Architectural Support for Hypervisor-Secure Virtualization

Jakub Szefer
Princeton University
szefer@princeton.edu

Ruby B. Lee
Princeton University
rblee@princeton.edu

Abstract

Virtualization has become a standard part of many computer systems. A key part of virtualization is the all-powerful hypervisor which manages the physical platform and can access all of its resources, including memory assigned to the guest virtual machines (VMs). Continuing releases of bug reports and exploits in the virtualization software show that defending the hypervisor against attacks is very difficult. In this work, we present *hypervisor-secure virtualization* – a new research direction with the goal of protecting the guest VMs from an untrusted hypervisor. We also present the HyperWall architecture which achieves hypervisor-secure virtualization, using hardware to provide the protections. HyperWall allows a hypervisor to freely manage the memory, processor cores and other resources of a platform. Yet once VMs are created, our new Confidentiality and Integrity Protection (CIP) tables protect the memory of the guest VMs from accesses by the hypervisor or by DMA, depending on the customer's specification. If a hypervisor does become compromised, e.g. by an attack from a malicious VM, it cannot be used in turn to attack other VMs. The protections are enabled through minimal modifications to the microprocessor and memory management units. Whereas much of the previous work concentrates on protecting the hypervisor from attacks by guest VMs, we tackle the problem of protecting the guest VMs from the hypervisor.

Categories and Subject Descriptors C.0 [*General*]: System Architectures

General Terms Security, Design

Keywords Computer Architecture, Cloud Computing, Security, Virtualization, Hardware Security, Hypervisor, Trust Evidence, Attestation, Confidentiality, Integrity

1. Introduction

We introduce the HyperWall architecture, which provides protections to guest VMs from attacks by a malicious hypervisor. We are interested in the increasingly popular IaaS (infrastructure-as-a-service) cloud computing model where the infrastructure provider, such as the Amazon EC2 service [1], maintains physical servers and leases VMs to the customers. While the infrastructure provider provides the hardware and the virtualization software, the customers provide their own guest operating system (OS) and applications to run inside the leased VMs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASPLOS'12, March 3–7, 2012, London, England, UK.
Copyright © 2012 ACM 978-1-4503-0759-8/12/03...$10.00

Cloud customers leasing VMs from an IaaS provider can have a more flexible computing infrastructure while also reducing their operational costs. However, adversarial parties or competitors can lease VMs from the same provider. These competitors may then have incentive to attack the hypervisor and use it to gain information about other VMs.

Past work in this area concentrated on hardening the hypervisors and protecting them from attacks. Even the hardened or minimized hypervisors, however, have memory management functionalities which they use to access the guest VM's memory. If the hypervisor is compromised, the management functions can be abused to violate the confidentiality and integrity of the guest VMs. Our work then aims to protect guest VMs from a compromised or malicious hypervisor. We call this *hypervisor-secure virtualization*.

HyperWall's key feature is the Confidentiality and Integrity Protection (CIP, pronounced "sip") tables, which are only accessible by hardware; they protect all or portions of a VM's memory based on the customer's specification. The customer can specify which memory ranges require protection from accesses by the hypervisor or DMA (Direct Memory Access). The tables essentially provide for a hardware-controlled protection of memory resources.

In HyperWall, the hypervisor is able to fully manage the platform, to start, pause or stop VMs or to change memory assignment. To cross-check that the hypervisor is not behaving maliciously, our architecture provides signed hash measurements to attest that the customer's initial VM image and specified protections were indeed instantiated at VM launch. Furthermore, trust evidence can be generated during the VM's lifetime to verify that the specified VM protections have not been violated, or indicate how many attempts at violating these protections have occurred. When the VM is terminated, its protected memory is zeroed out by the hardware to prevent leakage of data or code, and the hypervisor and DMA regain full access rights to the VM's memory.

HyperWall aims to extend a modern 64-bit shared-memory multi-core and multi-chip system. By building on a base architecture which supports virtualization, many elements can be re-used to minimize the modifications required to support hypervisor-secure virtualization. The main contributions of this work are:

- definition of *hypervisor-secure virtualization*,
- the HyperWall architecture which achieves *hypervisor-secure virtualization* by using hardware to isolate a VM's memory from malicious hypervisor or DMA accesses, in page-sized granularity as specified by the cloud customer, while allowing a commodity hypervisor to fully manage the platform,
- introduction of a general-purpose mechanism for hardware protection of memory (and other resources) that scales with memory size, e.g., used in HyperWall for the CIP tables, and
- formulation of trust evidence mechanisms which provide the cloud customer assurance that protections he or she specified were installed and are being enforced.

Table 1. Different hypervisors and OSes, their SLOC count and the components which are part of their TCB.

Hypervisor [44]	**SLOC**	**TCB**
Xen-4.0	194K	Xen, Dom0
VMware ESX	200K	VM kernel
Hyper-V	100K	Hyper-V, Windows Server
OS [25, 27, 34]	**SLOC**	**TCB**
Linux 2.6.23	13500K	OS kernel
Window XP	40000K	OS kernel
seL4	8K+	OS kernel

1.1 Excluding Hypervisor from TCB

Many of today's security problems with virtualization solutions occur because of the size of the Trusted Computing Base (TCB). Table 1 shows a number of hypervisors and OSes, their code size and the components that are part of their TCB. While hypervisors are two orders of magnitude smaller than commodity operating systems, they are still about an order of magnitude larger than seL4 [25], a formally verified micro-kernel. Hence, today's hypervisors are probably too large for formal verification for the foreseeable future. Also, in spite of developers' efforts to provide bug-free code, a search on NIST's National Vulnerability Database [7] shows a number of bugs for both Xen [14] and VMware ESX [12]: 98 and 78 were listed, respectively, when a search was performed in the summer of 2011. An example of a concrete attack on a hypervisor found in the database is a vulnerability in Xen which allowed arbitrary commands to be executed in the control domain via a specially crafted configuration file [3]. Because of difficulties in implementing bug-free hypervisors, our work concentrates on protecting guest VMs from a compromised hypervisor.

1.2 Enlisting Hardware Protections

We now discuss why software-only solutions may not be sufficient to protect VMs. In a software-only approach, the hypervisor could use encryption so that different views of the memory are presented to different entities [20]. If the protection is in the hypervisor, however, then the hypervisor has to be in the TCB. Exploits and vulnerabilities which are being constantly discovered in real-life hypervisors indicate that securing a commercial hypervisor is an open issue. If the hypervisor is subverted, the protections can not be ensured.

A number of minimized and hardened hypervisors have been designed. Such systems, however, can still be misconfigured or bypassed. For example, HyperSentry [15] shows how to use the system management mode (SMM) to bypass the hypervisor for introspection purposes, but such functionality can also be used for malicious purposes.

Nested virtualization, e.g. the Turtles project [16], could be used where there is an extra hypervisor layer below the main hypervisor. That software layer could be used for security and implement HyperWall-style protections. What is to prevent a malicious attacker from slipping in an extra layer below the current lowest layer?

Researchers have also demonstrated Ring -1 Hypervisor rootkits [38], Ring -2 SMM rootkits [47] or even Ring -3 Chipset-based rootkits [41]. These rootkits cleverly use different functionalities of hardware, highlighting the fact that software protections are vulnerable since they can be subverted if attacked from below, i.e. by using the hardware. However, these rootkits are not due to any problems with hardware, rather they are due to the problem of having protections in software while it is the hardware which is logically the lowest level in the system.

By enlisting hardware protections, we also capitalize on hardware having performance advantages over software. In addition, changing functionality implemented by hardware or probing the microprocessor chip to recover secrets are extremely difficult. Furthermore, various vendors already modify hardware to improve virtualization through extensions which are widely deployed in commodity microprocessors [2, 5]. Extensions in various hardware subsystems are also available today: the I/O MMU [13] or the SR-IOV [9] extension for virtualizing PCI devices. This indicates that the commercial vendors do not see the costs of modifying the hardware for enhancing virtualization performance and security as outweighing the benefits of the modifications.

1.3 Hypervisor-Secure Virtualization

We have made a case for hardware protection of guest VMs from compromised hypervisors in a position paper [40] which outlined a broad research agenda, but no specific solutions. In this work, *we define hypervisor-secure virtualization as protection of a VM's secret code and data from an attacker with hypervisor-level privileges, and provide a concrete architectural solution.* The focus of the protections is on the memory, as this is where code and data resides, during a VM's execution. Moreover, any keys for cryptographically securing communications and persistent storage will be placed by the VM in memory. By controlling which memory regions the hypervisor or DMA can access, the confidentiality and integrity of code, data and keys can be protected, thus protecting VM runtime execution as well as its cryptographically-secured network channels and storage.

The rest of the paper is organized as follows. Section 2 provides our threat model and assumptions. Section 3 describes the HyperWall architecture and explains the details of how a HyperWall system operates. Our implementation and evaluation is presented in Section 4, and the security analysis in Section 5. Section 6 describes related work, and Section 7 concludes the paper.

2. Threat Model and Assumptions

Consistent with the definition of hypervisor-secure virtualization, the HyperWall implementation aims to protect the confidentiality and integrity of a VM's code and data. The cloud customer is able to specify which memory pages inside the guest VM require the protection. These protections are installed in hardware on the HyperWall system where the VM is to run and the hardware enforces the protections. Throughout, however, a commodity hypervisor (with minimal additions for handling our new hardware state) is able to manage the platform. For example it can dynamically change the physical to machine memory mapping.

Considering that we enlist hardware to provide the protections, we assume that the hardware is trusted, is correctly implemented and its functionality has not been tampered with. Some of the "hardware" functionality may actually be implemented as code or microcode stored in read-only memory (ROM). This is also assumed to be correct.

It is not our goal to protect against bugs in the guest OS or the applications. Our work aims to provide a customer with a guarantee that running a VM in the cloud is as secure as running it in the customer's own facilities (if not more secure due to a cloud provider's ability to provide better physical security). It is up to the customer to verify that the OS and applications do what the customer expects. Also, we do not aim to provide protection from attacks by the OS on applications. Verifying the security of guest OS and applications are both assumed to be the responsibility of the customer.

Preventing DoS (denial-of-service) attacks by the hypervisor on a VM's availability is not covered in this work. Moreover, the cloud provider needs a method for terminating a VM if a customer does not pay for the leased resources. However, confidentiality should

still be protected – hardware zeroes out the state and memory so no information is leaked when a VM is terminated.

We assume that the SMM, or similar remote management mode, is disabled in the hardware so that new security mechanisms cannot be bypassed. Incorporating security measures in SMM is an orthogonal research problem. For example, the STM (SMM Transfer Monitor) [42] has been designed for Intel microprocessors, but this is not widely deployed.

Hardware attacks (e.g. snooping on the memory bus) are not in the threat model. This assumption is consistent with the fact that the IaaS providers have an incentive to secure their facilities and deter insider attacks. Cloud providers can afford to physically secure the servers, install monitoring equipment, etc., to prevent an intruder from installing a hardware probe, for example.

We do not aim to protect against side channels such as those due to a processor's functional units, caches or I/O buses. Various work has been done on minimizing such channels (e.g. dynamically randomized caches [45]). Debug registers or performance counters could be another avenue for hypervisor or malicious VMs to glean information about a target VM. We do not protect against these covert channels and side channels in our current work.

3. HyperWall Architecture

HyperWall's objective is to protect guest VMs from a compromised hypervisor. We use standard encryption techniques to protect confidential code and data in persistent storage and during I/O and networking. We also protect the processor state, e.g. the general-purpose registers, when a VM is interrupted or suspended. The protection of memory, however, is achieved through the new hardware-controlled isolation. The isolation approach incurs less overhead than using a cryptographic approach (no extra decryption and hashing overhead when VMs access memory). Also, memory read and write paths do not need to be modified with extra cryptographic hardware. Moreover, since hardware attacks are not considered, encryption to hide contents of DRAM is not needed. Instead, the isolation of the protected VM memory from hypervisor and DMA access ensures its confidentiality and integrity.

The protections are accomplished through our architecture which:

- enables cloud customer specification of desired confidentiality and integrity protection policy (at page-sized granularity) to the cloud provider's HyperWall-enabled server,
- ties the protection policy to the VM initiated by the hypervisor on the provider's server,
- enables hardware enforcement of these policies through the addition of minor modifications of existing hardware and the re-use of existing commodity microprocessor features such as TLBs (Translation Lookaside Buffers), DRAM (Dynamic Random Access Memory), and the memory management units,
- enables the server to attest to the protections provided by the hardware to the cloud customer, and
- performs hardware-controlled cleanup of a VM's state and memory upon termination so no information is leaked.

3.1 VM Protection Mechanisms

In order to protect a VM's memory, we introduce new Confidentiality and Integrity Protection (CIP) tables to protect both the confidentiality and integrity of the VM's data and code (for the regions of memory which the customer requested to be protected).

The CIP tables are only accessible to the hardware; they store mappings of hypervisor and DMA access rights to the machine memory pages. These are used to enforce protections of the guest VM's memory. For each machine memory page, the CIP table stores protection bits which are used to encode the protection information about the page. The page can be: not assigned to any VM (not protected), assigned to a VM with hypervisor and DMA access allowed, assigned to a VM with hypervisor access denied, assigned to a VM with DMA access denied, or assigned to a VM with no hypervisor nor DMA access.

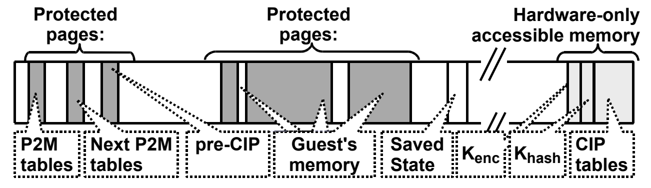

Figure 1. The main memory holds both the CIP tables (light gray) and other memory ranges (dark gray) which are protected by the CIP tables.

The default setting that a customer specifies for a VM can be to deny access to all pages, except for allowing the hypervisor or DMA access to the I/O buffers in a VM. At system boot up time, all bits in the CIP tables are zero, which gives the hypervisor and DMA access to all of the DRAM.

We note that it is not possible to just add additional access rights fields to the existing page tables because the hypervisor or the guest OSes need full access to their respective tables to manage memory mapping. Hence, a separate structure like our CIP tables is needed. In addition, we provide DMA protection, as a malicious hypervisor could configure a physical device to access the VM's memory through DMA and then read out that data from the device itself. Finally, if a page is not explicitly protected, it is marked as assigned to a VM but with hypervisor and DMA access allowed. This way a compromised hypervisor cannot maliciously map a machine page to two VMs so that an attacker VM could access a victim VM's memory.

CIP tables are consulted by a modified MMU (Memory Management Unit) in the microprocessor when hypervisor code is executing (or by logic in the I/O MMU on DMA accesses) to verify that a memory location can be accessed. The CIP tables are updated whenever a new VM is created or a memory assignment for a VM is changed. The CIP tables update will end with an error (a VM will not be started or new memory mapping will not be installed) if there is detection of a new VM being assigned memory already in use by other VM. The CIP tables are updated whenever a VM is terminated so pages can be reclaimed.

3.1.1 CIP Tables

We designed the architecture to use a portion of physical memory (DRAM) to store the CIP tables. This removes the need to add extra hardware storage for CIP tables in the microprocessor chip. It also allows the CIP table storage to be proportional to the amount of installed DRAM. Moreover, an important benefit of our approach of using DRAM memory for CIP tables is that it is a general-purpose, scalable, hardware mechanism that can be applied to many other uses.

When the computer boots up, the memory controller discovers the amount of physical memory and proportionally a section of it is reserved for CIP tables and made inaccessible to any software. Consequently, the memory controller announces a smaller DRAM size to the rest of the system. A single new register is needed in the memory controller to keep track of the end of visible DRAM; this register also marks the starting location of the CIP tables. The CIP logic can query this register to obtain the location of the tables so it can access them.

Figure 1 shows the host physical memory of the system. The part of the physical DRAM not accessible to any software is shaded light gray at the right of the figure. This region holds the CIP tables as well as an encryption key and a hash key, discussed below.

3.1.2 Protected Memory Regions

Other parts of DRAM are freely accessible unless they are protected by the CIP tables (sample protected regions for a VM are shaded dark gray in Figure 1). When a guest VM is running, the physical memory used by it is protected (as specified by the customer). A VM's memory does not have to be contiguous and the hypervisor is free to assign host physical memory to VMs by any algorithm, e.g. to avoid fragmentation.

Moreover, the CIP tables protect the memory which holds the page tables mapping the physical to machine memory (marked *P2M* tables) for the guest. During any memory mapping update, both the old mapping (*P2M* tables) and the new mapping (marked *next P2M* tables) are protected. A customer's requested protections are stored in the region marked *pre-CIP*. The pre-CIP specifies which pages are to be protected (hypervisor or DMA access denied) and which can be accessed by hypervisor or DMA. The pre-CIP needs to be saved so that the original requested protections can be checked each time the hypervisor attempts to update the guest's memory mapping.

The region marked *Saved State* in Figure 1 is shown to indicate that whenever the VM is interrupted, the (encrypted) general purpose registers, program counter, P2M page table pointer, and our new HyperWall register state need to be saved. Saving of VM state is already performed by today's hypervisors, and we only require that the code be updated to handle saving of the new state which we define. The saving and restoring of the new state is needed so that the CIP logic can verify the protections whenever the VM is resumed. The memory region holding the (encrypted and hashed) saved state is fully accessible to the hypervisor, e.g. so it can save it on disk when the VM is suspended.

3.1.3 Cryptographic Keys

Our CIP protection strategy is isolation through hardware access control, rather than through cryptographic methods; isolation is used to bar the hypervisor or DMA from accessing a VM's memory (as specified by the customer). We do, however, use cryptography to enable a customer to validate a genuine HyperWall server, to protect VM-related processor state as the hypervisor manages the VMs, and to protect communication between a VM and the external world, as is often done today. Hence, we discuss the cryptographic keys used in a HyperWall system.

The SK_{cpu} is a private signing key burned into the microprocessor chip and not accessible to any software (see Figure 2). If a system has multiple chip packages, each will have a different key. We envision that the hardware manufacturers would designate trusted certificate authorities which would sign the certificates of each chip's SK_{cpu}. Customers would obtain certificates for the chips of the servers where their VMs will run from the cloud provider. Customers would use the trusted certificate authority's public key to verify the certificates and thus whether the HyperWall hardware is genuine.

The K_{enc} and K_{hash} are two keys used by the HyperWall hardware and stored inside the protected memory (see Figure 1); the keys are generated during each boot up cycle. They are used whenever a VM is interrupted. Storing these keys in protected shared DRAM memory (rather than in new registers in a core) enables a VM to be paused on one core and resumed on a different core, even in a different chip package on the same HyperWall server.

Later we will also introduce a per-VM key, the PK_{vm}, which is generated inside each VM and should be unique to each VM (e.g. by using a hardware random number generator instruction, `trng` in Table 3, to ensure the hypervisor cannot orchestrate two VMs generating the same key pair by controlling inputs which affect collected randomness). This key should be stored inside the memory region that is specified as private and off-limits both to the hypervisor and to DMA.

3.2 Detailed Operation

We explain the reason for each of HyperWall's new hardware features as we describe its detailed operation at server boot up, VM initialization, VM interruption, VM runtime, and VM termination. For reference, the hardware modifications are shown in Figure 2, while Tables 2 and 3 give details about new registers and instructions. The architectural features highlighted in Figure 2 will be summarized in Section 3.4.

The HyperWall architecture is designed to incur most overhead on starting a VM and to minimize the overhead as the VMs run. Regular VM operation incurs no new overhead. There is overhead only when the hypervisor interrupts the VM, updates the VM's memory mapping or itself makes a memory access. For this purpose, we interpose on TLB updates to check the protections when a new memory address is accessed. Consequently, a load or store from the hypervisor only incurs new overhead if an address translation entry in the TLB is not found.

3.2.1 Initializing HyperWall Components

When the system boots up, the CIP logic queries the memory controller to determine the start and end address of the protected memory and issues commands (e.g. injects store instructions) to write all 0s to the memory range. This memory becomes hardware-only accessible. As initially there are no guest VMs running, this invalidates all the entries in the CIP tables since no protection is needed. In addition, the CIP logic generates encryption and hashing keys (K_{enc} and K_{hash} in Figure 1).

Once the HyperWall components are initialized, the regular boot sequence continues. This includes the BIOS zeroing out the memory which was not reserved by hardware (i.e. all memory visible to software). There is no special boot up procedure for the hypervisor.

3.2.2 Starting a VM

We assume that the initial kernel image and data sent to the IaaS provider along with the requested protections are not confidential and are visible in plaintext to the hypervisor. This allows the hypervisor to inspect the protections to make sure they match what the customer requested. The customer also needs to provide a nonce (NC, to prevent replay attacks) and the requested protections. A new VM id (VID) is issued to identify this VM.

As part of the request, the VM will be assigned some devices (e.g. network card or disk). The guest OS will access these devices via memory pages shared with the hypervisor (if the device is emulated in software) or DMA (if hardware device is used). Such pages can simply be specified as allowing hypervisor and DMA access. If more details are available about the virtualization environment (emulated devices vs. dedicated hardware devices), the customer can provide a more detailed protection request (e.g., allow only hypervisor or only DMA access for the shared pages). We do not, however, specify whether physical devices have to be dedicated to VMs or emulated by the hypervisor. The VM simply expects a device and uses shared memory pages to access it.

Enabling the protections at VM start time requires a number of steps. First, the guest OS kernel image has a certain memory layout for its physical memory. The requested protections specify which

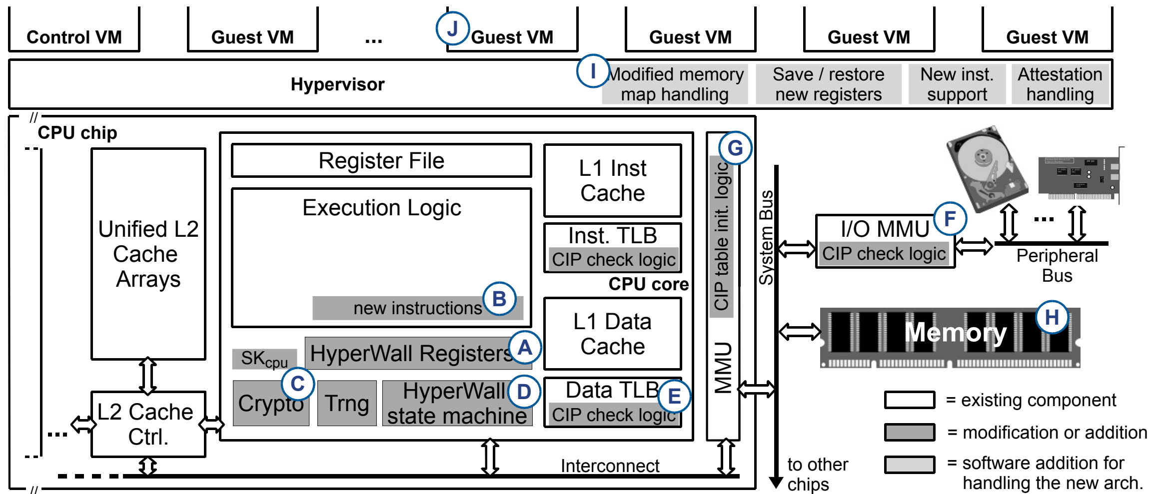

Figure 2. New hardware additions, and hypervisor updates, needed to support the HyperWall architecture.

Table 2. New HyperWall registers.

Register	Description (size)
SH	state keyed hash (32 bytes)
NC	customer's request nonce (8 bytes)
SS	initial state signature (128 bytes)
VID	VM id (8 bytes)
RPP	requested protections pointer (8 bytes)
IPH	initial protections hash (32 bytes)
IMH	initial memory hash (32 bytes)
VCNT	violation count (8 bytes)
VMAD	violation machine address (8 bytes)
VSIG	VCNT & VGPA signature (128 bytes)

Table 3. New instructions for HyperWall.

Instruction	Description
`sign_bytes`	Use CPU's private key to sign specified data.
`trng`	Access true random number generator.
`vmlaunch`	Signal hardware that a new VM is launched.
`vmresume`	Resume a VM.
`vmterminate`	Signal hardware to terminate the VM.

physical pages need to be protected. We call this specification the pre-CIP, it is provided by the customer to the cloud provider.

Second, when the VM is created, the hypervisor also sets up a mapping from the guest physical memory to the machine memory in the P2M page tables. By specifying protections for the physical memory and using the hypervisor's assigned memory mapping to obtain the corresponding machine pages requiring protections, we allow the hypervisor to freely manage physical to machine memory mapping.

Third, to enable the protections for a VM, the HyperWall architecture intercepts the `vmlaunch` instruction and updates the CIP tables with the new VM's protections. When the `vmlaunch` is intercepted, the CIP logic traverses the VM's memory mapping (the P2M tables). For each page referenced in the P2M tables, the hardware reads the pre-CIP data to check if this page requires protection. Initially all pages are marked as not assigned in the CIP tables. When the CIP logic detects that an un-assigned page is being assigned to a VM, the page's state transitions into one of the assigned states (hypervisor and DMA access allowed, hypervisor access denied, DMA access denied, or hypervisor and DMA access denied – see Figure 4), depending on the pre-CIP data. If a page being assigned does not have a corresponding entry in the pre-CIP data, it is automatically set to the state of "assigned with hypervisor and DMA access denied." If a page is already assigned in the CIP tables, it can not be assigned to more VMs and any action on that page, except freeing the page, will cause an error. If error is encountered at any time, the hardware rolls back any of the updates it did to the protections for the requested VM (i.e. the CIP tables do not hold any information about the requested VM), the VM is not started, and the `vmlaunch` instruction returns an error. The failed `vmlaunch` instruction will be detected by the hypervisor so it can attempt to remedy the problem (e.g. fix the memory mapping).

Fourth, if there has been no error during initialization, the hardware generates the hash of the initial protections and the hash of the initial memory contents (storing these in the new IPH and IMH registers in Table 2), and updates the attestation measurement signature (SS register). This SS value is a signature over all the HyperWall registers when the VM is created and uses the SK_{cpu} key:

$$SS = SIGN_{SK_{cpu}}(NC||VID||RPP||IPH||IMH \\ ||VCNT||VMAD) \quad (1)$$

The NC is a nonce specified by the customer each time the customer requests the HyperWall system to create a new VM or do an attestation. The nonce is part of signatures and hashes and ensures freshness. The VID is an identification number used to uniquely identify each VM and is set when the VM is first created. The RPP is a pointer to the memory location where the customer's original requested protections (the pre-CIP) are stored for future reference. The VCNT and VMAD are used for trust evidence, described later in Section 3.3, and are set to 0 when the VM is created.

After the VM is launched the hypervisor will send this SS signature (along with the values of other HyperWall registers associated with the VM, see Table 2) back to the customer to attest the VM that was started. The nonce, NC, and initial protection and memory hashes, IPH and IMH, can then be verified by the customer.

3.2.3 Runtime Protections Enforcement

Whenever the hypervisor runs and attempts to access some memory there will be a TLB miss on a first access to a memory page due to the lack of a translation entry in the TLB. Before the TLB

is updated, the HyperWall mechanisms traverse the CIP tables to check the access rights for the machine memory page. If no restrictions are present, a TLB entry is added as usual and there is no overhead on subsequent accesses. If there is a restriction, the TLB entry is not added and a trap to the hypervisor occurs, to signal the hypervisor that an illegal memory access was performed. The hardware will also record this violation (VCNT and VMAD in Table 2), and sign the violation record (VSIG in Table 2). The hypervisor already knows what the protected regions are so the trap conveys no new information, but allows it to make forward progress. The TLB entries need to be flushed whenever the CIP tables are updated to remove stale mappings. This can be performed via a TLB shoot-down to the remaining cores from the processor which updates the CIP tables. TLB shoot-down clears the entries in the TLBs, essentially flushing the old data held there.

A malicious hypervisor could configure a physical device to access the VM's memory and then read out that data from the device itself. To counter this in the HyperWall architecture, the I/O MMU is also modified so that the CIP tables are consulted on DMA accesses.

3.2.4 Interrupting and Resuming a VM

When the VM is interrupted and before the hypervisor begins to run, the VM's state needs to be protected: the general purpose registers are encrypted by hardware and the hash of the registers and the new HyperWall state is generated and stored in the SH register (see Table 2). This is a keyed-hash (using K_{hash} key) generated by HyperWall hardware each time the VM's execution is interrupted, to help preserve the VM's integrity.

The K_{hash} key is stored in memory (c.f. Figure 1) and thus accessible to HyperWall hardware on each core so a VM interrupted on one core can be resumed on any other core. Equation 2 describes how SH is generated.

$$SH = HASH_{Khash}(NC||SS||VID||RPP||IPH||IMH \\ ||VCNT||VMAD||VSIG||P2Mpointer \\ ||PC||E_{Kenc}(GPregisters)) \quad (2)$$

Then, the hypervisor can read out the registers and save them for when the VM is to be resumed.

When the hypervisor wants to schedule the paused VM again, it loads the saved state values back into the registers and calls `vmresume` to resume the VM. The CIP logic intercepts the instruction call and performs the inverse of the tasks performed when the VM was interrupted, i.e. it reads the saved hash of the VM state, verifies the hash and decrypts the registers before resuming the VM. The P2M pointer is included in the measurement so the hypervisor cannot restart the VM with a different memory mapping. A VM paused on one core can be resumed on any other core.

For hypercalls, whereby a guest OS requests some service from the hypervisor, the general purpose registers can not be encrypted (nor included in the SH hash). The guest OS uses the registers to pass some arguments and may return results also in the registers so the values will be different when the VM makes the call and when it is resumed. The HyperWall hardware can detect the case of a hypercall by comparing the trap number with the one associated with hypercalls (e.g. 0x100 on SPARC) and conditionally not do the encryption. For device emulation, we assume that all data copied between the hypervisor (or a driver VM emulating a device) will be copied in the shared memory. When the VM is interrupted to perform the device emulation, its register state will not need to be updated by the hypervisor. Consequently hypercalls are the only case when our hardware must not encrypt general purpose registers.

3.2.5 Runtime VM Management

During the lifetime of a VM, the hypervisor needs to perform a number of tasks. Most prominent are: scheduling VMs on different cores, memory management and VM migration.

Scheduling VMs requires the hypervisor to interrupt a VM, save the state and at some later time restore the state and resume the VM (on the same or different core). We have already discussed how the VM is protected when it is interrupted. These protections ensure that the sensitive code and data are protected when the VM is paused and resumed.

Memory management tasks involve the hypervisor updating the physical to machine memory mapping as VMs are created, the memory allocation is changed or VMs are terminated. The requested protections (the pre-CIP) are saved when the VM is created and made unreadable by the hypervisor or DMA. The pre-CIP data is consulted again whenever the memory mapping is updated. Such an update is triggered when the hypervisor writes the P2M pointer register with a new value. First, our hardware mechanisms deny access to hypervisor or DMA (set the pages as private) for the pages which occupy the *next P2M* tables (see Figure 1). Second, they traverse the new memory mapping in the *next P2M* tables and, using pre-CIP data, they update the CIP tables with the hypervisor-deny or DMA-deny protections for the machine pages corresponding to the physical pages in the new mapping. Just as when creating a mapping for the first time, if a page being assigned to the VM (as specified in the next P2M) is already assigned to some other VM in the CIP tables, the CIP table will not be updated and the memory mapping update will terminate with error. If new memory pages have been successfully assigned to the VM with no errors, our hardware mechanisms traverse the previous P2M mapping to remove from the CIP tables the memory pages not used anymore by the VM so they can be reclaimed. If a removed page was protected from hypervisor or DMA accesses, it is zeroed out by hardware, so no information leaks out, before clearing the protection bits. Finally, the machine memory pages where the old memory mapping was stored (P2M tables in Figure 1) is released back to the hypervisor, i.e., the protection bits in the CIP tables are cleared.

Through the above memory management, the HyperWall allows for the common technique of memory ballooning [43]. Memory ballooning is a technique for reserving some pages of a VM's allocated memory that can then be dynamically given to this or another VM when needed. Ballooning depends on the hypervisor's ability to dynamically change the memory mapping of a VM as it runs – as described above. To prevent information leakage, any (protected) memory reclaimed by the hypervisor will first be zeroed out by HyperWall hardware. Zeroing does not cause problems since before reclaiming memory in a VM, its balloon size is increased causing the OS to save data in any pages that will be reclaimed, so it is safe for hardware to zero them out.

To allow migration between different servers, we propose the use of guest-assisted migration [22]. For OSes which do not support such self-migration, we propose the use of a software shim to perform the migration tasks. Guest-assisted migration can be used for the majority, if not all, of the scenarios and the extra complexity of adding migration support in HyperWall hardware does not seem justified.

3.2.6 Communicating with Customer

Once the attestation measurements are verified, the customer knows that the specified VM and protections were started, and the VM's memory is isolated from the hypervisor and DMA. External communication, however, still needs to be protected. To secure communications between the customer and his leased guest VM, a cryptographically-secured channel is set up using a modified SSL

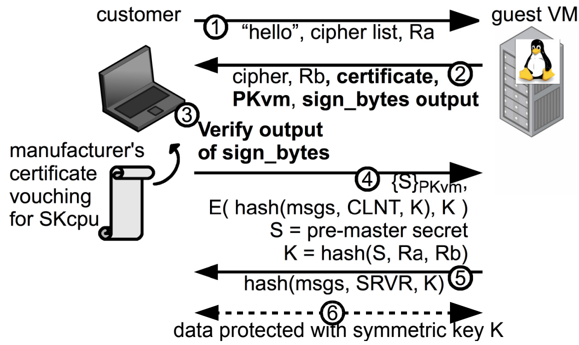

Figure 3. Establishing secure communication with the guest VM.

protocol. Note that this secure channel is between the VM (not the server) and the customer's machine. Hence, we generate a new public-private key pair for this VM, after its memory is securely isolated from the untrusted hypervisor and DMA.

The protocol is shown in Figure 3 with our modifications shown in bold. With our modifications, the servers's certificate and public key, used in standard SSL exchange, are replaced by PK_{vm} and the output of our new `sign_bytes` instruction. Initially, the VM has no public key which could be used in standard SSL exchange (recall that initial VM contents are sent in plain text so any key would be visible to the hypervisor and thus not secure). However, once the VM is started, its memory is off limits to the hypervisor, so it can generate a public-private key pair, e.g. using the OpenSSL library. Once the PK_{vm} and its associated private key is generated, the VM can use the `sign_bytes` instruction to have the hardware sign the PK_{vm} key. The PK_{vm} and the signature are sent back in our modified SSL exchange. By using the hardware manufacturer's certificate, the signature can be verified to have come from a HyperWall server. The CPU privilege level (priv_level), VM id (VID), and the nonce (NC) are included in the signature to verify that indeed the VM executed the instruction:

$$\begin{aligned} &sign_bytes(PK_{vm}) \\ &= SIGN_{SK_{cpu}}(\, PK_{vm} \,||\, priv_level \,||\, VID \,||\, NC \,) \end{aligned} \tag{3}$$

3.2.7 VM's Persistent Storage

If confidential data or code is sent when a VM is created, there would be a need to have hardware on the HyperWall-enabled server for doing complex mutual authentication between the customer and the hardware. Instead, the initial kernel image and protections are visible in plaintext to the hypervisor. The customer sends a vanilla VM image (with cryptographic libraries such as OpenSSL). The CIP protections are then locked in for this VM. Once a secure channel between the customer and the guest VM is established, the software inside the protected VM can perform the mutual authentication with the customer. Also, proprietary or confidential data or code can then be sent to the VM. An encrypted disk image (or remote storage) can be used to store the code or data, and the SSL connection can be used to give the VM keys to decrypt the encrypted storage. Because the communication is protected and the VM's memory is isolated from the hypervisor, the hypervisor will not be able to access the keys to decrypt the storage and see its plaintext contents.

3.2.8 VM Termination

When the VM is terminated, its memory is reclaimed by issuing our `vmterminate` instruction (see Table 3). The CIP logic intercepts the call to this instruction and traverses the P2M page table 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. If a hypervisor fails to issue the `vmterminate` instruction, it will remain locked out of the memory that has been assigned to the VM.

3.3 Trust Evidence

We propose trust evidence that can be used to provide a customer with information he or she can use to verify the target system, the VM that was started and the protections installed in hardware. Trust evidence is composed of two parts: attestation of the initial VM and the requested protections and attestation of the enforcement of the protection.

3.3.1 Attesting Initial VM and Protections

When the VM is created, the HyperWall hardware generates the initial state signature, SS, as defined by Equation 1. This locks in the initial VM state, and the hypervisor sends the values of the registers (including IMH and IPH) as an attestation to the customer. This attests that the requested VM started with the requested protections. Given the hardware manufacturer's certificate, the customer is able to verify the signature and compare the register state to the expected values. The measurement includes the initial memory hash to check that the provided VM image was used. It also includes the initial protections hash to check that the hypervisor did not alter the requested protections as it inspected them. If all the values are as expected, then a correct VM was started and the requested memory regions are protected.

3.3.2 Attesting Runtime Protection Enforcement

The customer may also want to verify that the protections are being enforced as the VM runs. For this purpose, HyperWall keeps track of the count of memory access violations (VCNT in Table 2) and the address where the last violation occurred (VMAD in Table 2).

To obtain the runtime attestation, the customer sends the VM id to identify the VM, a nonce for freshness and a challenge. The challenge specifies the physical address that the hypervisor or DMA should try to access so enforcement of protections can be verified. Next, the hypervisor performs the challenge (e.g., attempts to access the machine page which is mapped to the VM's physical page). If the access is in violation of protections, the hardware increments the violation count (VCNT) and saves the address that caused the violation (VMAD). At the same time, the VSIG signature is generated by the hardware using the SK_{cpu} key:

$$VSIG = SIGN_{SKcpu}(NC||VCNT||VMAD) \tag{4}$$

If there was no violation, the violation count and address are not updated. Violation count, the address and signature form the trust evidence which is read by the hypervisor and sent to the customer, as the runtime attestation response.

This is one possible implementation of runtime attestation. By comparing the challenge and the response, the customer can check that the protections are enforced for the address specified in the challenge. Given this base mechanism, a customer can design different schemes for verifying the whole VM's protections: randomly spot check the protections or fully scan all protected regions, for example.

We point out that our trust evidence attests to the protections of the VM requested by the customer; it is not a measurement of the current state of the OS or hypervisor or the entire software stack (as in the Trusted Platform Module's attestation [11]). We provide evidence that the protections of the VM used in the cloud are indeed being enforced.

3.4 Architecture Summary

Figure 2 shows the hardware modifications required to implement HyperWall and highlights our re-use of existing commodity microprocessor mechanisms.

New HyperWall registers (Ⓐ in Figure 2) are introduced to store protection information about the currently executing VM. Table 2 lists these new registers for each processor core. These registers are updated by hardware and can be saved and restored by the hypervisor when it switches VMs. Two new instructions, `sign_bytes` and `trng` are added (Ⓑ), while three existing ones are modified (see Table 3 and description in previous section). A hardware random number generator block is added to support the `trng` instruction.

A cryptographic engine (Ⓒ) is needed for performing encryption, decryption, hashing and signing (for which the SK_{cpu} key will be used). The bulk of the HyperWall logic is in a state machine (Ⓓ) which is responsible for updating the CIP tables when a VM is created or terminated. Moreover, 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 physical to machine (P2M) page mapping.

The TLB update logic (Ⓔ) is expanded to consult the CIP tables before inserting an address translation into the TLBs. To improve performance, the access checks are done when the address translation is performed. The address translation is cached in the TLB if there is no violation, and the CIP table check can be avoided in the future. To prevent stale mappings, the TLBs need to be flushed whenever the CIP tables are updated. Similarly to the address translation in the main processor, the I/O MMU (Ⓕ) needs to have extra logic to consult the CIP tables. The MMU (Ⓖ) is updated with 1 extra register to mark where the protected memory region starts, and CIP logic is added to walk the CIP tables on a hypervisor access. We re-use a portion of DRAM (Ⓗ) to store the CIP tables.

Although we introduce our modification only in the microprocessor and consider the hypervisor untrusted; the hypervisor (Ⓘ), as the entity in charge of the platform, will need to interact with our new architecture. In addition to the usual tasks, the hypervisor needs to perform a few additional functions. 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 handle the `vmlaunch`, `vmresume`, and `vmterminate` instructions to start, resume, or terminate a VM respectively. It needs to read attestation measurements from hardware registers. It needs to use a modified procedure for updating the memory mapping during VM runtime (i.e. swap old and new P2M tables rather than modify individual entries in old P2M mapping).

The guest OSes (Ⓙ) do not need explicit modification. The only change is the possible inclusion of a library which makes use of the new `trng` instruction. However, commodity Linux distributions already are able to use hardware random number generators found in computer chipsets (e.g. in Intel's 82802 Firmware Hub [6]) so no modification in such systems is needed. The software memory manager found in the guest OSes should never map the sensitive code or data such that it is located in the pages accessible to the hypervisor. If only pages which are used as buffers between VM and (emulated or read) devices are accessible to the hypervisor, the manager will never map any data or code to these pages as they are set to be used by the device drivers for those devices. If only a small subset of pages is protected, however, then guest OS modification may be needed so that sensitive data is never mapped into unprotected pages.

3.4.1 New Register State

Table 2 shows per processor core registers which are used to manage VM protections. These have all been described in Section 3.2 (Detailed Operation).

3.4.2 New Instructions

New or modified instructions are listed in Table 3. We add unprivileged `sign_bytes` and `trng` instructions. The `sign_bytes` is used to sign the data at the requested address. In addition to the data, the signature includes: current privilege level and the VID and NC register values. All the data are concatenated and then signed with the SK_{cpu} key. The privilege level specifies whether user software, OS, or hypervisor invoked the instruction. For user or OS invocation, the VID specifies the VM that invoked it (value is 0 for hypervisor invocation). The NC is used to ensure freshness, e.g. the customer requests the VM to sign some data and gives it the nonce to use. The `sign_bytes` is especially used when the PK_{vm} is generated by a VM so that the customer can verify that PK_{vm} indeed came from their VM. The `trng` uses a hardware random number generator and returns random bytes to the guest.

We make use of hyper-privileged instructions to start, resume and terminate a VM. Some of these are already present in many modern architectures (e.g. Intel's x86 with virtualization extensions). The `vmlaunch` signals the hardware that a new VM is launched and causes HyperWall protections for that VM to be enabled. The `vmterminate` instruction causes all of the VM's protected memory regions to be zeroed out by the hardware and their entries in the CIP tables cleared so the memory can be reclaimed. When a VM is interrupted the new HyperWall registers describing that VM are saved; the `vmresume` instruction signals that the VM is ready to be resumed, causing the hardware to check the integrity of the restored HyperWall registers.

3.4.3 Cryptographic Routines

We have included the cryptographic routines as an extra hardware unit in the microprocessor. This can be implemented as dedicated circuits (as in our simulations), microcode or software routines in on-chip ROM.

4. Simulation and Evaluation

4.1 Baseline Simulation

We have implemented HyperWall in an OpenSPARC T1 simulator. OpenSPARC is an open-source 64-bit, multi-threaded, multicore microprocessor from Sun Microsystems (now Oracle), modeled after their UltraSPARC commercial microprocessor. The whole microprocessor code (which can be synthesized to ASIC or FPGA), the code for the hypervisor, as well as a set of simulators (Legion and SAM) are available. We use the Legion simulator which is able to simulate multiple cores, boot the hypervisor, and run multiple VMs. HDL (hardware description language) code could be modified if the design is to be synthesized into a real chip or to obtain power and area measurements.

We have modified the Legion simulator with the new HyperWall functionality. We utilize the PolarSSL [8] library to simulate

Table 4. Simulator Configuration

Component	Parameter	Value
System Total	ITLB, DTLB	16 entries each
	DRAM	32 GB
	CPU cores	2
Each guest VM	CPU cores	1
	Memory, Disk	256MB, 2GB
	Guest OS	Ubuntu Linux 7.10

the cryptographic operations. We have configured Legion to simulate 2 cores and run 2 VMs (one per core), see Table 4. Each guest VM has 10 memory regions spanning 4GB of memory (e.g. RAM, memory-mapped disk image, emulated console, etc.) and the protections are consequently specified for a 4GB region in our test pre-CIP data.

For the performance evaluation we run the hypervisor, Ubuntu OS and selected SPEC 2006 benchmarks on the simulator to obtain instruction counts for various operations and events. We assume conservatively 5 cycles per instruction (CPI=5) for memory load and store instructions, CPI=1 for register-based functional instructions, and CPI=2 as average for all instructions to estimate cycle counts. We simulate: SHA-256 for hash generation with 0.13 cycles/byte [24], AES for encryption with 1.38 cycles/byte [28] and 1024-bit RSA digital signature which takes 283K cycles [36].

4.2 CIP Tables Implementation

There are many possible implementations for the CIP tables. We chose a flat table with one entry per machine page. This has simple addressing logic: the high-order bits of a machine page's address are used as an index into the CIP table.

Each CIP table entry is 4 bits. While 3 bits are enough for the 5 states currently defined per page as described in Section 3.1, we opted for 4 bits, a power of 2, see Figure 4. The extra (reserved) encodings can be used for future functionality. The first two bits of each entry are used to detect if a page has been assigned to a VM or not: 00 is unassigned, 11 is an assigned page that is private to a VM. While 01 and 10 are reserved, one future use for these encodings which we are investigating is for pages shared among VMs.

An alternative design is to use a dedicated hardware memory structure for storing the CIP tables. This, however, would impact scalability (having a dedicated hardware structure pre-determines the maximum amount of memory which can be protected, preventing installation of more memory at a later time) and increase hardware costs (having to add more hardware components). By using a hardware-only accessible portion of DRAM, we reduce the costs by using existing DRAM and provide scalability. At boot up time, the memory size is discovered and proportionally a part of DRAM is made hardware-only accessible and the tables are stored there. This allows the system to reserve exactly the size of memory needed for the tables; there is never a situation where some subset of memory cannot be protected due to insufficient hardware CIP table entries, nor are resources wasted by provisioning for the worst case. Moreover, the lost DRAM storage space is negligible (∼0.012% in our evaluation).

Another alternative design for the CIP tables that we explored is a page-table-like structure. This has the advantage that existing page-table walking hardware could be reused. However, more than

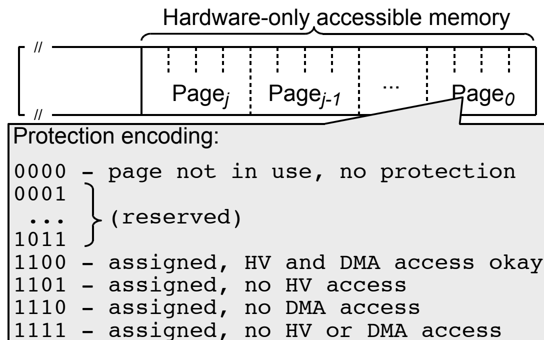

Figure 4. One design of CIP tables which uses 4 protection bits per page, all stored in hardware-only accessible memory.

Table 5. Worst-case Overhead of Accessing CIP Tables

Operation	Original System *total inst.*	Extra Memory Accesses *total (change)*
Startup (hypervisor boot)	58 M	
Init. CIP tables		524 K (0.9%)
Check CIP		21 M (36%)
VM Launch (guest OS boot)	2686 M	
Set CIP entries		2 M (<0.1%)
VM interrupt due to:		
Scheduling	∼76	8 (10%)
TLB miss	∼35	5 (14%)
VM terminate	- - -	
Clear CIP entries		1 M

one memory access is needed for a CIP table lookup (e.g., 2 memory accesses for a 2-level page-table structure). Also, CIP tables are located in fixed memory locations, and do not need the flexibility of locating them in arbitrary memory locations like conventional page tables. Hence, we chose the flat table structure for our CIP table implementation.

The size of the CIP tables depends on the amount of installed DRAM. For our implementation, we have chosen to simulate a system with 32 GB of addressable memory, which requires 4MB of storage (32GB $\rightarrow 2^{23}$ 4KB pages $*$ 4 bits per page $\rightarrow$ 4MB).

4.3 Storage Overhead

One benefit of storing CIP tables in DRAM is that the reserved amount of memory depends on the size of DRAM. Assuming 4KB pages and using 4 bits per page to describe the protections, 0.012% (i.e. 4MB out of the 32GB) of DRAM is made accessible only to the hardware and not usable by the rest of the system.

The new HyperWall registers in each microprocessor core sum to a total of only 392 bytes. There is also need for an extra register in the MMU to hold the start address of the hardware-only accessible memory region.

4.4 CIP Table Access Overhead

The worst case overhead of initializing the 4MB of CIP tables is ∼524K 64-bit accesses, which initialize the table entries by writing all 0s. This overhead can be greatly reduced if there is support for zeroing out whole blocks of memory at a time. Once the CIP tables are initialized and the system begins to run, they are consulted (in the worst case) on each hypervisor or DMA access. In this worst case, each memory access causes 1 extra access to the CIP tables to read the protections. In the average case, this will be much less as once an address translation is placed in the TLB, the CIP lookup does not need to be performed. Table 5 shows the worst case overheads for the simulated system.

During hypervisor boot up there are ∼58M instructions executed, of these 21M are memory instructions. Each memory instruction will cause in the worst case one extra access to read the CIP tables, hence the overhead of 21M extra memory accesses. During Ubuntu guest OS boot up, we observed ∼2686M instructions being executed, of these 2M were hypervisor memory accesses. Again, each hypervisor access in the worst case causes one extra access due to reading CIP tables, hence the overhead of 2M extra memory accesses. During regular OS execution, the OS often is interrupted by the hypervisor. For example, when a periodic interrupt to allow the hypervisor to perform scheduling duties happens, the hypervisor executes ∼76 instructions, of which 8 are memory accesses causing overhead of 8 extra memory accesses (due to reading the CIP tables). During VM termination, the CIP table entries need to be updated. In our simulation, the VMs are

Table 6. Approximate Runtime Cycle Overheads

Operation	Extra Overhead	
	Hardware (cycles)	Hypervisor (cycles)
VM Launch (5372 M cycles)		
set CIP entry	10 (per page)	
generate IMH	34900 K	
generate IPH	140 K	
generate SS	280 K	
read SS		4
VM Interrupt		
encrypt GPR	353	
generate SH	84	
save HyperWall regs.		21
VM Resume		
restore HyperWall regs.		21
verify SH	84	
decrypt GPR	353	
Runtime Attestation		
write NC		1
generate VSIG	280 K	
read VSIG		4
VM Terminate		
Zero VM memory	512 (per page)	
Clear CIP entries	10 (per page)	

assigned 4GB memory regions (1M pages) which requires 1M CIP entries to be updated.

4.5 Runtime Overhead

During runtime, a HyperWall system also experiences overhead due to cryptographic operations and updates of HyperWall registers. In addition, the hypervisor needs to perform new operations to save and restore the HyperWall registers; issue instructions to launch, interrupt, resume and terminate the VM; handle the attestation operations and handle the modified memory mapping update mechanisms. Recall, however, that the CIP tables are only consulted when the hypervisor is running and that there is no extra overhead for memory accesses from the guest OS or applications. Table 6 summarizes the different overheads and breaks them up into overheads incurred by hardware or by the hypervisor, while in the text below we highlight some important points.

Upon VM launch, most of the overhead is in updating the memory protection. We report overhead per protected page, which needs to be multiplied by the number of protected pages and will vary for each system. In our simulation setup, each VM occupies a 4GB space (memory, memory-mapped disk and memory-mapped devices) which is 1M pages. There will also be extra work in the cloud infrastructure to handle the protection specification and pass it to the hypervisor; this overhead is not reported here.

For the hash generation we emulate a hardware implementation of SHA-256 with 0.13 cycles/byte [30]. For our configuration (c.f. Table 4), hashing 256MB of memory and the pre-CIP tables takes about 35M cycles. One signature will take on the order of 280K cycles if a recent speed-optimized 1024-bit RSA design is emulated [36].

When a VM is interrupted, the overhead of encrypting the general purpose registers (GPRs) will be incurred on all VM interruptions, except due to a hypercall (when the registers are not encrypted, so information can be passed to the hypervisor). Similarly, during VM resume the decryption overhead will not be incurred if the VM is returning from a hypercall. The encryption of the 32 64-bit GPRs using a 1.38 cycles/byte AES implementation [28] takes 353 cycles. Generation of the signed hash, using a 0.13 cycles/byte [30] implementation, of the GPRs (32 64-bit registers) and our new state (392 bytes) requires hashing 648 bytes for a total of 84 cycles. The reverse of these operations is performed to verify the state when the VM is resumed.

For runtime attestation, there will also be extra work to deliver the attestation result to the customer. This is a complex process involving many software components, and the overhead is not reported. However, much similar overhead is incurred today, for example when the status of a VM executing in the cloud is reported to the customer, so the overall percentage change should be small.

When a VM is terminated, the protected memory regions need to be zeroed out by hardware and the CIP table entries removed. We report numbers per page-sized (4KB) region. Because now the hardware zeroes out the memory, the associated functionality can be disabled in the hypervisor, so the extra overhead will be partially compensated.

4.6 Impact on Memory Hierarchy Performance

Because there are extra accesses to the CIP tables, they will cause the cache hit rate to decrease. The cache hit rate changes can be used as a proxy for estimating performance impact.

To model the impact of the extra CIP table accesses on the system performance, we extended the OpenSPARC Legion simulator with a simulation of cache hierarchy. We simulate split L1 instruction and data caches, 4-way, 32KB each (there is one pair of caches per core); and a unified 12-way 3MB L2 cache. The parameters are set to match the OpenSPARC architecture. These are inclusive, write-back caches with a simple round robin replacement policy. We simulate the HyperWall state machine using the data cache to cache the CIP table entries, and hence it impacts software performance.

Our test runs included: booting the hypervisor, booting the whole guest OS, and running selected SPEC 2006 benchmarks. The cache hit rate changes were within 1 % in all cases. This suggests that HyperWall's impact should be minimal, but a more detailed cache and memory simulation is needed to get more exact estimates; this is part of our future work on optimizing a HyperWall implementation.

5. Security Analysis

We now evaluate the security of the HyperWall system. Keep in mind the goal of HyperWall is to protect the confidentiality and integrity of a VM's memory.

5.1 Protecting Confidentiality and Integrity

In any of today's virtualization solutions, the hypervisor is able to access all of a VM's memory or disk images and violate confidentiality and integrity of the code and data stored there. In HyperWall, the hypervisor has access to the initial guest VM image and the requested protections sent by the customer. Any of this data can be modified before the hardware is initialized and protections locked in. This attack is averted by the use of hardware-based attestation and sending the signed measurements when the VM is started. Moreover, during VM runtime, the customer can request trust evidence and verify that it came from a genuine HyperWall system (due to the hardware signature) and that the protections are still correctly specified for hardware enforcement.

Also, when the VM is running, the hypervisor could intercept communications between the customer and the VM. Initially, no confidential data is sent, so this does not gain anything for the attacker. Once the VM is started, customer to VM communication is protected with the modified SSL protocol. The hypervisor, which has no access to the decryption key, cannot break the SSL secure channel and extract any data sent in the SSL tunnel after the VM is running. The customer can also use this secure SSL tunnel to send the key for the VM to access and decrypt its encrypted storage where it can store proprietary code and confidential data.

Because the VM contains memory regions off-limits to the hypervisor or DMA, it can generate keys and keep them secure from a compromised hypervisor or DMA. Such keys should be used for hashing and encryption to protect the integrity and confidentiality of swap space that the guest VM may use, for example. In HyperWall, all disk storage, except for the initial VM image, should be encrypted and hashed.

5.2 Availability Considerations

HyperWall implementation of hypervisor-secure virtualization does not provide availability protection for the guest VMs. The confidentiality and integrity of the protected code and data, however, is maintained even if an untrusted hypervisor performs a denial-of-service (DoS) attack on the VM by suspending or terminating it. The memory with the code and data is protected from hypervisor or DMA access even if the VM is suspended and, moreover, it is zeroed out by hardware when the VM is terminated so that no information leaks out.

One reason we felt it was reasonable to leave out availability protection is because the IaaS cloud provider must be able to stop a VM anyway. In an IaaS hosted infrastructure cloud computing scenario, customers lease the resources and pay for them. The IaaS cloud provider (via using hypervisor mechanisms) needs to be able to stop a VM if the customer is not paying for the resources. Otherwise the customers could execute a DoS attack on the cloud provider's resources, or get to use the resources for free.

5.3 Other Security Concerns

One potential concern is the hypervisor's lack of ability to fully inspect the memory contents for the guest VM, and thus perform security checking of the VM via introspection.. Allowing the hypervisor to access a VM's memory, however, is at odds with our assumption of an untrusted hypervisor. But, the infrastructure provider can still use network-based intrusion detection systems (IDS) to monitor the network for any unusual activity that may hint at a malicious VM. Also while still using the protections, a customer could use in-VM monitoring and install a security agent that monitors the state of the VM and reports back to the customer and the infrastructure provider.

Another potential concern is a set of cooperating malicious hypervisors which may start one copy of the VM on a genuine HyperWall machine and a second copy on a different server and forward the same data to both so that the second system leaks the data or code. However, after receiving the attestation measurement sent when the VM is started, the customer knows which machine the VM is executing on (due to the unique SK_{cpu}). The guest VM generates a public-private key pair and signs it using the new `sign_bytes` instruction to establish a secure connection with the customer. When the customer receives and verifies the signed PK_{vm} key, he or she knows that it came from the same server which properly attested. The copy of the VM on the non-HyperWall machine will not have the keys needed to decrypt the communication or decrypt the customer's encrypted data or code.

The ability of the customer to verify the SK_{cpu} opens up the possibility of infrastructure mapping attacks [37] as now there is an easy way to identify a particular server. While we do not protect against such attacks in the current work, ideas from direct anonymous attestation [17] could be used.

A different concern is that a malicious hypervisor could attempt to assign memory pages to a victim VM that have already been assigned to a hostile VM. If this was possible, the hypervisor access and DMA access checks would be bypassed as memory protections are not checked on VM access. This attack is not possible in HyperWall. When a VM is created, the CIP tables are updated to hold information about its pages. Even unprotected, but assigned, pages are marked in the CIP tables (recall the CIP table entry `1100` encoding in our implementation). Consequently, all pages assigned to a potentially hostile VM would be marked in the CIP tables. When a hypervisor attempts to assign the same page to the victim VM, the CIP logic detects that the page is already in use, CIP table update terminates with error and the victim VM is not started.

All the above analysis assumes correctness of the underlying hardware which implements our new mechanisms. Also, the protocols used to transfer the data back to the customer need to work correctly. Consequently, our ongoing work looks at verification of the various parts of the architecture as well as of the protocols. While full formal verification of security architectures is an open research issue, by verifying components and protocols we can significantly raise the confidence in the HyperWall architecture.

6. Related Work

We summarize some past work on aspects of our architecture: hardware support for isolation, attestation, using hypervisors to monitor VMs, hardening or minimizing hypervisors, secure processor architectures, and the homomorphic encryption approach. None of this surveyed work met our goals of hypervisor-secure virtualization with hardware support.

In the past, manufacturers such as IBM and Sun Microsystems (now Oracle) have solutions which are used to partition physical resources and enforce strong isolation between the VMs [4, 10]. They, however, include the management software in the TCB and do not protect against compromised or malicious management software which can snoop on or attack the VMs, as we do.

TPM has been deployed extensively in notebook PCs and can be used for attestation of the software stack. For example, work by Krautheim et al. [26] looks at extending the TPM to the cloud computing environment. Also, Flicker [35] uses the TPM along with a trusted hypervisor. While work which uses TPM can provide attestation by creating a measurement chain of bootloader, hypervisor, OSes, etc., we are able to provide attestation separately to each VM and without inclusion of the untrusted hypervisor in the measurements. Also, we introduced the runtime attestation of protection enforcement not present in TPM-based systems.

The majority of hypervisor-related work has been done on using the hypervisors to inspect or monitor guest VMs (introspection) and on hardening the hypervisors. Such work does not consider attacks from the hypervisor on the VMs. Introspection schemes use the hypervisor's higher privilege to secure the guest VMs. One of the first mentions of the idea was presented by Chen, et al. [19]. HookSafe [46] is a hypervisor-based system for detecting guest OS rootkits. Also, KvmSec [33], an extension to Linux's kernel virtual machine, adds the ability to check the integrity of the guest VMs. Such work on inspecting and monitoring guest VMs have a different goal than HyperWall. In HyperWall, we consider attacks from the hypervisor. Also, we aim to provide confidentiality and integrity to the VM's data and code from a potentially compromised hypervisor, rather than monitoring the VMs to discover any anomalies when they have already been compromised.

The hardening and minimizing of hypervisors has also been explored. For example, Li, et al., [31] propose protecting the hypervisor kernel from an untrusted management OS while HyperSafe [44] is a proposed prototype of a hypervisor which aims to protect a hypervisor's code and data from unauthorized modification. These and similar works modify the hypervisor's code to make it more resilient to attacks. They can be combined with HyperWall to provide defense in depth to guard against any bugs or misconfigurations in the hardened hypervisors.

Our earlier NoHype architecture [23] proposed removing the hypervisor altogether. However, NoHype still has a small trusted management software which can maliciously access a guest's

memory, if it is compromised. HyperWall also allows more functionality than NoHype as it allows a hypervisor layer to run and actively manage the VMs.

A number of research projects have looked at secure processor architectures and adding hardware support to protect software. These include XOM [32], AEGIS [39], SP [29] and Bastion [18]. In Bastion, the hypervisor is trusted, and the others do not consider hypervisors at all. Unlike HyperWall, these proposals do protect against hardware attacks, but these are much less likely in physically-secured data centers for cloud computing.

Outside of hardware solutions, researchers have been exploring fully homomorphic encryption [21]. Its main advantage is that computation can be performed on encrypted data without revealing what the data is. The disadvantage is the (currently) very low performance and that only the data is protected, not the code. We offer a solution which can be realized with small modifications to today's hardware and protects both code and data.

7. Conclusions and Future Work

We presented the HyperWall architecture which relies on new Confidentiality and Integrity Protection (CIP) tables to protect the guest VM's memory from a compromised or malicious hypervisor. HyperWall allows hypervisors to freely manage the memory, processor cores and other resources of the platform. Yet, once VMs are created, CIP logic protects the memory of the guest VMs from a potentially compromised hypervisor and from DMA. The HyperWall architecture is also able to provide trust evidence attestation to the customer showing that the VM was started with the correct protections, and the protections are being enforced during runtime, in spite of a possibly malicious hypervisor. HyperWall can be especially useful in a setting such as IaaS clouds to enable customers to have the confidentiality and integrity of their VM's data and code protected while enjoying the economic benefits of cloud computing. Our evaluation shows reasonably small overheads due to the new mechanisms. Thus, HyperWall strikes a good balance between security and performance.

Our ongoing and future work includes studying HyperWall architecture's interaction with performance counters and other state which may leak information about the guest VM, protecting the code and data when it is executed outside the microprocessor chip (e.g. on GPUs), and providing nested protections where both the whole VM and some applications inside it can be protected separately. We are also investigating verification of various parts of the architecture as well as of the protocols. These issues suggest that this paper may open up many interesting and important new research topics in the new research direction that we have called *hypervisor-secure virtualization*.

Acknowledgments

We would like to thank the anonymous reviewers for their suggestions which improved this paper. We also thank Yu-Yuan Chen for his help in looking up hypervisor vulnerabilities and their statistics. This work is supported in part by NSF grants CCF-0917134 and EEC-0540832.

References

[1] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/.

[2] AMD Virtualization (AMD-V) Technology. http://sites.amd.com/us/business/it-solutions/virtualization/Pages/amd-v.aspx.

[3] Vulnerability Summary for CVE-2007-4993. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2007-4993.

[4] Introduction to the New Mainframe: z/VM Basics, section 1.9.2 and 2.4.1. IBM Redbooks, http://www.redbooks.ibm.com/abstracts/sg247316.html.

[5] Intel Virtualization Technology, . http://www.intel.com/technology/itj/2006/v10i3/1-hardware/6-vt-x-vt-i-solutions.htm.

[6] Intel 82802AB/82802AC Firmware Hub (FWH) datasheet, November 2000, . http://download.intel.com/design/chipsets/datashts/29065804.pdf.

[7] National Vulnerability Database, CVE and CCE Statistics Query Page. http://web.nvd.nist.gov/view/vuln/statistics.

[8] PolarSSL, Small Cryptographic Library. http://www.polarssl.org.

[9] PCI SIG: PCI-SIG Single Root I/O Virtualization. http://www.pcisig.com/specifications/iov/single_root/.

[10] Oracle VM Server For SPARC. http://www.oracle.com/us/oraclevm-sparc-ds-073441.pdf.

[11] Trusted Computing Group. TCG TPM Specification. http://www.trustedcomputinggroup.org/.

[12] VMWare. http://www.vmware.com/.

[13] Intel Corporation: Intel Virtualization Technology for Directed I/O. http://download.intel.com/technology/itj/2006/v10i3/v10-i3-art02.pdf.

[14] Xen. http://www.xen.org.

[15] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proc. of the 17th ACM Conference on Computer and Communications Security*, CCS, pages 38–49, October 2010. doi: http://doi.acm.org/10.1145/1866307.1866313. URL http://doi.acm.org/10.1145/1866307.1866313.

[16] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The turtles project: design and implementation of nested virtualization. In *Proc. of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI, pages 1–6, October 2010. URL http://portal.acm.org/citation.cfm?id=1924943.1924973.

[17] E. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In *Proc. of the 11th ACM Conference on Computer and Communications Security*, CCS, pages 132–145, October 2004. ISBN 1-58113-961-6. doi: http://doi.acm.org/10.1145/1030083.1030103. URL http://doi.acm.org/10.1145/1030083.1030103.

[18] D. Champagne and R. Lee. Scalable architectural support for trusted software. In *16th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1 –12, January 2010. doi: 10.1109/HPCA.2010.5416657.

[19] P. M. Chen and B. D. Noble. When virtual is better than real. *Workshop on Hot Topics in Operating Systems*, May 2001. doi: http://doi.ieeecomputersociety.org/10.1109/HOTOS.2001.990073.

[20] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proc. of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 2–13, March 2008. ISBN 978-1-59593-958-6. doi: http://doi.acm.org/10.1145/1346281.1346284. URL http://doi.acm.org/10.1145/1346281.1346284.

[21] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proc. of the annual Symposium on Theory of Computing*, STOC, pages 169–178, May 2009. ISBN 978-1-60558-506-2. doi: http://doi.acm.org/10.1145/1536414.1536440. URL http://doi.acm.org/10.1145/1536414.1536440.

[22] J. G. Hansen and E. Jul. Self-migration of operating systems. In *Proc. of the 11th ACM SIGOPS European Workshop*, September 2004.

[23] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. NoHype: virtualized cloud infrastructure without the virtualization. In *Proc. of the 37th*

annual International Symposium on Computer Architecture, ISCA, pages 350–361, June 2010. ISBN 978-1-4503-0053-7.

[24] M. Kim, J. Ryou, and S. Jun. Efficient Hardware Architecture of SHA-256 Algorithm for Trusted Mobile Computing. In *Information Security and Cryptology*, volume 5487 of *Lecture Notes in Computer Science*, pages 240–252. 2009.

[25] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *Proc. of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP, pages 207–220, October 2009. ISBN 978-1-60558-752-3. doi: http://doi.acm.org/10.1145/1629575.1629596.

[26] F. Krautheim, D. Phatak, and A. Sherman. Introducing the Trusted Virtual Environment Module: A New Mechanism for Rooting Trust in Cloud Computing. In *Trust and Trustworthy Computing*, volume 6101 of *Lecture Notes in Computer Science*, pages 211–227. 2010.

[27] G. Kroah-Hartman, J. Corbet, and A. McPherson. Linux kernel development. A White Paper By The Linux Foundation, December 2010. www.linuxfoundation.org/publications/whowriteslinux.pdf.

[28] R. B. Lee and Y.-Y. Chen. Processor accelerator for AES. In *Proc. of the 8th IEEE Symposium on Application Specific Processors*, SASP, pages 16 –21, June 2010. doi: 10.1109/SASP.2010.5521153.

[29] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang. Architecture for Protecting Critical Secrets in Microprocessors. In *Proc. of the 32nd annual International Symposium on Computer Architecture*, ISCA, pages 2–13, June 2005.

[30] Y. Lee, H. Chan, and I. Verbauwhede. Iteration Bound Analysis and Throughput Optimum Architecture of SHA-256 (384,512) for Hardware Implementations. In S. Kim, M. Yung, and H.-W. Lee, editors, *Information Security Applications*, volume 4867 of *Lecture Notes in Computer Science*, pages 102–114. 2007.

[31] C. Li, A. Raghunathan, and N. K. Jha. Secure Virtual Machine Execution under an Untrusted Management OS. *Proc. of the IEEE International Conference on Cloud Computing*, pages 172–179, July 2010. doi: http://doi.ieeecomputersociety.org/10.1109/CLOUD.2010.29.

[32] 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. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, November 2000.

[33] F. Lombardi and R. Di Pietro. KvmSec: a security extension for Linux kernel virtual machines. In *Proc. of the 2009 ACM Symposium on Applied Computing*, SAC, pages 2029–2034, March 2009. ISBN 978-1-60558-166-8. doi: http://doi.acm.org/10.1145/1529282.1529733.

[34] V. Maraia. *The Build Master: Microsoft's Software Configuration Management Best Practices*. Addison-Wesley Professional, 2005.

[35] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: an execution infrastructure for TCB minimization. In *Proc. of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, EuroSys, pages 315–328, March 2008. ISBN 978-1-60558-013-5. doi: http://doi.acm.org/10.1145/1352592.1352625. URL http://doi.acm.org/10.1145/1352592.1352625.

[36] A. Miyamoto, N. Homma, T. Aoki, and A. Satoh. Systematic Design of RSA Processors Based on High-Radix Montgomery Multipliers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, (99):1–11, 2010. ISSN 1063-8210. doi: 10.1109/TVLSI.2010.2049037.

[37] 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. of the conference on Computer and Communications Security*, CCS, pages 199–212, Nov. 2009. ISBN 978-1-60558-894-0.

[38] J. Rutkowska. Subverting Vista Kernel for Fun and Profit. Symposium on Security for Asia Network (SyScan), July 2006.

[39] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *Proc. of the 17th annual International Conference on Supercomputing*, ICS, pages 160–171, June 2003. ISBN 1-58113-733-8. doi: http://doi.acm.org/10.1145/782814.782838. URL http://doi.acm.org/10.1145/782814.782838.

[40] J. Szefer and R. B. Lee. A Case for Hardware Protection of Guest VMs from Compromised Hypervisors in Cloud Computing. In *Proc. of the Second International Workshop on Security and Privacy in Cloud Computing*, SPCC, June 2011.

[41] A. Tereshkin and R. Wojtczuk. Introducing ring -3 rootkits. Black Hat USA, July 2009.

[42] A. Vasudevan, J. M. McCune, N. Qu, L. Van Doorn, and A. Perrig. Requirements for an integrity-protected hypervisor on the x86 hardware virtualized architecture. In *Proc. of the 3rd international conference on Trust and Trustworthy Computing*, TRUST, pages 141–165, June 2010. ISBN 3-642-13868-3, 978-3-642-13868-3. URL http://portal.acm.org/citation.cfm?id=1875652.1875663.

[43] C. A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36:181–194, December 2002. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/844128.844146. URL http://doi.acm.org/10.1145/844128.844146.

[44] Z. Wang and X. Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Proc. of the 2010 IEEE Symposium on Security and Privacy*, S&P, pages 380 –395, May 2010. doi: 10.1109/SP.2010.30.

[45] Z. Wang and R. B. Lee. A novel cache architecture with enhanced performance and security. In *Proc. of the annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, pages 83–93, Nov. 2008. ISBN 978-1-4244-2836-6.

[46] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *Proc. of the conference on Computer and Communications Security*, CCS, pages 545–554, Nov. 2009. ISBN 978-1-60558-894-0. doi: http://doi.acm.org/10.1145/1653662.1653728.

[47] R. Wojtczuk and J. Rutkowska. Attacking intel trusted execution technology. Black Hat DC, Feb. 2009.