the-shelf virtualization stack.
- 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.

The rest of this paper is organized as follows. The next section describes necessary background and related work. Section 3 describes the goals and challenges underlying HyperCoffer, as well as an overview of the design of HyperCoffer. Section 4 illustrates the architecture extension required by HyperCoffer. Section 5 describes the specification of interactive data for VM-Shim, followed by an overview of how HyperCoffer leverages such extension to secure the life-cycle of a VM in section 6. We then present security analysis in section 7 and performance results of HyperCoffer in section 8. Finally, section 9 concludes this paper.

2. Background and Related Work

2.1. Virtualization & VM Protection

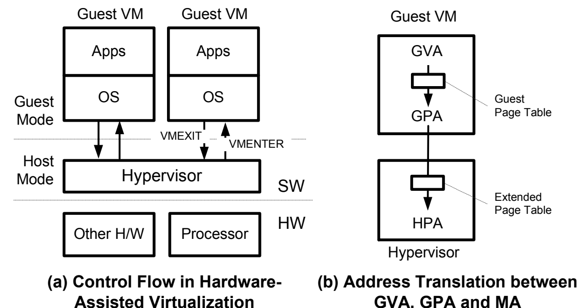


Figure 1: Hardware-Assisted Virtualization

Hardware-assisted virtualization 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 VM's 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 han-

dle 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 [21, 22] 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, HyperCoffer 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, HyperCoffer still retains most functionalities in a commercial hypervisor like time-multiplexing resources, which are currently absent in NoHype. To further guard the privacy and integrity of a VM's memory and disk images, CloudVisor [16] separates the security protection from resource management and leverages a tiny nested hypervisor to encrypt and check the integrity of VMs. However, it does not defend against physical attacks and still requires means to secure the nested hypervisor, which may suffer from single point of security failure.

On the hardware side, H-SVM [18] and HyperWall [19] 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. For example, H-SVM needs to handle the complex VM interactions and data sharing by patching both guest OSes and the hypervisor. Further, H-SVM does not protect data in external devices but instead insists the VM itself to secure its I/O data. HyperWall also requires the guest OS to decide which memory pages to be protected against the untrusted hypervisor, which is a non-trivial task for even a sophisticated programmer as specifying such pages requires deep understanding of different OSes and usages of memory pages by applications are dynamic. Further, HyperWall ignores the complex data interaction between the hypervisor and guest OS illustrated in this paper (section 3.1).

[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.

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, HyperCoffer also assumes an untrusted hypervisor. However, HyperCoffer does not trust the external memory or devices and uses memory encryption and integrity verification to protect off-chip data. Further, HyperCoffer captures complex VM interactions and data exchanging in a VM-Shim to retain OS transparency. Finally, HyperCoffer is designed to still support existing cloud maintenance operations (section 6).

	OS Transparent	TCB Size	Phys. Attk.	Cloud Func.
HyperWall	No	CPU + Memory + IOMMU	No	Part
H-SVM	No	CPU + Memory + IOMMU	No	Full
CloudVisor	Yes	All hw + CloudVisor (5.5K LOC)	No	Full
HyperCoffer	**Yes**	**CPU + VM-Shim (1.1K LOC)**	**Yes**	**Full**

Table 1: Comparison of Related Systems

Similar to HyperCoffer, Overshadow [23] also leverages the concept of shim [20], 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 HyperCoffer. 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 HyperCoffer is portable among different OSes and is much simpler.

2.2. Secure Processor

Secure processor has been extensively studied during the last decade [11, 12, 24, 13, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 14, 15]. For example, Bastion [14] and SecureME [15] also leverage secure processor to protect against hardware attacks. However, they both need to trust the hypervisor, while HyperCoffer only trust the processor chip. Further, unlike HyperCoffer, they break application/OS transparency in requiring either non-trivial OS modification [15] or refactoring applications into modules for protection. In contrast, HyperCoffer retains OS and application transparency by leveraging the VM-Shim mechanism.

For secure processor designs, we briefly review two major techniques: AISE-based data encryption and Bonsai Merkle Tree (BMT) [33], as they will be adapted to HyperCoffer to secure off-chip data.

AISE-based Data Encryption for Memory Privacy Protection: Rogers et al. proposed *counter-mode address-independent seed encryption* (AISE) [33] for memory encryption. Figure 2 shows the data flow of an AISE secure processor. Instead of encrypting/decrypting a data block (i.e., cache block) directly, the processor encrypts/decrypts a *seed* of the block to generate a pseudo-random pad, which is then XORed

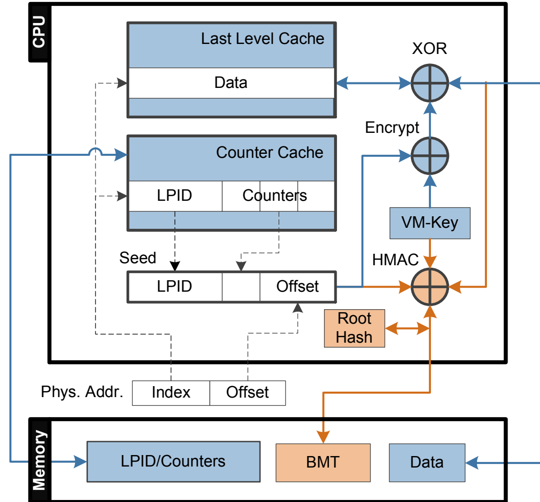


Figure 2: Secure Processor with AISE and BMT

with the plain-text of the data block to generate cyphertext. Similarly, plain-text is generated by XORing cyphertext with the same pad. A *seed* is composed of three parts: LPID, counter, and offset. LPID (Logical per-Page ID) is unique for every physical memory page. Its value is assigned at the initialization time and independent of the page address. The counter and page offset are for every cache block. Once a counter overflows, the corresponding page is assigned with a new LPID and re-encrypted. Therefore, an attacker cannot issue replay attacks by reusing a seed. All the LPID and counters are saved in the main memory and can be found using simple indexing for a given physical address. A separated cache, named *counter cache*, is introduced to host LPID and counters for optimization.

BMT for Memory Integrity Protection: *Bonsai Merkle Tree* (BMT) [33] has been proposed to do integrity checking. A hash value is generated from the cyphertext and the hash tree is updated accordingly. When loading data from memory to on-chip cache, the corresponding hash value is fetched and compared with the hash value calculated from the loaded data block. The root of the BMT is saved inside the chip. In BMT, only the memory region of counter and LPID is covered by the hash tree, the data region is protected only by hash without tree. The basic idea is to use counter as the version of data, thus the integrity of data is ensured as long as the counters are protected.

It is worth noting that the overhead of AISE+BMT is very low: merely 1.8% for SPEC2K [33]. The reasons are as following. First, encryption is needed only in cases of data cache miss. Second, the encryption/decryption is on the seed instead of the data thus can be parallelized with memory load-/store. Third, the hit rate of counter cache is very high since the counter size is small. Fourth, BMT significantly reduces the size of hash in both memory and cache.

3. Goals, Challenges and Design

The overall goal of HyperCoffer is to provide strong privacy and integrity protection of VMs against even physical attacks, while minimizing the size of the trusted computing base (TCB). Further, as one reason for the success of virtualization is backward compatibility with existing OSes, it is demanding that HyperCoffer should not sacrifice backward compatibility in requiring changes to guest OSes. Finally, as VM management operations like snapshot and migration are indispensable for maintaining cloud platforms, HyperCoffer should also support such operations.

3.1. Challenges

The hypervisor cannot work correctly if all data interactions with VM are forbidden. However, dividing protected and unprotected data is a non-trivial work due to the complex interaction between hypervisor and VMs.

Secure VM/Hypervisor Interaction: The interaction between VM and the hypervisor is complex. Let's take privilege instruction emulation as an example: Once a VM executes "*out %dx %eax*", it will trap to the hypervisor to emulate this instruction, since *out* is a privilege I/O instruction. Once the hypervisor start to run, it first gets the address of the trapped instruction, which is in GVA. It then translates GVA to GPA by walking through the page table of a VM, and further translates GPA to HPA. According to the HPA, it maps the memory page and fetches the instruction. In order to emulate the logic, the hypervisor also needs to get the value of register %eax and %dx. The process involves accesses to both memory and CPU context of the VM.

Interaction with the Outside World: When operating an I/O device, a VM needs to send metadata (e.g., DMA commands) to the device in plain-text, otherwise the device cannot perform correctly. Many prior approaches secured data exchanges with external devices by making some assumptions on the devices or requiring rewriting of device-specific parts in OS. For example, Shi et al. [38] required north bridge to keep track of the secure memory ranges and assumed an intelligent memory controller to automatically verify and convert the encryption scheme for I/O data. AEGIS [13] partitioned external memory into secure and insecure parts. XOM [11, 12] required rewriting of OS, which led to non-trivial engineering work and was not portable among different OSes.

3.2. HyperCoffer Design Overview

Figure 3 shows an overview of HyperCoffer. HyperCoffer leverages and extends traditional secure processor technology for virtualization environment. It uses AISE for encryption, BMT for integrity check, and introduces VM-Table for multiplexing. All the data within a VM is protected, including the data in CPU context, on-chip cache, memory and I/O device. Different VMs are isolated with each other since they

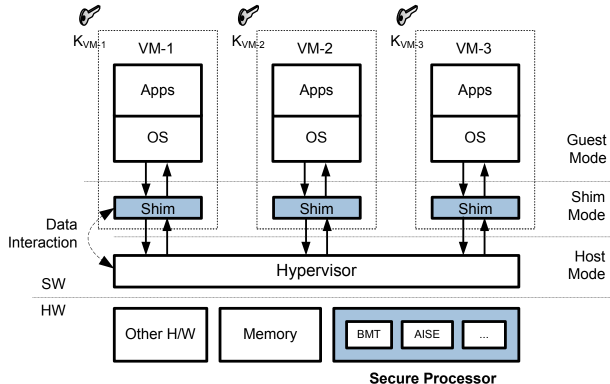


Figure 3: Overview of HyperCoffer

use different keys.

Applying secure processor alone is far from enough, since it cannot bridge the semantic gap between VMs and the hypervisor. A simple solution would require non-trivial modifications to both the hypervisor and guest OSes, which are resource-intensive, non-portable and error-prone. To retain OS transparency and only reveal necessary information of VM to the hypervisor, HyperCoffer provides a mechanism called VM-Shim, which consists of two components: 1) hardware support to enable control interposition between the hypervisor and VMs, as stated in section 4.6 and 2) a specification of interactive data between the hypervisor and VMs, which contains CPU context, I/O data and auxiliary information, as stated in section 5. 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. Unlike OS-specific drivers in para-virtualization system, VM-Shim is OS-independent and can run without any awareness of the guest OS. Thus, the VM-Shim mechanism enables HyperCoffer to retain OS-transparency in requiring no changes to guest OS, which is crucial to current multi-tenant cloud. It is also highly scalable in supporting an arbitrary number of VMs. 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 VM-Shim mechanism serves as a key means to minimize necessary hardware changes and bridge the semantics gap.

Unlike previous approaches relying on manipulating address translation [16, 18], or introducing additional access flag [19] to secure VM's memory, HyperCoffer is designed to be orthogonal to the existing memory virtualization schemes. HyperCoffer also makes no assumption on whether a system has IOMMU[4] or not.

The TCB of HyperCoffer contains only the secure processor, which can be easily verified by its public key. The clients 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. In addition, it introduces new instructions to securely support VM operations, including VM suspend/resume and migration, as described in section 6.

[4]IOMMU translates physical addresses to device addresses.

3.3. Threat Model and Security Guarantees

In HyperCoffer, neither the virtualization software stack nor physical environment is trusted, resulting in a strong adversary model and a minimized TCB. 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, given that primary goal of cloud providers is featuring utility-style computing resources to users with certain service-level agreement, HyperCoffer provides no guarantee on the execution correctness and availability of a VM. Third, we do not try to prevent against side-channel attacks in the cloud [39], which are usually hard to deploy and have very limited bandwidth to leak information. However, HyperCoffer ensures that an adversary controlling a subverted VM cannot further break other VMs through the tampered hypervisor or even abused hardware.

4. HyperCoffer Architecture

4.1. Memory Data Protection

HyperCoffer adopts AISE and BMT to protect memory due to their low overhead (as mentioned in section 2.2). One difference from traditional AISE and BMT is that in HyperCoffer 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 4. 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.

4.2. Cache Data Protection

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 HyperCoffer, 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 4.5.

4.3. CPU Context Protection

CPU context is also properly protected when execution transfers from a VM to the hypervisor or other VMs. The

New Instruction	Environment	Instruction Semantic
vm_install, addr1, addr2	Hypervisor	Install vm_key (addr1) and vm_vector (addr2). Return VMID.
vm_uninstall, VMID	Hypervisor	Remove the vm_vector indexed by VMID from the VM-Table.
vm_snapshot, VMID, addr	Hypervisor	Encrypt the vm_vector indexed by VMID and save it to memory.
ept_st, addr, val	Hypervisor	Update data in EPT memory. Invalid cache only if an GPA_2_HPA mapping is modified or deleted.
VMENTER (modified)	Hypervisor	Resume VM-Shim instead of the VM
VMEXIT (modified)	Guest VM	Transfer control to VM-Shim instead of the hypervisor
shim_to_host	Shim	Trigger VMEXIT and switch to host mode
shim_to_guest	Shim	Switch to guest mode and resume VM
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

Table 2: New Instructions in HyperCoffer

Key	Context	Protection
K_{vm}	per VM	Encrypt VM memory and disk image
K_{mem}	per Chip	Encrypt CPU reserved memory for VM-Table
SK_{cpu}	per Chip	Private key of the CPU

Table 3: Keys involved in HyperCoffer

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 5.1.

4.4. EPT Protection

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.

HyperCoffer 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 to enable the new mapping. To avoid unnecessary cache invalidation, e.g., during VM launching and destroying, HyperCoffer 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.

4.5. 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:
  - $HRoot_{vm}$ is the root hash of the VM's BMT.
  - $Addr_{cnt}$ and $Addr_{BMT}$ are start addresses (GPA) of counters and BMT in memory.
  - $Addr_{shim}$ is the entry address (GPA) of VM-Shim.

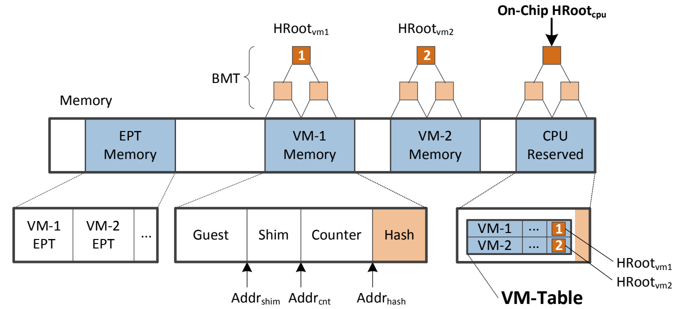


Figure 4: The root hash of guest BMT is saved in VM-Table. VM-Table is in CPU-reserved memory region, whose root hash is saved on-chip.

In our design, the VM-Table is saved 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 saved 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 4. By using memory to save 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., *vm_install*, *vm_uninstall* and *vm_snapshot*, as listed in table 2.

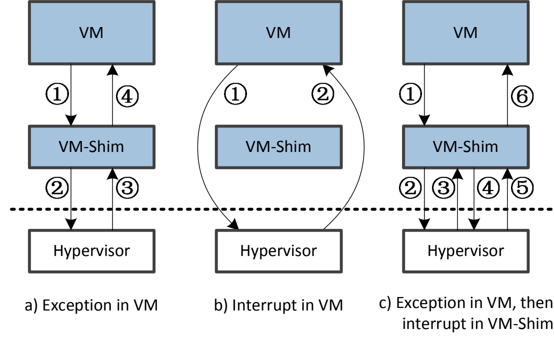


Figure 5: VM-Shim Interposition

4.6. VM-Shim Mode

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, HyperCoffer 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.

HyperCoffer 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.

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 4.3, the *virtual interrupt vector* is not protected, thus the hypervisor can modify it directly.

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.

4.7. Summary of HyperCoffer Architecture

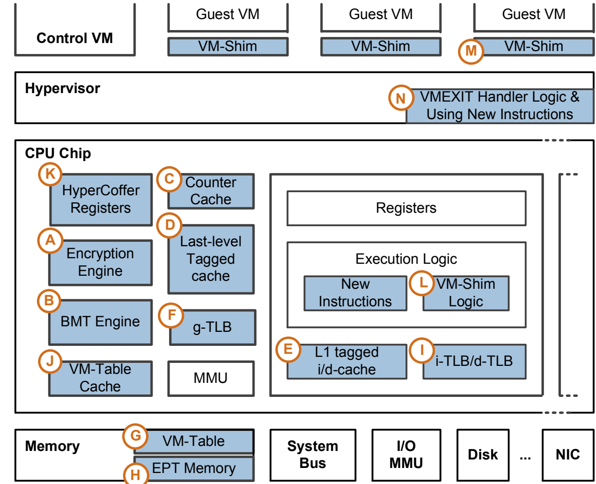


Figure 6: Hardware/software Modifications for HyperCoffer

Figure 6 shows the hardware and software components of HyperCoffer. 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 the 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 $\{VMID, K_{vm}, HRoot_{vm}, Addr_{cnt}, Addr_{BMT}, Addr_{shim}\}$. The VM-Table is saved in CPU protected memory region Ⓖ, and most recent used entries are cached on chip Ⓙ. Three new instructions,

vm_install, *vm_snapshot* and *vm_uninstall* are introduced to operate the VM-Table. EPT memory Ⓗ is used to store EPT, and can only be modified by *ept_st*. It triggers invalidation of cache when address mapping is changed to defend against intra-VM remapping attack, as stated in section 4.4. The invalidation is delayed to TLB update Ⓘ for optimization.

Registers Ⓚ of HyperCoffer include two non-volatile registers. One is used for generating LPID to ensure that LPID is unique for each page, even after system rebooting. The other is used for logging that gets updated every time a VM is booted/resumed or a snapshot is made, triggered by *vm_install* and *vm_snapshot*, respectively. 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: Interactive Data Specification

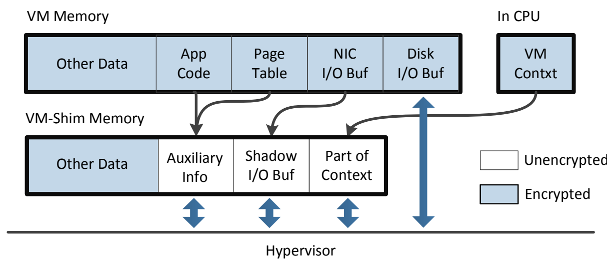


Figure 7: Data Interaction

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.

5.1. CPU Context

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

5.3. Network I/O

Network I/O is handled differently from disk I/O as HyperCoffer 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, HyperCoffer, like other similar systems [16, 23], 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 HyperCoffer, 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.

5.4. 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. VM Life-cycle Protection

6.1. Secure Processor Initialization

When a system boots up, the processor initializes a region of the main memory to store VM-Table. It randomly generates a key as K_{mem} and initializes BMT over the region, with its root hash saved on-chip. Once the reserved memory region is initialized, the system continues booting as usual.

6.2. VM Bootup & Shutdown

Before booting a VM, the owner of the VM needs to offer following components, which are generated offline by users using our provided tool:

- A disk image of the VM. The metadata of the disk includes the start address of the counter zone and hash zone, as well as the root hash of the BMT.
- 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}. An wrong formatted image will be denied by a secure processor.
- K_{vm}, which is encrypted by the SK_{cpu} of host chip.
- A *vm_vector*, which is encrypted by K_{vm}. The vector summarizes the initial VM memory image and is used by the the processor to verify the image.

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 *vm_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. All the essential information for booting the VM are now ready.
5. During initialization, the VM will get the BMT root hash of the disk image. After that, each disk read can be checked to ensure disk data's integrity.

A user can verify the identity of a running VM by putting some secrets in the VM, and challenging it during runtime.

The termination process of a VM is much simpler. When a VM is shutting down, it does not have to zero all the memory pages, since they are encrypted already. After a VM is shutdown, the hypervisor only needs to execute *vm_uninstall* to revoke the slot of VM-Table.

6.3. VM Snapshot & Restore

When a hypervisor takes a snapshot of a VM, it first saves the VM's current CPU context, memory data and disk data, all in cipher-text. It then issues *vm_snapshot* 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 *vm_install*, similar as booting a VM.

There is one difference between VM booting and VM restoring. A VM can boot up from any disk image, as long as the disk image is encrypted using K_{vm} and has not been tampered with. On the other hand, a VM can only boot up from the disk image that is used during the snapshotting. Since when a VM is running, the disk's metadata (which includes the root hash of the disk's BMT) is already kept in VM-Shim's memory. When the VM resumes, it still uses that $HRoot_{disk}$, which has only one corresponding disk image.

6.4. VM Migration

VM migration is similar to 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 *vm_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}. Thus the VM migration is done by the hypervisor without any involvement of VM-Shim.

6.5. Memory Sharing and VM Introspection

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.

7. Security Analysis

7.1. Data Privacy and Integrity

In HyperCoffer, 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.

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.

Another possible attack is inter-VM remapping attack, which can be made by a hypervisor with an accomplice VM. As a VM's data is decrypted in cache, a malicious hypervisor may map a VM's memory page to a conspiratorial 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.

Similarly, intra-VM remapping attack is done by remapping different pages of the same VM. If the pages are in cache, then the BMT check is bypassed and the integrity of VM's memory is violated. HyperCoffer defeats this attack by invalidating cache at the time of remapping (EPT modification), as stated in Section 4.4.

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.

7.2. Security of VM-Shim

A 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 very 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.

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 user's intention.

7.3. Other Security Issues

As discussed in section 3.3, HyperCoffer 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 wrong result when processing VMEXIT. However, HyperCoffer ensures that these attacks on availability cannot get user's private data or tamper with the execution of a VM.

The attack surface composed of interactive data between a VM and the hypervisor is quite small. Most VMEXITs just need a few pieces of data, e.g., contents of CR0 or %eax. Thus, the VM data exported through VMEXITs is very limited, which is far from enough to mount a security attack.

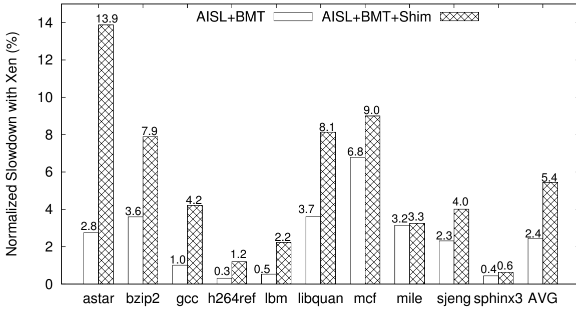
A malicious hypervisor may manipulate VM rollback attack by taking a snapshot of a VM and repeatedly restoring it. HyperCoffer uses logging and auditing to defend against VM rollback attack [40]. The log cannot be tampered with since each time a secure processor executes *vm_install* or *vm_resume*, the hash of the *vm_vector* will be accumulatively chained in a non-volatile register, which can be audited by the user. Meanwhile, the memory snapshot image is encrypted and protected by BMT. The root hash of BMT is stored in *vm_vector*, which is also encrypted by K_{vm}. Thus, as long as the K_{vm} is safe, the snapshot is protected.

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.

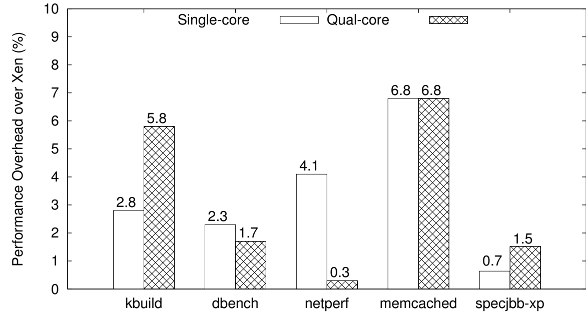
Currently we support multi-core chips, but not multi-chip processors. The challenge is that the data encryption mechanism used in HyperCoffer does not suit data exchanging among chips. SENSS [28] utilizes the Cipher Block Chaining mode of the advanced encryption standard (CBC-AES) for encryption/decryption of shared bus between chips. Supporting multi-chip processors and multi-processor will be our future work.

8. Evaluation

We implemented a working prototype by modifying QEMU full-system emulator to validate the applicability of HyperCoffer. 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



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



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

Figure 8: Performance Overhead on both Simulated and Real Machine

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 HyperCoffer. The VM-Shim currently consists of around 1,100 LOCs. We also change around 180 LOCs in Xen to secure VM booting and cooperate with the data exchange mechanisms in VM-Shim.

8.1. 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 [41] to do trace-based simulation. We use a set of benchmarks from SPECINT-2006. 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 [15].

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. 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 8a shows the performance overhead caused by HyperCoffer. The average overhead is about 5.4%, in which 2.4% is due to AISE and BMT, and 3.0% is due to the VM-Shim.

8.2. 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, netperf and memcached-1.4.5. 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 8b 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 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.

8.3. 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 (1.56% overhead). Meanwhile, the BMT of

the 64MB counters occupies 128 bits per cache line (64B), which takes another 16MB memory (0.39% overhead). The total memory overhead is 1.95%. 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%.

9. Conclusion

This paper considered a strong adversary model for multi-tenant cloud and proposed a hardware-software framework called HyperCoffer that transparently guards the privacy and integrity of tenants' VMs against untrusted hypervisor and even physical attacks. HyperCoffer 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 secure control transitions and data interaction. The resulted system retains transparency with guest VMs, is non-intrusive to commercial hypervisor, retains VM management operations and makes few assumptions on external memory and devices. Performance evaluation shows that the performance overhead is small.

References

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

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

[3] 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.

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

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

[6] 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.

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

[8] 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.

[9] J. Heiser and M. Nicolett, "Assessing the security risks of cloud computing," http://www.gartner.com/DisplayDocument?id=685308, 2008.

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

[11] 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.

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

[13] 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.

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

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

[16] 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.

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

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

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

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

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

[22] J. Szefer, E. Keller, R. B. Lee, and J. Rexford, "Eliminating the Hypervisor Attack Surface for a More Secure Cloud," in *Proc. CCS*, 2011.

[23] 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.

[24] D. Lie, J. Mitchell, M. Horowitz, and S. Ca, "Specifying and verifying hardware for tamper-resistant software," in *ISSP*, 2003.

[25] B. Gassend, G. Suh, D. Clarke, M. van Dijk, and S. Devadas, "Caches and hash trees for efficient memory integrity verification," in *Proc. HPCA*, 2003, pp. 295–306.

[26] 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.

[27] W. Shi, H.-h. S. Lee, and C. Lu, "High Efficiency Counter Mode Security Architecture via Prediction and Precomputation College of Computing," in *Proc. ISCA*, 2005.

[28] 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.

[29] 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.

[30] B. Rogers, M. Prvulovic, and Y. Solihin, "Efficient data protection for distributed shared memory multiprocessors," in *Proc. PACT*, 2006, pp. 84–94.

[31] W. Shi and H.-H. S. Lee, "Authentication control point and its implications for secure processor design," in *Proc. Micro*, 2006, pp. 103–112.

[32] 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.

[33] 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 *Proc. MICRO*, 2007, pp. 183–196.

[34] J. Dwoskin and R. Lee, "Hardware-rooted trust for secure key management and transient trust," in *Proc. CCS*. ACM, 2007, pp. 389–400.

[35] B. Rogers, C. Yan, S. Chhabra, M. Prvulovic, and Y. Solihin, "Single-level integrity and confidentiality protection for distributed shared memory multiprocessors," in *Proc. HPCA*, 2008, pp. 161–172.

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

[37] R. Elbaz, D. Champagne, C. Gebotys, R. Lee, N. Potlapally, and L. Torres, "Hardware mechanisms for memory authentication: A survey of existing techniques and engines," *Transactions on Computational Science IV*, pp. 1–22, 2009.

[38] W. Shi, H.-H. S. Lee, M. Ghosh, and C. Lu, "Architectural support for high speed protection of memory integrity and confidentiality in multiprocessor systems," in *Proc. PACT*, 2004, pp. 123–134.

[39] 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.

[40] 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.

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