Hardware-Enhanced Security for Cloud Computing

Jakub Szefer and Ruby B. Lee

Abstract Cloud computing has ushered in an era where cloud customers are able to rapidly access on-demand computing resources made available by third party cloud providers. The cloud providers who maintain these computing resources and lease them out to customers leverage economies of scale and sharing of resources to be able to provide these resources to customers at favorable prices. Cloud computing and this sharing of resources, however, introduces a number of security concerns. These concerns include other, potentially malicious, customers who are co-located on the same system as the customer; or even untrusted system software running on the remote systems where a customer's code and data execute or reside. To tackle these security concerns, we explore how secure hardware architectures can provide more protections to a customer's code and data in a cloud computing setting. In particular, we want to show that with hardware enhancements we can make computing in the cloud as secure as in your own dedicated facilities.

1 Introduction

Figure 1 shows the IaaS cloud computing model. Other cloud computing models, such as Platform-as-a-Service (PaaS) or Software-as-a-Service (SaaS), can be built on top of the IaaS model and could leverage the hardware security enhancements we present for the IaaS scenario. Before switching to using cloud computing, users may have run applications and an operating system (OS) on their own hardware. Now, these cloud customers use resources and remote servers of the cloud provider. Rather than have a physical machine, they now have a virtual machine (VM) that runs alongside other customers' VMs. The cloud provider runs a hypervisor, or a

J. Szefer (✉) • R.B. Lee
Department of Electrical Engineering, Princeton University,
Olden St., Princeton, NJ 08544, USA
e-mail: szefer@princeton.edu; rblee@princeton.edu

virtual machine monitor, that virtualizes the system and orchestrates the sharing of the physical resources so that it can support many VMs on one physical system. This way, many customers' VMs can run on one server, consolidating resources and allowing the cloud provider to lease out the VMs to the customers at favorable prices. The cloud provider maintains many servers where customers' code and data are executed or stored. They also have management infrastructure including dedicated cloud management servers. This infrastructure is in place so that the customer's can easily provision and release computing resources. Customers often use the VMs to run some service (e.g., a web site) that is accessed by end users.

1.1 Security Concerns

While cloud computing provides many economic benefits, there are a number of new security concerns that need to be addressed. There are two main differences when using a virtual machine, versus executing or storing code and data on your own physical machine. First, there are the other customers' VMs that are running on the same system. These VMs should be properly isolated when running on top of a trusted hypervisor. Unfortunately, current hypervisors are susceptible to various vulnerabilities and bugs. A malicious cloud computing customer who is co-located on the same system as his or her competitor may attack the virtualization layer. Once the virtualization layer is compromised, its privilege level can be used to examine or obstruct other VMs. The second difference is the virtualization layer

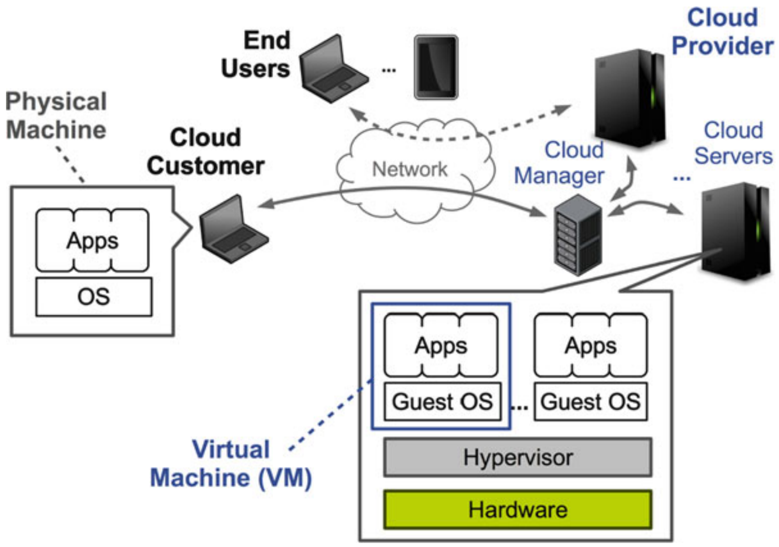

Fig. 1 The IaaS cloud computing model

itself, running underneath all the VMs. It is a privileged software layer which has access to all the resources of the system. It can affect confidentiality, integrity or availability of the different VMs. Normally, the hypervisor is trusted and is used to provide security for the rest of the system. If customers used virtualization on their own systems, they would know exactly the version and type of hypervisor used. In a cloud computing setting, however, customers have no control over the hypervisor. While the hypervisors are designed by reputable commercial vendors (e.g. VMWare [2]) or open-source projects (e.g. Xen [3]), they are still susceptible to bugs and vulnerabilities. Unlike in your own facilities, where you will not likely attack yourself, in a cloud computing setting one of the co-located VMs may belong to a different customer who may have incentive to compromise the hypervisor. Moreover, some cloud providers may be coerced to install a malicious hypervisor to spy on certain victim customers' VMs.

1.2 Approaches to Securing Cloud Computing

These security issues have been recognized by various researchers and many have worked on different approaches to securing cloud computing. Since the hypervisor is the key virtualization technology needed for cloud computing, most have focused on securing the hypervisor. Researchers have looked at minimizing the hypervisor code size [15, 17], as the number of bugs or vulnerabilities is often correlated to the code size. Others have explored re-writing the hypervisor to harden it against potential attacks [26]. Work has also been done on protecting different parts of the hypervisor, such as protecting the core hypervisor from the management OS [11]. A key duty of the hypervisor is to isolate the different VMs and research has been done on improving the isolation [8, 14]. Some researchers have also attempted to come ahead of the threats, and analyze or introspect the VMs to try to find attacks before they actually happen [9, 13, 16, 27]. While these approaches improve the security of the system, they still require a trusted hypervisor to be present for correct secure operation of the system. These are all software-based approaches, so far.

Hardware security approaches have also been explored. One example is the Trusted Platform Module (TPM) [22], which is a co-processor used for security-related functionality such as measurement and attestation of the software stack. The TPM can help measure the software at load time, but not actively protect it during runtime. More powerful co-processors [7] provide physical tamper prevention and a secure execution environment inside the co-processor. Such co-processors allow for secure execution of applications, but not entire VMs. Commercial processors have also included new extensions inside the main processor, mainly to support the extra software layer, i.e., the hypervisor. This, however, assumes a trusted all-powerful system software which runs in this new privileged mode. Academic projects have also looked at hardware-enhancements to improve security. A number of architectures have been proposed [5, 6, 10, 12, 18], many of these architectures focus on protecting software modules.

Rather than protecting software modules in an application, below we will describe how new hardware architecture can be used to enhance system security by protecting an entire virtual machine in a cloud computing setting – against the hypervisor as an adversary. Unlike software, hardware is mostly immutable. Any security features introduced in hardware chips are thus very difficult, if not impossible, to alter after the chip is manufactured. Also, hardware is logically the lowest layer in the system. For example, a security feature implemented in an OS running inside a VM can be bypassed by a malicious or compromised hypervisor (which is logically below the VM); there is not a layer below the hardware that can bypass security features implemented in hardware. Unlike other hardware approaches, we focus on protecting the entire VM, and consider the aggressive new threat model of a malicious or compromised hypervisor.

In particular, we describe our proposed concept of *hypervisor-secure virtualization* [19–21]. Architectures implementing hypervisor-secure virtualization include new hardware for protecting the confidentiality and integrity of a VM's memory, even from the (previously) all-powerful hypervisor. Hypervisor-secure virtualization architectures allow for a hypervisor to manage many VMs per system, share processor cores among different VMs, or even oversubscribe memory resources. These are all the features that can be done today, but the key difference is that thanks to the new hardware additions, the hypervisor can be untrusted. Using such architectures brings customers closer to the goal of being able to run their virtual machines remotely, and be as secure as if they were running the OS and applications locally on their own physical machine.

2 Hardware-Enhanced Security with HyperWall

We have realized hypervisor-secure virtualization in our HyperWall architecture [19–21]. The architecture uses resource isolation (focusing on the memory of the virtual machines), as opposed to cryptographic isolation, to implement hypervisor-secure virtualization. The architecture enhances today's multi-core server architectures by introducing new hardware additions. These additions enable selected portions of a virtual machine's memory to be isolated from the hypervisor, from DMA (direct memory access) by peripheral devices, and from other virtual machines. HyperWall's target usage scenario is the Infrastructure-as-a-Service (IaaS) cloud computing model, presented in the introduction, but the other cloud computing models can be built on top of it as well.

With HyperWall, cloud customers can run their virtual machines on the hosted infrastructure. Simultaneously, the infrastructure provider can host many other customers' virtual machines and run a hypervisor that the customer need not trust for the confidentiality and integrity of his or her code and data, because of our hardware enhancements. To enable the customer to protect a VM's code and data, the cloud customers are given the means to provide some specification of the confidentiality and integrity protection they want for the data and code that will run inside the VM.

Given the VM image and the requested protections, a HyperWall-enabled server can start and measure the VM and the protections. The attestation of the initialized VM and protections is communicated to the customer, and once they verify that the correct VM and protections indeed started, they can establish a secure channel with the VM. Any sensitive code or data can now be sent to the VM through the secure channel and executed remotely. When the VM is finished, the hardware properly cleans up the protected memory. The architecture and the various stages of the operation are discussed below, after the threat model.

2.1 Threat Model

The goal of HyperWall is to protect against confidentiality and integrity attacks by a malicious or compromised hypervisor. We want to retain an active hypervisor so that most of the features of today's cloud computing offerings can be supported (e.g., VM sharing processor cores or memory oversubscripiton). Thanks to the new, trusted hardware we can provide these protections. However, we do not consider attacks on availability, side-channels or physical attacks. In particular, a cloud provider needs a way to turn off VMs if customers stop paying or misbehave, so availability can not be guaranteed for customers. Side-channels are a separate research topic that has been explored [24, 25]. Hardware side-channel protections can be integrated with architectures such as HyperWall. We also expect a non-malicious infrastructure provider and secure facilities so that physical attack protection is not needed.

The protections focus on protecting entire VMs. The OS and applications inside a VM are assumed to be trusted by the customer. The customer is also assumed to know which memory regions inside the VM (in terms of guest physical pages) need protection, and the OS will not allocate sensitive code or data to the unprotected memory regions.

2.2 Memory Protection

With HyperWall, we opt for an isolation-based approach to memory protection where individual guest physical memory regions (e.g. consisting of multiple memory pages) are assigned to a virtual machine and the hardware enforces that only the owner virtual machine can access these pages. Memory isolation is not a new concept, but in today's commodity systems, the hypervisor software is relied on to provide and enforce the isolation. A powerful attack by the hypervisor is to give itself (directly, via a colluding VM, or via a device and direct memory access) the ability to snoop on some target VM's memory. To counter this, our hardware can enforce the memory isolation between the VMs and the hypervisor. Because hardware is logically below the hypervisor software, it can store protection specification data and contain security functionality which cannot be altered by the hypervisor.

Figure 2 shows the different memory regions in a HyperWall system. DRAM represents the machine memory. That memory is allocated to different VMs. In Fig. 2 we highlight different memory regions relevant to a protected VM.

First, there are the page table specifying address translation from guest physical pages (managed by the OS inside the VM) to the machine pages (managed by the hypervisor). This page table mapping is set by the hypervisor when it allocates memory for the VM. It is locked and protected by our new hardware when the VM is launched – thus preventing the hypervisor from updating the memory mapping without intervention by our new hardware. Memory update is discussed later in the chapter.

Second, there are the actual memory pages allocated to the VM and which have been loaded with the VM image (i.e., the code and data that makes up the OS and applications). These memory regions are defined by the page tables. The hardware uses the page tables to locate these pages and protect them according to the customer's specification.

Third, the customer's requested protections for the VM are a final part of memory which need to be protected. When the hypervisor loads the VM, it loads the VM images as well as these requested protections. The hardware needs to lock and protect these memory pages so that the hypervisor can not alter the requested protection data as the VM runs.

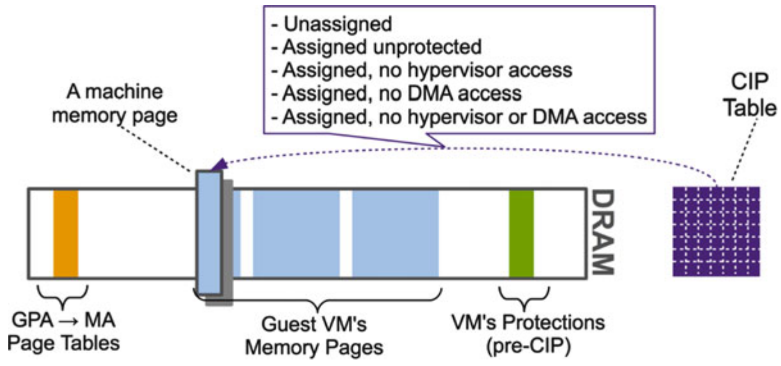
- Unassigned
- Assigned unprotected
- Assigned, no hypervisor access
- Assigned, no DMA access
- Assigned, no hypervisor or DMA access
A machine memory page
CIP Table
DRAM
GPA → MA Page Tables
Guest VM's Memory Pages
VM's Protections (pre-CIP)

Fig. 2 The Confidentiality and Integrity Protection (CIP) table and the different protected memory regions

2.3 Confidentiality and Integrity Protection Table

A new feature introduced in HyperWall is the Confidentiality and Integrity Protection table (CIP table). The CIP table, shown on the right of Fig. 2, store the protection information for all machine memory pages, for all the VMs. An interesting aspect of the CIP table, is that it is actually stored in DRAM (as shown on the right side of Fig. 3).

We re-use existing DRAM to store the CIP protection data, which eliminates the need for special memory structures inside the processor or other parts of the system. That portion of DRAM is made hardware-only accessible, and is off limits to the hypervisor, the VMs, and the devices. During system boot up, our new hardware locks part of the DRAM so no software can access it. This is a very flexible approach as, for example, the memory can be updated – just install more DRAM – and when the system is rebooted the hardware will allocate a proportionally sized portion to be hardware-only accessible and to store the CIP table. Now hardware has an exclusive memory storage region where it can keep protection information.

A customer's specified protections come from the requested protection information (also called pre-CIP). Given a guest physical address (from the page tables), the pre-CIP data can be looked up to check the protections needed for the corresponding page. The guest physical to machine address translation from the page tables can be used to obtain the machine address where the guest physical page is mapped into. This information can be combined and is stored in the CIP table.

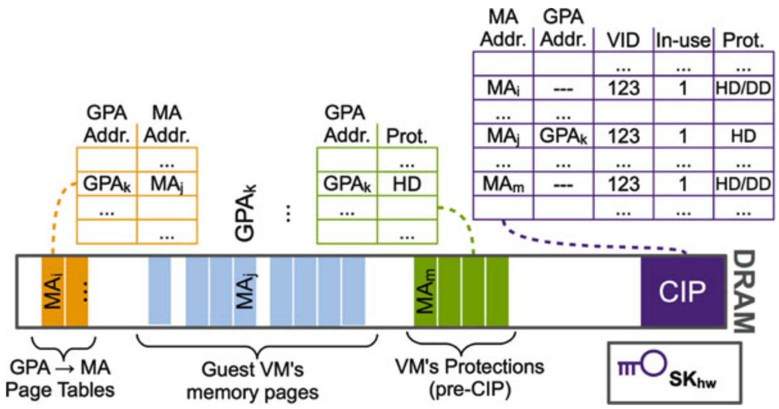

Fig. 3 Different tables and memory regions utilized by the HyperWall architecture: hypervisor-assigned page table is protected so the hardware knows the current memory mapping of the VM; the VM's memory itself is protected; the pre-CIP table is protected so the hypervisor cannot modify the requested protections. The CIP table stores the information about these three memory regions, and is stored in the hardware-only accessible memory. HD means Hypervisor Deny and DD means DMA Deny

Figure 3 shows more details of the different tables stored in the memory, along with sample guest physical addresses (GPA) and machine addresses (MA). The figure also shows a signing key, SK_{hw} that is unique to each processor supporting HyperWall architecture. Initialization and the use of the different memory regions, and the key, is discussed below.

2.4 Protecting Confidentiality and Integrity of VMs Under an Untrusted Hypervisor

There are three phases of the VM's execution during which memory needs to be protected by our new hardware to ensure the VM's confidentiality and integrity. First, the VM is initialized. Second, the VM runs. Finally, the VM is terminated.

VM Initialization

We have already discussed different memory regions relevant to a VM, shown in Fig. 2. These memory regions are loaded by the hypervisor before the VM is actually started. When the VM is launched, e.g., through the *vmlaunch* instruction, the new HyperWall hardware is triggered. The first duty of the hardware is to protect the memory regions by writing appropriate entries in the CIP table. The hardware assigns a VID (this VID is different from the VM identification assigned by the hypervisor, the hardware controls VIDs so the hypervisor cannot spoof them). The CIP table entries, as shown in Fig. 3 identify the owner VM of each machine memory page. For each page, the hardware first checks that the page is not in use (by using the machine address to index the CIP table and ensure that the page is free). If the page is free, the hardware assigns it to the VM. It writes the VID and marks the page as in-use. It also writes the protection information for this page, e.g., deny hypervisor accesses (HD) or deny DMA accesses from devices (DD). For the memory holding the page table and the requested protections, the memory is made inaccessible to the hypervisor and DMA.[1] For the memory pages of the VM, the hardware reads the page table entries and the requested protections information to see what protections were requested for the corresponding machine pages. If there is an error at any time, the VM launch is aborted.

Once all the memory pages are protected, and before the VM actually begins execution, the hardware calculates a cryptographic hash of the VM image and the requested protections. This is done because the (untrusted) hypervisor may have modified either of these before launching the VM. Once the memory pages are protected, the hypervisor cannot modify them and the contents of these pages can be measured. The measurements done by HyperWall are sent back to the customer. The measurements are cryptographically signed with the processor's signing key (shown as SK_{hw} in Fig. 3). The processor also has a digital certificate from the manufacturer. A customer, given reference measurements of the known good VM image and protections, signature of initialized VM's measurements, a hardware certificate and the hardware manufacturer's certificate can validate the received measurements. The measurement can be sent by the (untrusted) hypervisor and cloud infrastructure to the customer. We assume strong cryptography and that without access to the

[1] Note that the hardware checks that the page is not in use, so it is automatically not accessible to other VMs.

private portion of the cryptographic key, the signatures can not be forged. To ensure freshness and prevent replay attacks, the customer sends a nonce when requesting the VM, and that nonce is included in the measurements that are sent back to the customer.

VM Runtime

Once a VM is launched and running, the customer can start utilizing the protected environment offered by the HyperWall architecture. The first step is to establish a secure channel with the remote VM. Recall that the customer has already verified that his VM image and requested protections were started properly; or found out that they were not and stopped using that VM. Now the customer has the hardware certificate for the remote machine where their VM is running and the VID of their VM. We assume that the VM image originally contained no sensitive information, but only stock OS and common applications and libraries, e.g., OpenSSL library for cryptographic operations.

Establishing Secure Channel between Customer and VM Figure 4 shows how a VM running on a HyperWall system can establish a secure channel with the customer. The key to the secure communication is the VM's protected memory. Once the VM is launched and the memory is protected, it can generate a public-private key pair (EK_{vm}, DK_{vm}). The hypervisor has no access to the protected memory, so it can not see these newly generated keys. Next, the VM can use HyperWall's new *sign_bytes* instruction to sign the EK_{vm} key. The signature also includes the VID and a nonce that the customer sends. It is made with the hardware's signing key. A hypervisor or another VM can not spoof the signature as they

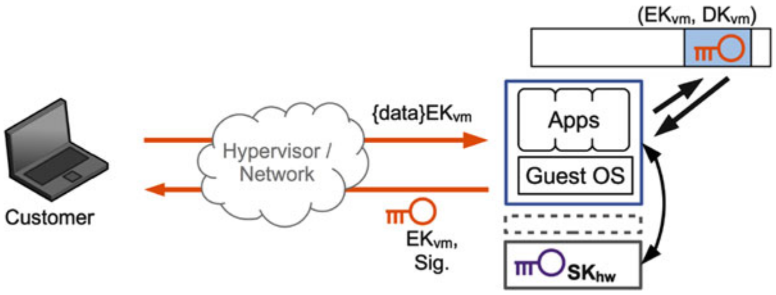

Fig. 4 Establishment of a secure communication channel with the VM

do not have access to the hardware signing key, SK_{hw}, and the VID is included automatically by hardware so other VMs cannot invoke the instruction and pretend to be a different VM. The key, EK_{vm}, and the signature are sent back to the customer. Once he or she verifies the signature, he or she can use the key to send sensitive code or data back to the VM (e.g., use it as part of a modified SSL protocol to establish a secure channel).

The code or data to be protected should be stored by the OS in the protected memory regions. Now, the code and data can execute, with the HyperWall hardware protecting the memory according to the customer's specification. Even a malicious or compromised hypervisor is not able to see into this memory.

Sharing Processor Cores Among VMs One of the features which makes cloud computing appealing is that many VMs can share the same system. Often, this requires scheduling more VMs than there are physical resources available, and switching between the VMs as they run. One key resource needed by VMs is the allocation of processor cores where the code actually executes. Scheduling many VMs on the same physical cores requires suspending them (when the hypervisor reads and saves processor state) and resuming them (when the hypervisor replaces saved state and triggers the VM to run again). It is critical to protect the VM when it is being suspended and resumed. The memory is already protected through the CIP table, even if the VM is suspended as the CIP table entries remain in the CIP until VM termination. The needed protections for suspending a VM are: protect the VM's virtual cores' state so that it is correctly resumed later, and protect the general-purpose registers which hold some of the code or data of the VM.

Figure 5 shows how a hypervisor could access contents of memory indirectly when it is reading the registers of a processor core. It could not only read the contents, but also modify them. To counter this, HyperWall encrypts general purpose registers[2] and generates a hash over the register state. When the VM is resumed, the hash is checked and the registers decrypted, if the hash verification succeeds.

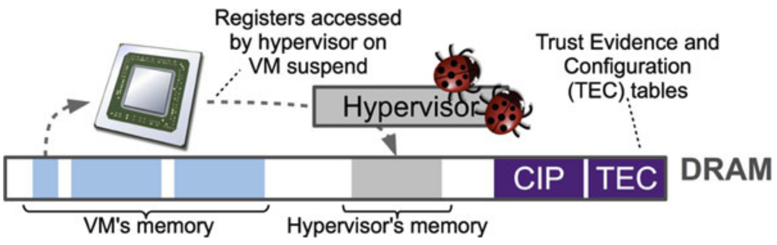
Registers accessed by hypervisor on VM suspend
Trust Evidence and Configuration (TEC) tables
Hypervisor
CIP
TEC
DRAM
VM's memory
Hypervisor's memory

Fig. 5 Memory contents, copied to processor registers when the code executes, could be read or modified by the hypervisor as it suspends a VM and copies registers to its own memory. HyperWall protects register state on VM suspend and resume with the help of the Trust Evidence and Configuration (TEC) table, which hold one entry for each VM

The original HyperWall architecture [21] has been improved to prevent certain types of replay attacks during VM suspend and resume [19]. In our improved design, a new set of Trust Evidence and Configuration (TEC) table is introduced, which is also stored in the hardware-only accessible memory. These TEC table are used to keep some attestation and configuration related information about each VM. In particular, for each VM (as identified by its VID, which is the same VID as

[2]If the VM suspend reason is a hypercall then the registers are not encrypted as they are used to pass arguments to the hypercall.

used in the CIP table), the TEC table store a counter of the number of times the VM has been suspended. Each time the VM is suspended, and before hypervisor code gets to run and read the registers, the counter is incremented. This counter value is also used in a cryptographic keyed-hash generated by the hardware. When the hypervisor gets to execute, all the general purpose registers are encrypted, if it is not a hypercall, and a special new register holds the hash value of the general registers' contents concatenated back-to-back. These can be stored anywhere by the hypervisor.

When the VM is to be resumed, the hypervisor writes the register values and the hash value into processor registers. Before VM code starts to run, the hardware checks the values. If the hypervisor were to try to modify the register values or the hash, the hardware can detect it when it regenerates the hash and compares the values. Also, the hypervisor can not modify the suspend count stored in the TEC table, thus it cannot replay an old set of register values.

Memory Oversubscription In addition to being able to share processor cores, memory oversubscription is another key feature of many cloud computing deployments which our architecture supports. Memory ballooning [23] is a technique for dynamically changing the allocation of memory of a guest virtual machine, while the machine is running. Ballooning depends on the hypervisor's ability to dynamically change the guest physical to machine memory mapping of a VM as it runs.

Figure 6 shows conceptually the idea of memory ballooning. As an example, suppose a system has a total of 6 GB of memory, however, the administrator oversubscribes memory by allocating to each of two customers up to 4 GB of memory each for their VMs. In Fig. 6a we see that each VM initially has only 3 GB of memory allocated and a "balloon" taking up 1 GB of memory. At runtime, the hypervisor can cooperate with a "balloon driver" inside each VM, to change the memory allocation. If the first VM requires more memory, the hypervisor can take some memory pages away from the other VM (inflate its ballon) and give these pages to the first VM (deflate its balloon). This is shown in Fig. 6b. The reverse is shown in Fig. 6c. By adjusting the memory allocation as needed, many more VMs can run on the system than there are actual resources.

The key operation which allows memory ballooning is the ability of the hypervisor to change the memory allocation during a VM's runtime, i.e., change the page tables mapping guest physical addresses (GPA) to machine addresses (MA). The hypervisor can change the mapping and remove pages (i.e., inflate the balloon) or add new pages (i.e., deflate the balloon). The problem, however, is that a malicious hypervisor may try to read the contents of the memory pages it just removed from a VM, potentially leaking the VM's code or data. Alternatively, a hypervisor may try to add memory mappings such that two VMs would share some memory pages, thus again potentially leaking code or data. The HyperWall hardware tightly controls the type of memory update that could be performed by the hypervisor while still allowing the hypervisor to change the guest physical address (GPA) to machine address (MA) mapping.

Three security requirements needed to ensure a VM's confidentiality and integrity protection during memory update are:

- Scrubbing of memory pages: a machine memory page that is to be freed should be scrubbed before it becomes free and can be allocated to another VM (to prevent leaks leading to confidentiality breaches),
- No adding of in-use pages: a VM should not be allocated machine memory pages already in use by another VM (to prevent another VM from compromising the confidentiality or integrity of a victim VM's memory), and
- No swapping of pages within a VM: during the memory update, a VM's guest physical to machine memory page mappings should not be swapped (to prevent integrity breaches, and potential confidentiality breaches, where the hypervisor can swap memory contents).

To perform the memory update in HyperWall, the hypervisor specifies a new page table mapping. It then suspends a VM, and writes a pointer to the new page table mapping. On VM resume, the hardware can compare the page table pointer to recognize the changed value. This triggers the memory update. Now, hardware checks the new page tables and compares them to the contents of the CIP table.

The hardware can use the CIP table to recognize if a machine page is already in use. The VID in the CIP table entry is used to recognize the owner VM. If a hypervisor creates a new mapping, and assigns a new page to the VM, the hardware can check the VID in the CIP entry for this page – if the VID is not null, then the page is already in use and cannot be added. The update must abort. If there is no error, the hardware can read the requested protections data for the new page, and set the protections in the CIP table accordingly.

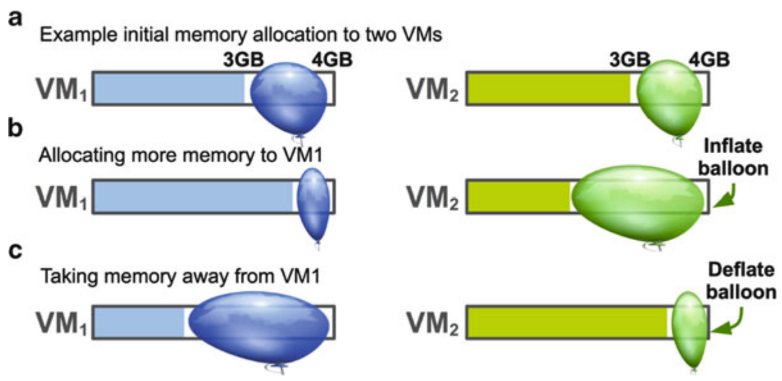
a
Example initial memory allocation to two VMs
3GB
4GB
VM₁
3GB
4GB
VM₂
b
Allocating more memory to VM1
VM₁
Inflate balloon
VM₂
c
Taking memory away from VM1
VM₁
Deflate balloon
VM₂

Fig. 6 Memory ballooning example

For pages that are to be deleted, we introduce a new "to-be deleted" bit in the page tables; the hypervisor marks pages to be deleted with this bit and the hardware can easily recognize which pages to delete. The hardware can compare the VIDs to make sure the page is indeed currently in use by the VM, then it can scrub the machine memory page and clear the CIP table entry to mark the page as free.

Memory swapping attempts can be recognized by comparing the guest physical page address (GPA) from the page tables, to the guest physical page address in the CIP table. If the page is present in the page tables (not a new added page) the hardware can read the CIP table entry to make sure that the machine page in the CIP table is for the guest physical page. If this reverse mapping does not match, then there is an attempt to swap pages and the update must be aborted.

VM Terminate

When the VM is terminated, its memory needs to be reclaimed. This can be done by the hypervisor by issuing the new *vmterminate* instruction with the VID being the identification of the VM that is to be terminated. The HyperWall state machine intercepts this instruction and begins VM termination. The HyperWall hardware traverses the page tables mapping to find all pages used by the VM. After each protected page is zeroed out by hardware, its entry in the CIP table is cleared so that this memory page can be freely accessed again. Once all the memory pages are removed from being protected, the memory holding the protection data needs to be unprotected. Then, the memory holding the page table mappings needs to be unprotected as well. Finally, all the entries for the VM in the TEC table are cleared. This clears and returns the memory as well as makes the VID number available for another VM to use. If a hypervisor fails to issue the *vmterminate* instruction or otherwise misbehaves, it remains locked out of the protected memory – this is a loss of availability of these memory pages, but no code or data is leaked.

3 HyperWall Architecture Summary

Figure 7 shows the hardware and software modifications required to implement the HyperWall architecture and to support the operations described above. Also, Table 1 lists the new or modified instructions used by HyperWall. The new instructions (A), e.g. *vmterminate*, were introduced along with new registers (B), e.g. the VM_suspend_hash register. A cryptographic engine (C) is needed for performing encryption, decryption, hashing and signing (using the SK_{hw} key). A hardware random number generator (D) is added to support the new *trng* instruction used in secure channel establishment. The bulk of the HyperWall logic is in a state machine (E) which is responsible for updating the CIP and TEC tables when a VM is created, updated or terminated. In particular, the state machine ensures the protections are maintained when the memory mapping for a guest VM is updated. This is done by the hardware mediating updates to the guest physical address to machine address page table mappings. The TLB logic (F) is expanded to consult the CIP table before inserting an address translation into the TLBs. To improve performance, the access checks are done when the address translation is performed during the handling of a TLB miss. If there is no violation, the address is cached in the TLB

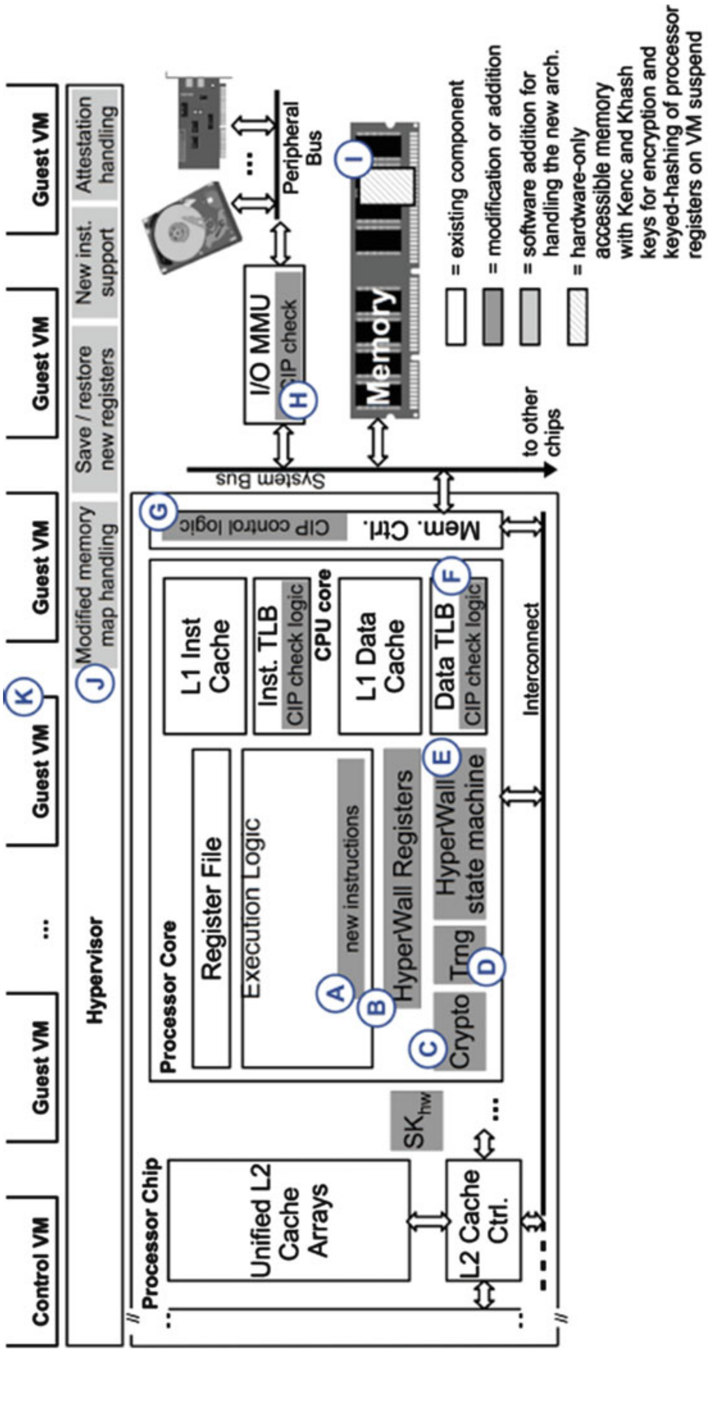

Fig. 7 Summary of the hardware, and related software, modifications in the processor needed to support the HyperWall architecture

Table 1 Summary of new or modified instructions in HyperWall architecture

Instruction (Inputs)	Description
`generate_trust_evidence ( VMID )`	Request current trust evidence of VM with VID to be copied into processor registers
`sign_bytes ( Addr, Size )`	Use hardware's private key to sign specified data
`trng ()`	Access true random number generator to retrieve 64 bits of randomness
`vmterminate ( VID )`	For a VID, signal hardware to scrub the VM's memory and terminate the VM
`vmlaunch ()`	Existing instruction, modified to trigger our HyperWall mechanisms on VM launch

and the CIP table check can be avoided in the future, for a TLB hit. To prevent stale mappings, however, the TLBs need to be flushed whenever the CIP table are updated. The memory controller (G) is updated with configuration registers and CIP control logic to walk the CIP table on a hypervisor or DMA access. Similarly to the address translation in the main processor, the I/O MMU (H) needs to have extra logic to consult the CIP table. We re-use a portion of DRAM (I), the hardware-only accessible memory, to store the CIP and TEC tables.

While HyperWall is a hardware architecture, the software needs to be modified to interact with the new modified hardware. The hypervisor (J), as the entity in charge of the platform, needs to interact with our new hardware architecture. It needs to save and restore the new registers when VMs are interrupted and resumed (as it does already with other state today). It needs to use a modified procedure for updating the memory mapping during VM runtime (i.e., specify a new page table mapping, rather than modify individual entries in the old page table mapping). During a VM's runtime, it needs to issue our new *generate_trust_evidence* instruction to read the trust evidence data and return it to the customer when requested. When terminating the VM, it needs to issue our new *vmterminate* instruction.

The guest VM (K) needs small modifications to use the *trng* instruction for obtaining randomness (rather than from other means, such as from interrupts, that could be controlled by the hypervisor). It also needs to use the *sign_bytes* instruction to get information, e.g. an attestation report, signed by the hardware before sending it to the customer. The OS should properly load code and data so that sensitive code or data are never placed in the unprotected guest physical memory regions.

4 Trust Evidence

An interesting new feature introduced in HyperWall is the trust evidence it collects and can provide to the customer. We already discussed how at VM launch, the HyperWall hardware measures the VM image and the requested protections. This information is digitally signed and can be sent back to the customer for verification. More interestingly, however, HyperWall also performs measurements at VM runtime. In particular, we introduced new counters, akin to performance counters, which keep track of attempts to violate memory protections. For each VM, there is a set of trust evidence counters ($VCNT$ and $VMAD$, described below), stored in the Trust Evidence and Configuration table (TEC table).

As the VM runs, the hardware protects its memory from hypervisor and/or DMA accesses (Direct Memory Access from/to I/O devices). From Fig. 8, we can see that the hypervisor or DMA could attempt to access the VM's memory. The hardware intercepts and blocks such accesses (if the memory is specified to be protected in the CIP table). Moreover, when such a malicious (or erroneous) access is detected, the hardware counters associated with the VM are incremented. There is the $VCNT$ counter which keeps track of the number of attempted violations. There is also the $VMAD$ register, which keeps track of the last memory address where an attempted memory access violation occurred.

These measurements are digitally signed by the hardware, again using the SK_{hw} key and can be sent back to the customer, upon the customer's request. The customer can then use this information to examine the state of the system. While the hardware protects the VMs, if there is a large number of attempted violations, the customer may choose to stop utilizing that VM, as something suspicious is happening on the remote system.

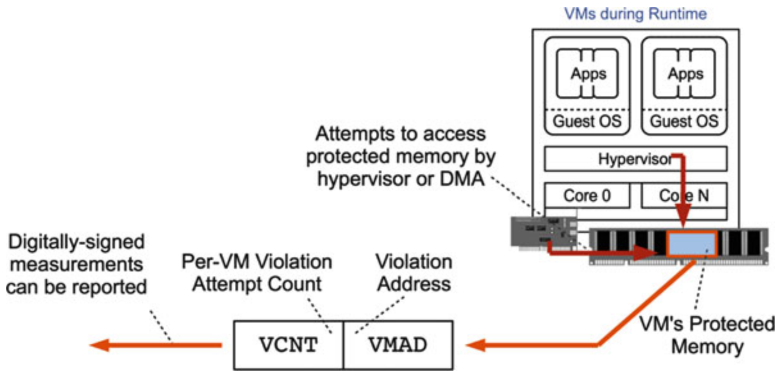
VMs during Runtime
Apps
Apps
Guest OS
Guest OS
Hypervisor
Core 0
Core N
Attempts to access protected memory by hypervisor or DMA
Digitally-signed measurements can be reported
Per-VM Violation Attempt Count
Violation Address
VCNT
VMAD
VM's Protected Memory

Fig. 8 New trust evidence mechanisms keep track of attempts to violate memory protections. These attestation measurements are digitally signed by the hardware and can be reported to the customer for checking

5 Further Research Directions

HyperWall provides a significant step towards making computing in the cloud as secure as in your own dedicated facilities. But many research challenges remain. In particular, can computing in the cloud be made even more secure than on your own machine?

There are many other interesting and relevant threat models, where research in hardware-enahanced security architectures could yield significant improvements in system security. Figure 9 shows a few of these different threat models. On the left in Fig. 9a we show a conventional trust chain, where each level of the software stack must be trusted by the level above. This is the case today, where the hardware is assumed trusted, the hypervisor is assumed trusted, and the OS is also trusted, in order to run trusted software applications securely. Architectures built on this threat model require trust in all the different software components, in addition to the hardware. TPM [22] or ARM's TrustZone [1] assume this threat model.

Moving to the right, Fig. 9b shows a threat model where the hypervisor is trusted, but the OS is untrusted. A cloud provider can run a trusted hypervisor as the virtualization layer, and try to provide protection for its customers' applications. But the cloud provider may not want to have to trust today's bloated, commodity OSes which are vulnerable to bugs. Bastion [4, 5] architecture is one example of a hardware-software security architecture which assumes this threat model. Bastion's strategy is to combine software flexibility (it uses a trusted hypervisor to protect and manage the TSMs) with hardware immutability and performance (to protect the hypervisor). For example, Bastion's hardware offers mechanisms for the secure launch of the hypervisor, as well as for protecting the hypervisor during runtime. Bastion's trusted hypervisor in turn protects the Trusted Software Modules (TSMs); it can securely launch TSMs during runtime, perform secure memory management, provide secure inter-module control flow, and provide secure storage, in addition to providing runtime memory integrity and confidentiality protection against physical attacks. One of the key features of Bastion is its tailored attestation. Unlike HyperWall's trust evidence which gives information about an entire VM, Bastion's attestation can provide information about individual TSMs. Future work

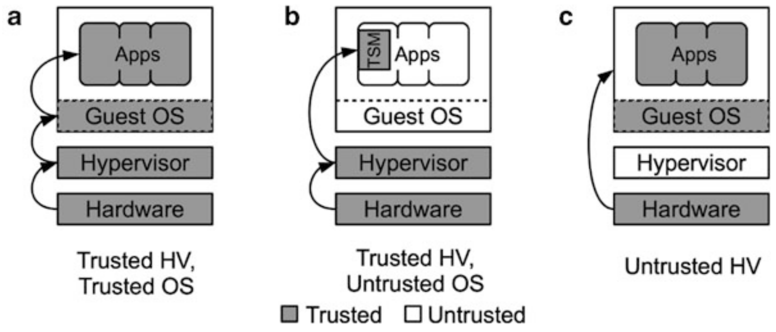

Fig. 9 Figure showing different threat models, dark-gray components are trusted; HV is the hypervisor, OS is the operating system, TSM is a Trusted Software Module

could look at how to partition the applications into the TSMs (something that is currently done manually and in an ad-hoc manner today), as just one example.

Another threat model relevant to cloud computing is shown in Fig. 9c. Here the guest OS is assumed trusted (by the customer of the virtual machine in which it runs), but the hypervisor may be untrusted or compromisable. This is a likely situation for a customer who has fully tested his own trusted applications and trusted OS, but is hoping to run this in a virtual machine to benefit from the lower cost and flexibility of cloud computing, where he has no control of the hypervisor. We described HyperWall as one example of an architecture which fits this threat model. Future work could explore how to do even more layer-skipping of untrusted software layers – yet still have a secure trust chain, by using new hardware security mechanisms. Prior to cloud computing, hardware security architectures explored protection of applications by hardware, but did not consider cases that involve a hypervisor [6,10,12,18]. Also, ideas of Bastion's tailored attestation could be combined with HyperWall's trust evidence to provide even better attestation mechanisms.

In addition to providing secure cloud servers, future research should also address security in the client devices that use cloud computing resources. Mobile devices like smartphones and tablets are the portals into cloud computing, and can access all kinds of important and sensitive information though the cloud. Hence, security in client devices is an important research direction, and very relevant to secure cloud computing.

As important as designing new hardware-enhanced security architectures is the security testing and verification of these new architectures and new hardware-software mechanisms. Security testing with known attacks is invaluable, but it can only show the presence of certain security vulnerabilities – not the absence of all exploitable vulnerabilities. Security verification tries to show that security properties hold, or will not be violated. While it may be able to leverage some tools from functional verification, security verification has additional requirements. Hence, research towards a systematic methodology and tools that enable security verification at design time can go a long way to providing better security in tomorrow's computing systems.

6 Summary and Further Readings

We have defined *hypervisor-secure virtualization* architectures and described our HyperWall architecture as an example. HyperWall uses new hardware features to protect the confidentiality and integrity of a VM's memory from an untrusted or malicious hypervisor. With HyperWall, a hypervisor, while untrusted with respect to the confidentiality and integrity of the VMs' memory, is still able to run and perform management duties, such as sharing processor cores among VMs or performing runtime memory reallocation. Having an untrusted hypervisor is an aggressive new threat model, not previously tackled by other architectures, which almost always assume a trusted hypervisor. Our new CIP (Confidentiality and Integrity Protection) table and new hardware mechanisms ensure that the memory of the VMs is protected and the untrusted hypervisor cannot maliciously alter these protections. Hence we can allow an untrusted, commodity hypervisor to run, thus providing rich runtime functionality for the VMs. We also introduced the concept of the hardware-only accessible DRAM memory, which is used to store the CIP table and the TEC (Trust Evidence and Configuration) table for the VMs.

Interested readers are encouraged to read the original paper describing hypervisor-secure virtualization [20]. Details of the HyperWall architecture are available in a conference paper [21]. Improvements and updates, as well as full details of the architecture, are available in [19]. Other hardware-enhanced approaches to security are also discussed in [4–7, 10, 12, 18, 22, 24, 25].

Acknowledgements This work was supported in part by NSF grants CNS-1218817, CCF-0917134 and EEC-0540832.

References

1. ARM, TrustZone. http://www.arm.com/products/processors/technologies/trustzone.php, accessed April 2013.
2. VMWare. http://www.vmware.com/, accessed April 2013.
3. Xen. http://www.xen.org, accessed May 2013.
4. David Champagne. *Scalable Security Architecture for Trusted Software*. PhD thesis, Princeton University, 2010.
5. David Champagne and Ruby B. Lee. Scalable architectural support for trusted software. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture*, HPCA, pages 1–12, 2010.
6. Jeffrey S. Dwoskin and Ruby B. Lee. Hardware-rooted trust for secure key management and transient trust. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 389–400, 2007.
7. Joan G. Dyer, Mark Lindemann, Ronald Perez, Reiner Sailer, Leendert van Doorn, Sean W. Smith, and Steve Weingart. Building the IBM 4758 Secure Coprocessor. *Computer*, 34:57–66, 2001.
8. Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. *SIGOPS Oper. Syst. Rev.*, 37(5):193–206, 2003.
9. Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings Network and Distributed Systems Security Symposium*, pages 191–206, 2003.
10. Ruby B. Lee, Peter Kwan, John Patrick McGregor, Jeffrey Dwoskin, and Zhenghong Wang. Architecture for protecting critical secrets in microprocessors. In *Proceedings of the International Symposium on Computer Architecture*, ISCA, pages 2–13, 2005.
11. Chunxiao Li, Anand Raghunathan, and Niraj K. Jha. Secure virtual machine execution under an untrusted management OS. In *Proceedings Conference on Cloud Computing (CLOUD)*, pages 172–179, 2010.
12. David Lie, John C. Mitchell, Chandramohan A. Thekkath, and Mark Horowitz. Specifying and verifying hardware for tamper-resistant software. In *Proceedings of Symposium on Security and Privacy*, S&P, pages 166–177, 2003.
13. Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In Richard Lippmann, Engin Kirda, and Ari Trachtenberg, editors, *Recent Advances in Intrusion Detection*, volume 5230 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 2008.

14. Reiner Sailer, Enriquillo Valdez, Trent Jaeger, Ronald Perez, Leendert Van Doorn, John Linwood Griffin, Stefan Berger, Reiner Sailer, Enriquillo Valdez, Trent Jaeger, Ronald Perez, Leendert Doorn, John Linwood, and Griffin Stefan Berger. sHype: Secure Hypervisor Approach to Trusted Virtualized Systems. Technical Report RC23511, IBM Research, 2005.
15. Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. *SIGOPS Oper. Syst. Rev.*, 41(6):335–350, 2007.
16. Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure in-vm monitoring using hardware virtualization. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 477–487, 2009.
17. Udo Steinberg and Bernhard Kauer. NOVA: A microhypervisor-based secure virtualization architecture. In *European Conference on Computer Systems*, pages 209–222, 2010.
18. G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual International Conference on Supercomputing*, ICS '03, pages 160–171, 2003.
19. Jakub Szefer. *Architectures for Secure Cloud Computing Servers*. PhD thesis, Princeton University, 2013.
20. Jakub Szefer and Ruby B. Lee. A Case for Hardware Protection of Guest VMs from Compromised Hypervisors in Cloud Computing. In *Proceedings of the Second International Workshop on Security and Privacy in Cloud Computing*, SPCC, pages 248–252, 2011.
21. Jakub Szefer and Ruby B. Lee. Architectural Support for Hypervisor-Secure Virtualization. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 437–450, March 2012.
22. Trusted Computing Group Trusted Platform Module main specification version 1.2, revision 94. http://www.trustedcomputinggroup.org/resources/tpm_main_specification, accessed April 2013.
23. Carl A. Waldspurger. Memory resource management in VMware ESX server. In *5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 181–194, 2002.
24. Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual International Symposium on Computer Architecture*, ISCA '07, pages 494–505, 2007.
25. Zhenghong Wang and Ruby B. Lee. A novel cache architecture with enhanced performance and security. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 83–93, 2008.
26. Zhi Wang and Xuxian Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, S&P, pages 380–395, May 2010.
27. Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS, pages 545–554, 2009.