Architectural Support for Secure Virtualization under a Vulnerable Hypervisor

Seongwook Jin, Jeongseob Ahn, Sanghoon Cha, and Jaehyuk Huh
Computer Science Department, KAIST
Daejeon, Korea
{swjin, jeongseob, shcha, and jhuh}@calab.kaist.ac.kr

ABSTRACT

Although cloud computing has emerged as a promising future computing model, security concerns due to malicious tenants have been deterring its fast adoption. In cloud computing, multiple tenants may share physical systems by using virtualization techniques. In such a virtualized system, a software hypervisor creates virtual machines (VMs) from the physical system, and provides each user with an isolated VM. However, the hypervisor, with a full control over hardware resources, can access the memory pages of guest VMs without any restriction. By compromising the hypervisor, a malicious user can access the memory contents of the VMs used by other users.

In this paper, we propose a hardware-based mechanism to protect the memory of guest VMs from unauthorized accesses, even with an untrusted hypervisor. With this mechanism, memory isolation is provided by the secure hardware, which is much less vulnerable than the software hypervisor. The proposed mechanism extends the current hardware support for memory virtualization with a small extra hardware cost. The hypervisor can still flexibly allocate physical memory pages to virtual machines for efficient resource management. However, the hypervisor can update nested page tables only through the secure hardware mechanism, which verifies each mapping change. Using the hardware-oriented mechanism in each system securing guest VMs under a vulnerable hypervisor, this paper also proposes a cloud system architecture, which supports the authenticated launch and migration of guest VMs.

Categories and Subject Descriptors

C.1.0 [**Processor Architectures**]: General; D.4.6 [**Operating Systems**]: Security and Protection

This research was partly supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2011-0026465), and by the IT R&D Program of MKE/ KEIT(2011-KI002090, Development of Technology Base for Trustworthy Computing)

General Terms

Security, Design

1. INTRODUCTION

Cloud computing has emerged as a promising future computing model, which enables elastic resource provisioning to meet unpredictable demands [5, 48]. In cloud computing, users run their programs in virtualized systems, and multiple virtual machines from different users may share the same physical system. However, such cloud computing based on virtualization poses a difficult challenge to securely isolate co-tenants sharing a physical system. The concern for security and privacy protection is one of the most important reasons that have been hindering the rapid adoption of cloud computing [38, 40].

In a virtualized system, a hypervisor, a software layer creating and managing virtual machines (VMs), is responsible for isolating any illegal accesses across virtual machine boundaries. However, there have been concerns over vulnerabilities in hypervisors [36, 37]. Although several recent studies improve the security of hypervisors [8, 30, 46], the hypervisors, which have been increasing their code sizes for better performance and more features, have become too complex to simply verify their secure execution [8]. If hypervisors cannot be trusted, even a trustworthy cloud provider cannot guarantee the protection of a virtual machine from a malicious co-tenant.

Memory protection across virtual machines is a critical component to support the secure execution of guest virtual machines, and to guarantee the privacy of user information. For such memory isolation, current virtualization techniques are based on traditional software and hardware supports for virtual memory. A hypervisor maps the memory of guest VMs to the real memory with guest-physical to machine address translation. Although several different techniques are used to support efficient address translation to the machine address space, all these techniques rely on hypervisors to prevent any illegal memory access across virtual machines [4, 31, 43]. A successful attack on the hypervisor can completely expose the memory contents of guest virtual machines.

In the current memory virtualization techniques, at the highest privilege level, hypervisors can control both aspects of memory virtualization, *memory allocation*, and *memory isolation through address translation*. A hypervisor determines a set of memory pages to be allocated for a VM, and maintains a mapping table from guest-physical to machine address (nested page table) for each VM. In this paper, we propose the decoupling of memory isolation from memory allocation, both of which are currently performed by the hypervisor. With decoupling, the role of a hypervisor is limited to the memory resource allocation to utilize the physical memory efficiently. The hardware processor is responsible for updating the

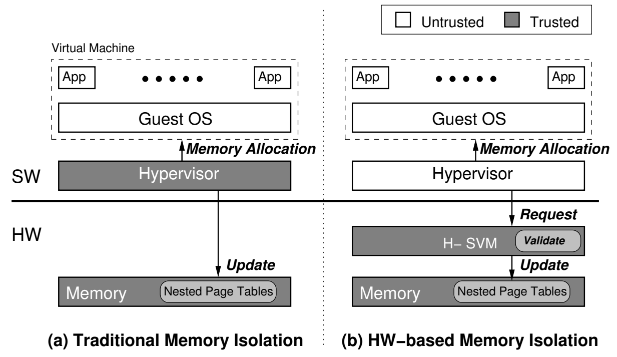

Figure 1: TCB of hardware-based VM memory isolation

page mapping and setting the pointer to the nested page table to schedule a VM on a core. Whenever the nested page table for a VM changes, the hardware checks whether the update is valid. By this decoupling, Trusted Computing Base (TCB) for memory isolation is reduced to the hardware processor from the combination of the hardware and hypervisor in conventional mechanisms. Figure 1 compares the hardware-based memory isolation with the traditional memory management by hypervisors. The shaded parts represent the TCB for memory isolation for each case.

There have been several prior studies to reduce TCB to the hardware processor to securely execute applications or virtual machines. AEGIS discusses a broad range of hardware-only architectures, or hardware architectures with trusted software to run applications in authenticated environments, even under a compromised operating system [39]. As AEGIS assumes hardware attacks, the sensitive data residing in the external memory is encrypted. In this paper, we focus on the protection of guest VMs when the hardware is securely protected in the data center of a cloud provider. With the restricted threat model, our goal is to minimize necessary changes in the current processor architectures and hypervisors as much as possible. This paper shows that modest extensions on the currently available hardware-assisted virtualization in commercial processors can lead to a significantly improved memory protection under an untrusted hypervisor, as long as the hardware is securely protected in the server room of the cloud provider. NoHype eliminates a software hypervisor entirely [22]. With NoHype, a VM runs only on one or more fixed cores, and the memory is partitioned for each VM. Unlike NoHype, our approach still supports the flexibility of software hypervisors, but the memory protection mechanism is decoupled from the hypervisors, and moved to the hardware processor.

This paper proposes a practical design for the hardware-based VM isolation, called *hardware-assisted secure virtual machine (H-SVM) architecture*, which aims to minimize the changes from the currently available architectural supports for virtualization. Based on H-SVM on each computing node, this paper also describes a cloud system architecture to initiate a VM with its integrity checked, and to migrate VMs across physical systems securely. The paper shows how H-SVM supports the confidentiality and integrity of guest VMs even under an untrusted hypervisor, and discusses our design decisions not to support availability.

The rest of the paper is organized as follows. Section 2 describes the current memory virtualization mechanisms, and the threat model for our study. Section 3 presents the architecture of the proposed H-SVM mechanism. Section 4 presents the cloud system architecture to guarantee the integrity of virtual machines during VM creations and migrations. Section 5 discusses security analysis and possible performance overheads. Section 6 discusses prior work, and section 7 concludes the paper.

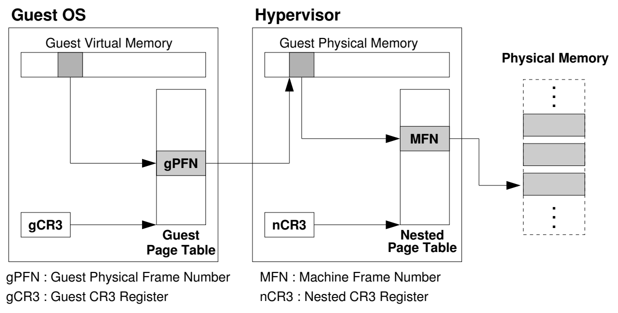

Figure 2: Address translation with nested paging

2. MOTIVATION

2.1 Hardware-Assisted Virtualization

Memory isolation in the current virtualization techniques is based on the support for virtual memory with hardware address translation and page tables. In processors, a virtual address is translated to a physical address with translation look-aside buffers (TLBs). If the corresponding entry does not exist in the TLBs, either a HW or SW-based page table walker fetches the entry from the page table of the current address space. In this paper, we assume a HW-based page table walker in our architecture, as the proposed architecture aims to move the responsibility of page table management from hypervisors to HW processors. For each context switch between address spaces, the page table register, which points to the top-level page table entry, must be properly set, so that the walker can traverse the correct page table. In the popular x86 architectures, the `CR3` register stores the address of the top-level page table.

Virtualization adds an extra translation layer. A virtual address in guest VMs must be translated into a guest-physical address (virtual physical address) like non-virtualized systems, and, moreover, the guest-physical address is translated to a machine address in the real physical memory. The guest OS maintains the virtual to guest-physical address mapping in per-process page tables, and the hypervisor maintains per-VM guest-physical to machine address mapping tables, called *nested page tables*. When the processor supports only single-page table walks designed for traditional native operating systems, the hypervisor maintains direct translation tables, called shadow page tables, to map virtual addresses to machine addresses directly.

The recent advancement of architectural support for virtualization allows a nested hardware page table walker to traverse both per-process page tables and per-VM nested page tables [4]. Figure 2 depicts the nested address translation with a per-process guest page table and a per-VM nested page table. Each core has two registers pointing to the two tables, one for the guest page table (`gCR3`), and the other for the nested page table (`nCR3`). With the hardware-assisted virtualization, the hypervisor has its own address space, but unlike guest virtual machines, the hypervisor address space uses a single translation without nested paging.

The hardware-assisted virtualization also facilitates a world switch between a VM and the hypervisor contexts. For example, in the AMD-V architecture [2], the context of each VM is defined in a Virtual Machine Control Block (VMCB). The hypervisor, at the host mode, executes the `vmrun` instruction to switch to a guest VM context. The hardware processor, by executing micro-coded routines, saves the current hypervisor context to a specified area and restores the guest VM context from the VMCB to the processor. The VM context contains the register states including the pointer to the nested page table. If an event, which must be handled by the hypervisor, occurs, the hardware saves the guest VM context in the

VMCB, and restores the hypervisor context.

To isolate the memory of each VM, a hypervisor must protect nested page tables from illegal modifications by guest VMs. Guest VMs cannot read or modify the nested page tables. Also, for each context switch between virtual machines on a core, the hypervisor must change the nested page table pointer directly or by executing the `vmrun` instruction. The hypervisor manages memory allocation by monitoring memory usages of virtual machines, and it can allocate and deallocate pages for a VM. For such mapping changes, the hypervisor can modify the nested page table of the VM. Note that the hypervisor also accesses its own memory space through the address translation mechanism. Since the hypervisor has a complete control over the modification of nested page tables, a compromised hypervisor can read or modify the physical memory assigned for any guest VMs. A malicious user can map physical memory pages already allocated for other VMs to the address spaces of its own VM or to the hypervisor.

2.2 Threat Model

To protect the memory of VMs even under a compromised hypervisor, the proposed mechanism allows only the hardware (H-SVM) to validate and update nested page tables, reducing the trusted computing base (TCB) to the hardware system. The proposed mechanism can be vulnerable to hardware attacks, such as probing external buses [6] or reading DRAM after power-off [17]. However, we assume that the cloud provider is trustworthy and it does not intentionally attempt to compromise the hardware system. The trustworthy cloud provider protects its servers with physical security measures, as the provider not only has a legal obligation not to access customers' data without explicit permission, but also has a strong business interest in protecting its reputation.

The TCB to protect the memory of guest VMs in the proposed system does not include the hypervisor. We assume that hypervisors are vulnerable to remote attacks by malicious guest virtual machines. The threat model assumes that an adversary with the root permission of the hypervisor may attempt to access the memory of guest VMs. The proposed system can protect the memory of guest VMs as long as the adversary cannot compromise the physical servers directly. With proper security measures on the server room, getting accesses to the server room, and physically compromising the systems are much harder than getting the root permission by remote attacks.

By not supporting hardware tamper-resistance, we simplify the requirements for H-SVM significantly. H-SVM moves the minimum functionality in traditional hypervisors for updating nested page tables to the hardware processor. The memory protection mechanism in this paper addresses the isolation among virtual machines or from the hypervisor. It does not improve the security of guest operating systems and applications by themselves.

2.3 Requirements for H-SVM

To guarantee VM isolation even under a compromised hypervisor, hypervisors must not be able to change nested page tables arbitrarily. Instead, the H-SVM hardware directly modifies nested page tables. One of the important requirements for H-SVM is to minimize necessary changes in the current hardware architecture and hypervisor, which are already very complex. The H-SVM support should be integrated into the current processor designs without a significant increase of complexity, and the interfaces to the hypervisor must also be simple. Furthermore, the changes to the guest OS must be very small or none to use many commodity operating systems.

Another requirement for H-SVM is that it must still allow hypervisors to manage memory pages as flexibly as the conventional memory management mechanism. Although a hypervisor loses the control for updating nested page tables directly, it must be able to allocate and deallocate memory pages dynamically for virtual machines. The static assignment of memory for each VM restricts memory management too severely to effectively utilize limited physical memory resources.

Furthermore, some physical memory can be shared between a VM and the hypervisor for data transfers with I/O devices. VMs also may share some physical pages for fast direct networking or content-based memory sharing. For such cases, a physical memory page can be mapped to the guest-physical pages of multiple VMs or the hypervisor. H-SVM must allow such sharing only after the guest OSes agree to the sharing of certain memory pages.

H-SVM must support the authenticated deployment and migration of virtual machines, even if hypervisors are compromised. Unless an authenticated guest OS is running on a virtual machine, the memory protection mechanism itself does not provide a trusted virtual machine to users. In this paper, we describe how to launch a VM with an authenticated guest OS image, and how to migrate a VM to different physical systems with its privacy protected. We will discuss the cloud system architecture in Section 4.

3. ARCHITECTURE

3.1 Overview

H-SVM improves memory isolation among VMs by blocking direct modifications of nested page tables by a hypervisor. Nested page tables for VMs are stored in the protected memory region, which can be accessible only by the H-SVM hardware. For any changes in memory allocation for a VM, the hypervisor, at the privileged level, makes a request to H-SVM to update the nested page table for the VM. If the hypervisor is compromised, it can try to allocate a physical memory page already assigned to a VM to the address space of the hypervisor or a malicious VM. Before updating the nested page table, H-SVM checks whether the request may violate memory isolation among VMs. If a physical memory page is deallocated from a VM, H-SVM cleans up the deallocated page by setting all the bytes to zeros.

Figure 3 presents the overall architecture of H-SVM. H-SVM can be implemented either as a separate controller in the processor chip, or can be added as microcode routines. Such microcode routines are commonly used to implement complex features in the x86 architecture. In the rest of the paper, we will assume such a microcode implementation. The nested page tables of all VMs are stored in protected memory pages. The protected memory region is just part of the physical memory, which is accessible only by H-SVM. H-SVM blocks accesses to the protected memory even from the hypervisor, by denying page mapping requests to the protected pages.

H-SVM maintains several data structures, including VM control information, nested page tables, and a *page ownership table*, in the protected memory region. The VM control information stores various control information for each VM, including the area to save register states, the address of the top-level nested page table, and an encryption key created for the VM. The VM control information is similar to the VMCB in the AMD-V architecture. The page ownership table tracks the owner of each physical memory page, and thus the number of entries is as large as the number of physical memory pages in the system. Each entry, corresponding to a physical page, records the ownership of the page. A VM, hypervisor, or H-SVM itself can be the owner of a page. If H-SVM is the owner of a page, the page is used for the protected memory area. The page

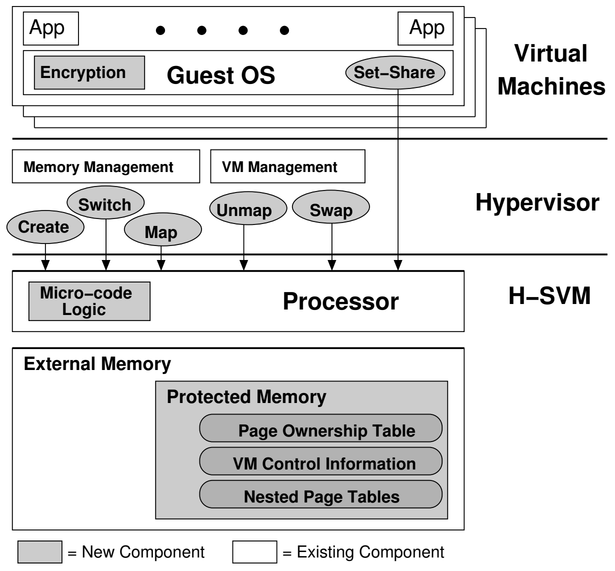

Figure 3: Hardware-assisted secure virtualization

ownership table is used to verify whether a page map request from the hypervisor is valid or not.

When the control is transferred to the hypervisor by interrupts, H-SVM must save the register states of the current virtual CPU (vCPU) to the VM control information. After saving the VM state, H-SVM sets the page table pointer to the page table used by the hypervisor. When the hypervisor schedules a vCPU to a physical core, the hypervisor requests H-SVM, by executing a privileged instruction, to place a VM to a core. For the scheduling request, H-SVM restores the VM state from the VM control information, including the nested page table pointer. As discussed in Section 2.1, the current x86 processors already support a similar world switch operation, such as the `vmrun` instruction in AMD-V. A main difference of H-SVM from the current support for `vmrun` is that H-SVM requires that the VM control information must not be accessible by the hypervisor.

With H-SVM, the hypervisor conducts normal memory management operations for VMs. For a VM creation, it decides a set of memory pages for the new VM, and makes requests to update the nested page table of the newly created VM. The hypervisor can also deallocate memory pages from a VM, often by the ballooning technique, but the actual updates of nested page tables occur in H-SVM. The role of H-SVM is limited only to the protected update of nested page tables and validation before any modification to nested page tables. Hypervisors still have a control over the memory resource management to assign memory pages to VMs.

3.2 Basic Interfaces

This section describes the basic interfaces of H-SVM. Hypervisors or VMs execute special instructions to make requests to H-SVM. There are four basic interfaces to initialize the VM control information, to update nested page tables, and to schedule a VM.

Create VM: When a hypervisor creates a VM, it requests H-SVM to create a new nested page table for the VM. H-SVM initializes the VM control information, and creates a nested page table for the VM. After the data structures are created in the protected memory area, H-SVM returns a VM identifier, which the hypervisor will use to designate the created VM for subsequent interactions with H-SVM. H-SVM also creates a per-VM encryption key, which will be used for page swap requested by the hypervisor.

Page map: To assign a physical memory page to a VM, the hypervisor requests a *page map* operation to H-SVM. A page map operation maps a machine memory page (frame) to a guest-physical page by updating a nested page table entry. The critical component for memory isolation is to check the ownership of a physical page for each page map operation by H-SVM. Before updating the nested page table entry, H-SVM must check, by looking up the page ownership table, whether the physical page is owned by another VM. If another VM already owns the requested physical page, the map operation is aborted. When a nested page table entry is updated for a VM by the request of the hypervisor, the VM becomes the owner of the physical page. This checking mechanism prevents a compromised hypervisor from creating any illegal mapping to the pages already used by other VMs.

Page unmap: To deallocate a physical memory page from a VM, the hypervisor makes a *page unmap* request to H-SVM. H-SVM modifies the corresponding nested page table entry, and clears the content of the memory page before completing the operation. H-SVM also resets the owner of the page in the page ownership table, marking it as a free page. With clearing, the contents of free pages cannot contain any information from prior guest VMs.

Schedule VM: When the hypervisor needs to schedule a VM to a core, the hypervisor makes a *schedule VM* request to H-SVM. The hypervisor cannot directly update the page table pointer to the nested page table. The hypervisor makes a VM schedule request with a VM identifier, and H-SVM sets up the register state from the VM control information. The hypervisor does not know the resister states including the address of the top-level nested page table. As only H-SVM can update the nested page table pointer and register states, the hypervisor cannot force a running VM to use a compromised nested page table. This operation is similar to `vmrun` in AMD-V, except that the VM control information is protected from the hypervisor.

3.3 Page Sharing and Swapping

Page Sharing: Some physical pages can be shared between a guest VM and the hypervisor, or among guest VMs. Such page sharing is used for communication for I/O requests, or content-based memory sharing [15, 29]. By default, H-SVM allows a physical memory page to be mapped only to a single VM. However, to make a page sharable, H-SVM needs an explicit permission from the guest VM. By requiring the permission from the guest OS, the guest OS can be prepared not to store any sensitive data on such shared pages.

Guest OSes, during their boot processes, may declare a subset of guest-physical pages as sharable pages. For example, in virtualized systems with the Xen hypervisor, guest OSes share some pages with the hypervisor to transfer I/O data. Guest OSes can identify such pages and allow page sharing with the hypervisor. Guest OSes execute an untrappable instruction to send a request to H-SVM to mark a page sharable.

Page Swap: In virtualized systems, both guest OSes and the hypervisor can swap out memory pages to disks to use the limited physical memory efficiently. For a given guest-physical memory capacity, a guest OS manages the memory with traditional OS-level memory management policies. However, in virtualized systems, the memory allocated for each VM can change dynamically. One of the most common techniques for dynamic VM memory management is a *ballooning* technique [45]. With ballooning, the guest OS decides which memory pages are rarely used, and deallocates the pages by assigning them to the balloon driver space. The ballooning technique can provide more efficient memory management than direct swapping by the hypervisor, since the guest OS on each VM can make much better decisions on selecting victim pages than the hypervisor. Using the technique, the guest OS releases unused memory pages to a pool of free memory pages, which can be used by other VMs. Furthermore, when a system is running out of the physical memory, the hypervisor may request each VM to free some memory pages.

For these page swap operations by a guest OS, the guest OS is responsible for encrypting the contents, and maintaining the integrity of swapped pages. Since the guest OS needs to protect its other files in addition to the swap file, it must use a secure file system, which can guarantee the confidentiality and integrity of its data in untrusted storage.

In addition to ballooning, the hypervisor may also evict memory pages to its disk swap space directly. However, the direct page swaps by the hypervisor are much less commonly used than the ballooning technique [45]. Although its uses are less common then ballooning, H-SVM can support direct page swapping by the hypervisor. As the hypervisor writes the memory pages to the swap disk, the contents of evicted memory pages must be encrypted by H-SVM before being accessible to the hypervisor. When a VM is created, H-SVM creates an encryption key for the VM. When the hypervisor picks a victim page, H-SVM encrypts the page with a per-VM key, and allows the hypervisor to read the page. Per-VM keys are stored in the protected memory region.

To maintain the integrity of swapped pages, H-SVM also needs to store the hashes of swapped out pages in the protected memory region. Since maintaining the list of hash values for swapped pages with a simple micro-coded routine may be complicated, H-SVM can use hash trees, which are used for integrity checking of data stored in untrusted storage [28]. Each VM has a hash tree for its swapped pages, and the hypervisor maintains the per-VM hash trees. However, H-SVM stores the top hash of each tree in the protected memory region. For a page swap request, the hypervisor must provide H-SVM with the hash values of the associated branch in the hash tree in a memory page. H-SVM will verify the provided hash tree by comparing the computed value with the stored top hash, and update the top hash with the hash of the new swap-out page. For a swap-in, the hypervisor must provide H-SVM with the hash values of the associated branch from the hash tree. H-SVM decrypts the page first, and its hash value is computed and verified with the branch.

3.4 Implementation Issues

H-SVM requires minor changes in the current HW support for virtualization, as the most of its basic functions can be implemented in microcodes. In this section, we discuss a possible implementation of H-SVM with microcodes and its performance overheads.

Microcode Implementation: Operations in H-SVM can be implemented with microcodes which are commonly used to support complex instructions. Each function requires relatively simple micro-ops sequenced from the microcode ROM in the front-end of the processor. Among the four basic interfaces, the `create VM` operation mostly copies a template of necessary initial setups for a new VM to memory. The `schedule VM` operation is the same as the `vmrun` instruction already supported by the current hardware-assisted virtualization. To show the complexity of H-SVM implementation, in this section, we show a possible implementation of the other two basic interfaces, `page map`, and `page unmap`. In this implementation, we assume that microcode executions in H-SVM use machine addresses directly for load and store operations to eliminate the overheads of changing address space between the hypervisor and H-SVM.

For all the implementations, a global spin lock is used to protect the ownership table and nested page tables, when multiple cores

Algorithm 1 map (VMID, GPA, MFN)

```
Input: VMID : virtual machine identification
       GPA : Guest-physical address
       MFN : Machine Frame Number
 1: lock
 2: $ownership \leftarrow ownership\_table[MFN]$
 3: if $ownership.owner$ is not null then
 4:     trap
 5: end if
 6: walk $nested\_page\_tables[VMID]$ for $GPA$
 7: if walk fails then
 8:     $return\_register \leftarrow disconnected\ mapping$
 9:     unlock
10:     exit
11: end if
12: $page\_table\_entry[GPA] \leftarrow MFN$
13: $ownership.owner \leftarrow VMID$
14: unlock
```

Algorithm 2 add_pt (VMID, GPA, MFN)

```
Input: VMID, GPA, MFN
 1: lock
 2: $ownership \leftarrow ownership\_table[MFN]$
 3: if $ownership.owner$ is not null then
 4:     trap
 5: end if
 6: initialize $memory\ page[MFN]$
 7: $ownership.owner \leftarrow HSVM$
 8: walk $nested\_page\_tables[VMID]$ for $GPA$
 9: if current level is not last level then
10:     $current\_entry.mfn \leftarrow MFN$
11:     if current level + 1 is not last level then
12:         $return\_register \leftarrow continue$
13:     else
14:         $return\_register \leftarrow success$
15:     end if
16: end if
17: unlock
```

request the operations. With a correctly running hypervisor, such locking may not be necessary, as the hypervisor must serialize the execution of the mapping change codes, which request map or unmap operations. However, malicious hypervisors may try to corrupt nested tables or the ownership table by causing a race condition intentionally. To avoid such an attack, we add a simple global spin lock to serialize the operations, which checks a fixed memory word in the protected memory region as a lock variable. In the example implementation, we did not attempt to optimize the performance with more fine-grained locking.

Algorithms 1-3 present the pseudo codes for `map`, `add_pt`, and `unmap` functions. A `map` operation in Algorithm 1 with a given target VM ID, guest physical address (GPA), and machine frame number (MFN), updates the nested page table entry of GPA with a new mapping to MFN. It first checks whether the current owner of the machine frame is *null* (free page). Otherwise, the execution will cause a fault. If the ownership checking succeeds, it walks the nested page table for the VM to fetch the page table entry for GPA. When a new machine frame is added to the VM, it is possible that intermediate and leaf entries in the page table are not yet constructed. Therefore, a nested page table walk can fail, if it encounters a missing intermediate or leaf entry in the page table during the walk. If a page walk fails, it returns the failure status to the hypervisor, and exits the map function. Otherwise, it updates the page table entry to the new MFN, and updates the owner for MFN in the ownership table.

If a page walk fails due to an incomplete page table, the hypervi-

Algorithm 3 unmap (VMID, GPA, MFN)

```
Input: VMID, GPA, MFN
 1: $clear\ memory\ page[MFN]$
 2: lock
 3: $ownership \leftarrow ownership\_table[MFN]$
 4: if $ownership.owner$ is $HSVM$ then
 5:     trap
 6: end if
 7: walk $nested\_page\_tables[VMID]$ for $GPA$
 8: $page\_table\_entry[GPA] \leftarrow NULL$
 9: $ownership\_table[owner] \leftarrow NULL$
10: $TLB\_flush(GPA)$
11: unlock
```

Algorithm 4 a map operation called from the hypervisor

```
1: $result \leftarrow map(VMID, GPA, MFN)$
2: if $result$ is not success then
3:     while $result$ is not success do
4:         $result \leftarrow add\_pt(VMID, GPA, freepage->mfn)$
5:         $freepage \leftarrow freepage->next$
6:     end while
7:     $result \leftarrow map(VMID, GPA, MFN)$
8: end if
```

sor calls `add_pt` with a free memory page, and H-SVM builds the incomplete part of the nested page table. Algorithm 2 shows how to add a new page table entry to a nested page table. The function first checks if the given MFN is a free page, and initializes the new page. Also, the owner of MFN must be updated to H-SVM, to include the newly added part of the nested page table in the protected memory. After initialization, the operation walks the nested page table to the last available entry, and sets the last entry with the address of the new free page. If the added one is the last level entry in the nested page table, it will return a *success* status. If the `add_pt` function returns a *continue* status, the hypervisor executes the operation again with a new free page to add the next level page table entry. This page table construction mechanism relieves H-SVM from the burden of managing its own free pages for nested page tables. The hypervisor provides H-SVM with a free page. H-SVM checks the ownership status of the free page, and constructs the nested page table.

Algorithm 4 shows how the hypervisor executes the map operation. If the initial map operation fails due to a page walk failure, it will try to add the necessary intermediate or leaf page table entry by calling `add_pt` with a free memory page. Once the page table construction is completed, `map` is called again.

The implementation of `unmap` is straightforward as shown in Algorithm 3. It clears the content of the given machine frame, and updates the corresponding nested page table entry. It also updates the page ownership table entry, and flushes the TLB for the given GPA. These microcode-based implementations may overwrite some register values. Those registers whose values are destroyed by the H-SVM operations, should be defined in the ISA, so that the hypervisor can save and restore those values.

To quantify the complexity of the microcode implementations, we have implemented micro-coded routines for the operations. The implementation is based on the instruction sequences compiled from the routines written in C. We slightly modified the compiled instruction sequences to have only simple RISC-like instructions. Table 1 summarizes the numbers of instructions for the operations. The numbers of static instructions are relatively small between 70 and 100 instructions for all the operations, showing that the complexity of each routine is low. As those routines are contained in the microcode ROM, the compromised hypervisor cannot modify them. The numbers of instructions executed in a normal execution without any fault are much larger in `unmap` and `add_pt` than in `map`, since those two operations contain a loop to clear or initialize a page.

Operation	# of inst.	# of inst. executed	# of mem insts. executed
Page map	82	79	18
Page unmap	70	1600	528
Add PT	94	1627	536

Table 1: Instructions for basic interfaces in microcode

Hypervisor Changes: To use H-SVM, the hypervisor needs to be slightly modified, but the modification is minor. The most of the hypervisor changes are to move some functions from the hypervisor to H-SVM. For example, for a map operation, the current hypervisor updates the nested page table directly, but with H-SVM, it is replaced with a request to H-SVM. To support the integrity of swapped pages by H-SVM, the hypervisor must maintain the hash tree for each VM for swapped pages, and request H-SVM to update the top hash value in the protected memory.

Guest OS Changes: By default, guest OSes do not need to be modified, if a VM always uses VM-private pages. To support page sharing, guest OSes must be slightly modified to add features to make page sharing requests. The guest OS can identify the region of guest physical memory used for device drivers to communicate with the hypervisor, and it sets the region as shared pages. To enable guest-level swapping, guest OSes must encrypt pages before sending them to guest swap spaces, and maintain the hash values for the swapped pages for integrity checking.

Performance Overhead: The performance overheads of H-SVM occur during page map and unmap operations. During a page map operation, H-SVM looks up the page ownership table, which may need several memory references. However, a page allocation operation is a relatively uncommon event, and even a few thousand extra processor cycles for checking should have a negligible impact on the overall performance. For page unmap operations, the page content must be cleared. However, page deallocation is often done in background.

Encryption overheads for swapping pages by guest OSes may reduce the system performance. Compared to the slow disk operation latencies, the encryption latency is relatively short. With the improving CPU performance and recent supports for hardware-accelerated encryption [19], the overhead will decrease. Furthermore, the guest OS may reduce the encryption overheads by not swapping out sensitive data as much as possible.

3.5 Protection from DMA

Securing nested address translation protects the memory of guest VMs from accesses through instruction executions in processors. The other source of memory accesses in systems is Direct Memory Access (DMA) for I/O handling. As the hypervisor or the domain0 VM controls I/O devices and DMA mechanisms, a compromised hypervisor can potentially read or corrupt the guest VM memory through DMA. H-SVM protects the memory of guest VMs from DMA, by extending IOMMU (I/O memory management unit), which is supported by both Intel VT-d and AMD-V [1, 18]. IOMMU supports an address translation mechanism from a device address space to the physical memory space. The address range for a DMA access from I/O devices must be translated to physical addresses with an I/O page table. Currently, the hypervisor manages and updates the I/O page table for each VM. During the address translation process, IOMMU enforces memory protection, by checking whether DMA accesses are allowed or not for the physical address range.

As DMA accesses are processed with an address translation mechanism using page tables, similar to memory accesses from CPUs, the same mechanism of protecting nested page tables is used to protect I/O page tables. Unlike the current IOMMU architecture, I/O page tables should reside in the protected memory region along with nested page tables, and only the hardware H-SVM must be allowed to update the I/O page tables directly. Guest VMs or the hypervisor can request H-SVM to update the DMA protection status of memory pages owned by themselves. H-SVM must allow the change of the protection status, only when the requesting VM or hypervisor owns the corresponding page by checking the page ownership table.

An alternative way of securing guest VM memory from DMA is to extend Device Exclusion Vector (DEV) [3]. DEV is a bit vector representing the entire physical memory at page granularity [3]. For each DMA operation, the hardware system checks whether the physical address range for the operation is allowed by checking the corresponding bits in DEV. Unlike IOMMU, DEV does not support address translation, but it can enforce memory protection for DMA. Currently, the hypervisor can change the content of DEV, and thus control whether DMA is allowed for each physical memory page. To protect guest memory from accesses through DMA, DEV must be moved to the protected memory region controlled by H-SVM.

4. CLOUD SYSTEM ARCHITECTURE

In the previous section, we presented the H-SVM architecture on a single physical system. However, deploying a VM securely to a system, and migrating a VM across multiple physical systems require the coordination of the cloud management system (cloud manager) and computing nodes. In this section, we describe how H-SVM interacts with the cloud manager for the authenticated creation and migration of VMs.

4.1 Overview

In the previous section, H-SVM protects a guest VM, assuming the VM is running a correct guest OS image. However, if the hypervisor is compromised, it can change the guest OS image to create a VM running a compromised OS, and after the guest OS starts, the compromised OS can make the sensitive memory region sharable. Therefore, the integrity of the created VM must be validated, and only after such a validation process, the VM should become available to users.

In this paper, we describe a simple cloud system organization using a separate cloud management system. The management system, or *cloud manager*, is a physically separate system from the computing nodes, and it does not run any user virtual machines. To deploy a user VM, the cloud manager makes requests to the hypervisor in a computing node. The user VM is created by the hypervisor, and its integrity is checked by a chain of validation processes rooted at H-SVM. The cloud manager also supervises the migration of a VM from a computing node to another. For the integrity checking of a newly created VM, and the secure transfer of VM images during migration, the cloud manager and H-SVM in the computing node must be able to communicate in an authenticated manner.

To support a trusted communication path between the cloud manager and H-SVM, each hardware processor has a unique public and private key pair. The private key never leaves the processor chip, and the public key is registered in the cloud manager. A similar technique is used for TPM (trusted platform module) [42], which is widely available in server systems. Unlike TPM, which is a separate chip connected as an I/O device, the private key for H-SVM is embedded in each processor. Such embedding of a unique key pair has also been discussed in several prior studies for hardware-rooted trustworthy systems [10, 24].

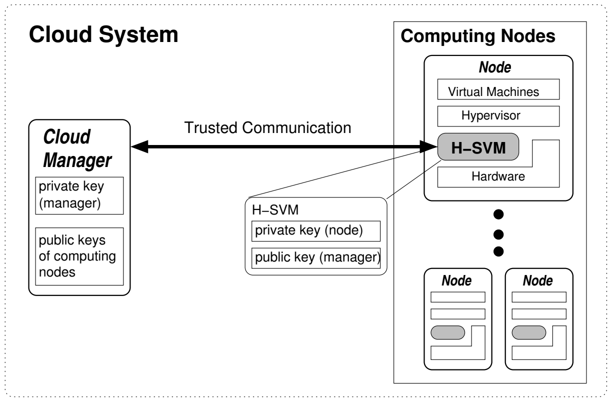

Figure 4: Authenticated communication between the cloud manager and a computing node

During the initial installation of computing nodes, the cloud manager keeps the list of public keys of all the computing nodes in the cluster. Any messages created by H-SVM are signed with its private key, and the cloud manager can authenticate the message by checking it with the public key of the sender system. Although the messages must be transferred through I/O paths provided by the potentially malicious hypervisor, the authentication prevents the hypervisor from forging messages between H-SVM and the cloud manager. We also assume H-SVM knows the public key of the cloud manager. The public key of the cloud manager is stored in the BIOS of the system, and cannot be updated by the hypervisor. During a boot process, BIOS copies the public key of the cloud manager to the memory area which will be used by H-SVM. Figure 4 depicts the cloud system organization with the cloud manager and computing nodes.

As the cloud manager and H-SVM in a computing node know the public key of each other, subsequent communications between the two systems can be authenticated, as even the compromised hypervisor, which provides network I/Os, cannot forge messages. In the next two sections, we sketch the processes for deploying a VM, and migrating a VM to another system.

4.2 Deploying a Virtual Machine

When a VM is created, H-SVM and the guest OS on the VM communicate to validate the integrity of the OS. During the booting procedure, the OS initiates a validation step by calling H-SVM. If the OS does not initiate the validation process, H-SVM does not allow the created VM to be used. The validation process consists of two steps. Firstly, H-SVM checks a small fixed location of guest-physical memory space, where a validation program resides. H-SVM receives a hashed value of the memory image of the program from the cloud manager, and verifies the program in the memory is a correct one. Secondly, once the integrity of the program is validated, H-SVM launches the program to check the entire memory content of the newly booted OS. Such program integrity checking has been discussed in prior work to validate OS and application program images [7, 8, 27, 39].

As the cloud manager must know the correct image of the booted OS, it sends the hash value of the memory image of the guest OS to H-SVM. Note that the communication between the H-SVM of computing nodes and the cloud manager is secure, even with an untrusted hypervisor between them. The hypervisor may block any packets from the cloud manager, to avoid the validation process, but in that case, the created VM will not be available to a cloud user. The two-step validation process minimizes the memory pages

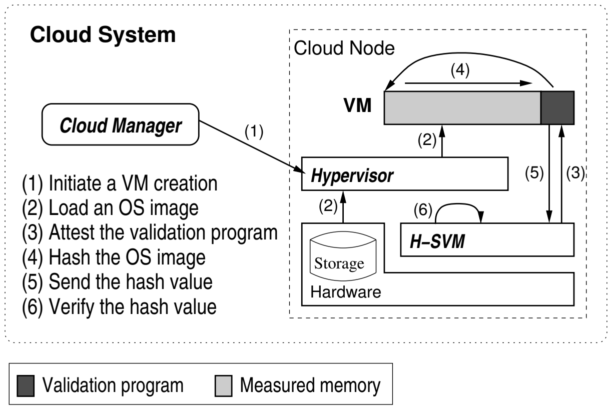

Figure 5: Virtual machine deployment procedure

which must be directly checked by H-SVM. Figure 5 depicts the authenticated VM creation initiated by the cloud manager (1). After the hypervisor loads the OS image (2), H-SVM attests the validation program (3). The validation program gets the hash of the entire OS image, and returns it to H-SVM (5). H-SVM compares the hash value with the one provided by the cloud manager (6).

4.3 Checkpointing and Migrating VMs

Virtual machines can be suspended and resumed later in the same or a different system. For a VM suspension operation, hypervisors create a checkpoint containing the memory contents and register states for a VM. To create such a checkpoint, conventional hypervisors must directly read the memory contents and register values of a VM. The created checkpoint is written to disk files or transferred to another system by networks. However, since a hypervisor cannot read the memory and register contents of a VM directly with H-SVM, H-SVM must encrypt those contents before they are exposed to the hypervisor. Live migration is similar to checkpointing, but it must be able to process each memory page independently. During a live migration process, the hypervisor will send the memory pages of a VM to another system, but some pages will be updated during the transmissions, as the VM is running. Therefore, the updated pages may be sent multiple times, until a short suspension of the VM to finalize the migration process. In both checkpointing and live migration, H-SVM must encrypt the contents of a VM.

In this paper, we assume that the cloud manager knows the list of computing nodes in the cloud system, and has the public keys of the computing nodes. Each computing node knows the public key of only the cloud manager, and it cannot directly communicate with another computing node in a secure manner. Therefore, the cloud manager must act as an intermediary for the transfers of encrypted data for migration from a source node to a destination node. The source node sends the encrypted data to the cloud manager using the public key of the manager. The cloud manager must decrypts the message with its private key, and must encrypt the decrypted message with the public key of the destination node, so that only the destination node can decipher the message. Although it may be possible to support a secure direct communication among computing nodes by exchanging the public key of each other through the cloud manager, it may increase the complexity of operations implemented in H-SVM.

This paper does not provide a complete protocol to securely transfer a checkpoint or an encrypted memory page. The protocol must provide the integrity of data as well as the confidentiality while minimizing the complexity of the H-SVM operations. As a computing node and cloud manager know the public key of each other, we believe such secure transmissions are possible. For example, H-SVM in a source node encrypts a memory page with a random symmetric key, and it encrypts the symmetric key, the VM identifier (VMID), and the guest physical page number (GPN) with the public key of the manager, to support the confidentiality. Each message should be signed with a digital signature using the private key of the source node to support the integrity of the message. The cloud manager must check the integrity of the received message, and decipher the symmetric key, VM ID, and GPN. To relay the message to the destination computing node, the cloud manager encrypts the symmetric key and the other information with the public key of the destination node, and signs the message with its private key for integrity. This overly simplified sketch is incomplete and includes possible security flaws. The simple scheme is vulnerable to replay attacks, as the hypervisor may use obsolete messages from the previous migration sessions. To avoid such replay attacks, the cloud manager must initiate a migration session with a nonce, and the sending node should include the nonce and message sequence number in all the messages for the migration session. Image transfers between the manager and destination node should use a similar protocol. Designing a secure protocol which is simple enough for H-SVM, and evaluating its performance overheads, will be our future work.

5. EVALUATION

5.1 Security Analysis

H-SVM supports the confidentiality and integrity of guest VMs, even if a software hypervisor is compromised. However, the availability of guest VMs is not fully guaranteed by H-SVM. In this section, we discuss what aspects of security can be supported by H-SVM.

H-SVM is vulnerable to hardware attacks, and cloud providers must protect the systems in its data center with physical security measures. The cloud management system, which provides authenticated launches and migrations of guest VMs, must be protected with higher security measures than computing nodes. Not exposing the cloud manager directly to cloud users will improve the security of the system. Also, the root password of the cloud manager must be available to fewer employees with a higher security clearance than the root passwords of computing nodes. H-SVM only protects guest VMs from malicious co-tenants or the compromised hypervisor. The security problems of guest operating systems and user applications, which exist even in isolated non-virtualized systems, will not be mitigated by H-SVM. However, the integrity checking mechanism for guest OS images, which is necessary to support H-SVM, will indirectly enforce cloud users to use at least validated OS images.

Confidentiality: The physical memory of a system can be accessed through two possible ways, the execution of memory instructions and DMA operations. Although some architectures support memory access modes, which do not use page-based address translation (e.g. real address mode in x86), H-SVM must disable such modes for guest VMs and the hypervisor, and force all memory instructions to use the hardware-based nested address translation. As H-SVM protects nested page tables for processor-side accesses, and I/O page tables with IOMMU for DMA-side accesses, even a compromised hypervisor cannot access the memory region not owned by itself.

However, H-SVM may not guarantee the confidentiality of guest VMs from side-channel attacks. As physical systems including caches can be shared by guest VMs, a malicious VM may attempt to extract confidential information through side-channel attacks [9].

There have been several studies for solutions to such attacks [32, 33, 47]. It is necessary to employ such solutions, orthogonal to the H-SVM mechanism, to prevent side-channel attacks.

Integrity: For the effectiveness of H-SVM, the integrity of system BIOS must be guaranteed. BIOS contains a critical setting information, which turns on or off H-SVM and sets the addressing mode. Furthermore, BIOS for x86 architectures also contains patches for microcodes used in processors, with which the routines of H-SVM are implemented. H-SVM may use a conventional mechanism using TPM (Trusted Platform Module) to validate the content of BIOS [42]. The cloud manager must remote-attest the BIOS content using TPM-based remote attestation protocols.

H-SVM in computing nodes and the cloud manager protect the integrity of guest OS images. As discussed in Section 4, H-SVM verifies the validation program, and the validation program verifies the OS image in the guest memory. Using the authenticated communication between the cloud manager and H-SVM, the hash of the OS image is compared to the known hash values stored in the cloud manager.

The confidentiality and integrity of I/O data used by a guest OS must be guaranteed by the guest OS, as the untrusted hypervisor transfers or stores the data. There have been numerous studies for protecting data in untrusted networks or storage, and the guest OS can use one of the mechanisms, which meet its security requirements [20, 23, 34]. If the guest OS uses a secure file system, protecting the integrity of swapped pages by the OS is straightforward, as the swap file is part of the file system. However, H-SVM is responsible for protecting the integrity of pages swapped by itself. As discussed in Section 3.3, H-SVM uses a hash tree for each VM to protect the swap file integrity.

A limitation of this study is that it does not provide a secure protocol for VM migration which must support the integrity as well as the confidentiality of migration data. The simplified sketch of a possible protocol in Section 4.3 is incomplete, and contains possible security flaws. The protocol should be simple enough to be processed by H-SVM, and the design of the protocol will be our future work.

Availability: H-SVM does not guarantee the availability of guest VMs. To allow complex but effective resource management by hypervisors, the hypervisors still control the resource allocation policies for guest VMs even with H-SVM. By allowing such a flexible management, a compromised hypervisor may not schedule guest VMs to CPUs, deallocate memory, or drop packets for VMs. Guest VMs may become unavailable, or their performance may drop. The use of system monitoring through the cloud manager can mitigate the availability problem. For example, the cloud manager can periodically check the availability of guest VMs by sending a message, and waiting for a response. Also, the guest VMs may track how many scheduling slots they actually receive from the hypervisor, and report any scheduling problem to the cloud manager. With H-SVM, the availability of guest VMs is sacrificed in favor of the flexibility of resource management by hypervisors.

Required hardware changes: Compared to traditional VM isolation supported by a hypervisor, H-SVM reduces the size of TCB for VM memory protection, and provides a strong isolation between the trusted HW components and the untrusted hypervisor, as the hypervisor cannot alter the H-SVM microcodes and the states of their execution. H-SVM requires modest changes in the hardware. Firstly, H-SVM must add the new instructions implemented in the microcode ROM. Secondly, certain registers must be accessible only during the executions of H-SVM microcodes. The registers specifying the context of a VM, including the register for the nested page table (`nCR3`), must be updated only by H-SVM microcodes. Thirdly, IOMMU must be supported, and H-SVM must protect I/O page tables. Also, only H-SVM must be allowed to change the register for the current I/O page table. Fourthly, each processor must have a unique pair of public and private keys for secure communications with the cloud manager. For performance, the processor may support a hardware acceleration for encryption instructions, but it is not required.

5.2 Performance Evaluation Methodology

To evaluate the performance impact of H-SVM, we modified the Xen hypervisor to emulate the overheads of page map and unmap operations. For a map operation, extra codes for checking and updating the ownership table have been added to the Xen hypervisor. The rest of operations defined by Algorithm 1 are already processed by the original Xen hypervisor. For an unmap operation, codes for clearing the page and updating the ownership table, have been added to Xen. For the overheads of encryption and decryption for swapping by a guest OS, we use an encryption file system, eCryptfs, in the guest OS for swap files[16]. To evaluate the impact of encryption for swapping, all guest swap-out operations encrypt the victim pages.

However, the performance evaluation in this paper includes neither the performance impact for page swap by the hypervisor, nor the costs for VM migration. In our setup, the Xen hypervisor uses only the ballooning technique for dynamic memory management for VMs, not supporting page swap by the hypervisor. Therefore, the evaluation only shows the performance costs of swap by the guest OS. Building a cloud system architecture, and evaluating the costs of authenticated VM initiation and migration will be our future work.

All experiments run on a quad core 2.8GHz i7 machine with 8GB DDR3 Memory. The i7 CPU supports an extensible page table (EPT), which is a variant of nested page tables. We use Xen 4.0.1 as the hypervisor, and Ubuntu 10.04 runs on each virtual machine as the guest operating system. We run the Xen hypervisor with the hardware-assisted full virtualization mode. The performance results are averaged from five runs.

In virtualized systems, page map operations occur intensively when a VM is created, to allocate the initial memory for the new VM. To evaluate the performance impact of extra steps of page map operations, we use a microbenchmark, `create_vm`, which creates VMs with various sizes of memory. As a VM has more memory, more page map operations must occur.

To evaluate the effect of dynamic memory allocation for virtual machines, we use the balloon driver for each VM. To increase dynamic memory allocation and deallocation rates, the balloon driver is configured to aggressively free any unused memory pages to a pool of free pages every five seconds. All guest VMs share the pool of free pages, and if a VM requires more physical memory pages, the hypervisor allows the requesting VM to use free pages in the pool. Each guest VM is created with 4GB memory initially, but the actual allocated physical memory for a VM changes dynamically. The second microbenchmark, `dyn_mem_alloc`, repeats the allocation and deallocation of 1GB memory 20 times. It allocates and touches a 1GB memory region, and deallocates the memory. The allocation and deallocation are repeated after a 10 second sleep period. During the sleep period, the balloon driver in the guest VM will deallocate unused memory pages to the free page pool.

To show the effect of dynamic memory allocation, we use two realistic mixed workloads, each of which runs three applications. The `mix1` workload consists of kernel compile, SPECjbb 2005, and SPECweb 2005 workloads. The kernel compile workload compiles Linux 2.6.38 vanilla kernel. The `mix2` workload consists of

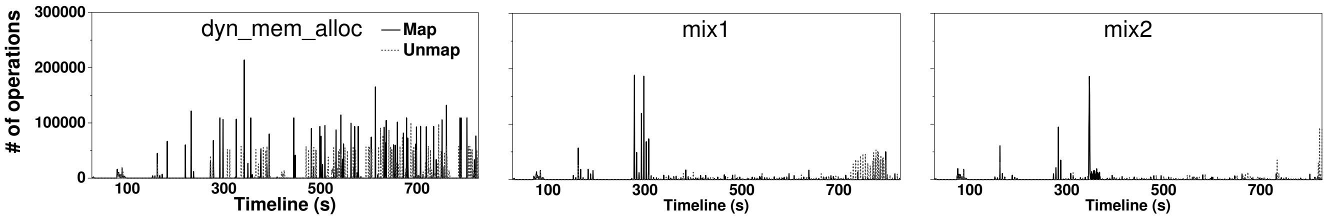

Figure 7: Page map and unmap operations during the execution of workloads

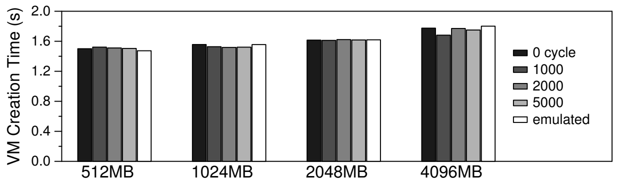

Figure 6: VM creation times with page map overheads

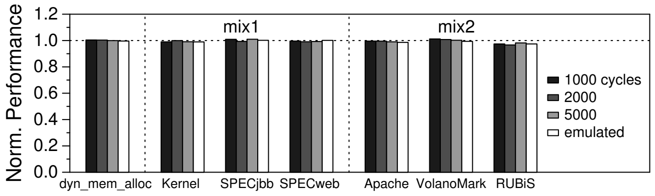

Figure 8: Performance with memory ballooning: normalized to zero-overhead runs

Apache compile, VolanoMark [44] and RUBiS [35]. The Apache compile workload compiles the Apache web server 2.219 version. For the two compile workloads, the inverse of elapsed time is used as the performance metric, and for SPECjbb2005, we use the performance score as the performance metric. For SPECweb2005, VolanoMark, and RUBiS, the average throughput is used as the metric.

5.3 Page Map Overhead for VM Creation

We first evaluate the performance impact of page map operations for a VM creation. The performance overhead of a page map operation is due to the extra steps accessing the page ownership table to verify each change and to update the ownership table. Figure 6 shows the total elapsed times for a VM creation, with the emulation codes for accessing the ownership table. We also show the effect of 1000, 2000, or 5000 extra cycles for each page map operation, which adds fixed extra cycles instead of the emulation codes. We use four VM types with memory sizes from 512MB to 4GB. The VM creation time is from the initiation of a VM creation, to the completion of VM construction. It does not include the time to actually boot the guest OS on the constructed VM. Note that in Xen, by default, a VM is created with a large page size of 2MB for most of its allocated memory, and large pages are broken down to smaller 4KB pages once they are reclaimed by the balloon driver. The VM creation times increase as the memory sizes increase. However, the emulation codes do not increase the latencies in a meaningful way. The performance impact by the increased overhead of page map operations is negligible. The results show that even during a VM creation period, when page map operations occur very intensively, the performance impact of H-SVM is negligible.

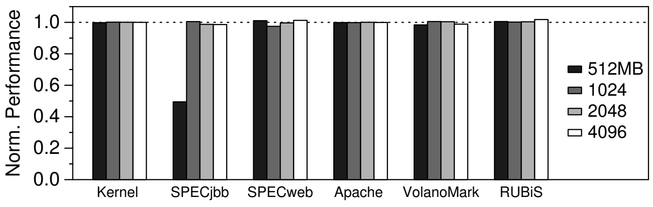

Figure 9: Performance with encrypted swapping: normalized to un-encrypted runs

5.4 Dynamic Memory Mapping Changes

Using the balloon driver in each VM, we evaluate the impact of dynamic memory allocation and deallocation with H-SVM. Figure 7 presents the number of page map and unmap operations during the execution of `dyn_mem_alloc`, `mix1`, and `mix2`. The solid lines are the numbers of page map operations, and the dotted lines are the numbers of page unmap operations. For all three cases, there are some page map and unmap operations soon after the VM creations (50-200 seconds). It is because the balloon driver on each VM with 4GB memory releases the unused memory, and during the release, large pages are divided into many small 4KB pages, executing map and unmap operations. For `mix1` and `mix2`, there are high rates of page map operations between 300-400 seconds, when the applications on VMs start requesting more pages. For `dyn_mem_alloc`, the memory mappings for VMs change very dynamically, as we designed the microbenchmark to generate high rates of memory mapping changes.

However, the performance impacts by page map and unmap operations are negligible for all three workloads. Figure 8 presents the performance of each application for the three workloads. In the figure, we also evaluate fixed extra cycles of 1000, 2000, or 5000 for map operations, to compare them against the emulated overheads. The figure shows that the performance changes are negligible in all the results within possible error ranges, as the performance slightly improves or degrades in random patterns. Even for the microbenchmark, which is designed to generate many memory allocations and deallocations, the performance impact by page map and unmap operations is negligible.

5.5 Impact of Encryption for Page Swap

Another possible performance overhead of supporting H-SVM is the cost for encryption and description for page swaps. In this evaluation, we assume all pages are encrypted for swap-outs, and decrypted during swap-ins. Figure 9 presents the performance with the swap encryption and decryption. In these runs, only one guest VM is created, and an application runs on the VM. For each VM, the memory size varies from 512MB to 4GB. Each bar presents the performance normalized to an unencrypted run for each configuration. As shown in the figure, the encryption for swapping does not cause significant performance overheads, if a VM has more than 1GB memory for our benchmark applications. Although there are some swap operations with the memory larger than 1GB, the performance impact for encryption is minor, as the disk latency dominates the swap overheads. However, for SPECjbb2005, with 512MB memory, the VM is continuously swapping in and out the memory pages because of memory thrashing. In that case, the overall performance can drop by as much as 53%. Such case is the worst case scenario, when the entire system is thrashing.

6. RELATED WORK

There have been many prior studies to embed security features into hardware processors, under the assumption that compromising hardware components is harder than attacking software systems. Such hardware attacks often require physical accesses to systems as well as more sophisticated attacking methods than remote software attacks. Prior studies for hardware-oriented protection reduce TCB to the hardware system without software stacks, or with a thin verifiable software layer [24, 26, 39].

The Execution-Only Memory (XOM) architecture provides the integrity and privacy of applications with a hardware mechanism, supporting copy and tamper-resistance[26]. A subsequent work extends the supports for copy and tamper-resistance for applications even under an untrusted OS [25]. AEGIS supports tamper-evident and tamper-resistant environments using a combination of secure kernel and hardware mechanisms [39]. It proposes a broad range of solutions, assuming either a trusted or untrusted kernel. Both XOM and AEGIS can reduce TCB to the hardware processor and assume that all the other system components, such as external memory, are vulnerable. To provide solutions for attacks to the hardware components other than the processor, the systems use sophisticated hardware-based integrity checking and encryption mechanisms.

The Secret-Protected (SP) architecture proposes an architectural support to protect sensitive information in applications [24]. Even if the OS and other part of an application are untrusted, SP protects a certain part of the application space (trusted software module). Bastion supports multiple trusted domains based on attestation mechanisms and secure storage, and protects the hypervisor from physical attacks as well as software attacks [10]. Loki enforces application security policies by using a tagged memory architecture [51]. In the tagged memory architecture, a tag associated with a memory region carries security policies at word granularity, and the hardware processor enforces the policies embedded in tags.

Recently, there have been several studies for software-based VM memory protection mechanisms, relying on trusted hypervisors. In such approaches, the hypervisor must be included in TCB. Overshadow uses memory shadowing to protect sensitive applications from a malicious OS [11]. When an application is running, the hypervisor provides the application with a normal memory view. However, if the kernel is running, the hypervisor provides the kernel with a limited memory view for applications, which shows only the encrypted application data. SP3 uses a similar memory shadowing mechanism, but SP3 determines the memory view according to the access permission set to each memory page [50].

Using virtualization techniques, systems can be divided into trusted and untrusted compartments. The Terra system supports closed-box virtual machines called trusted execution environments [14]. It verifies the components of virtual machines such as the boot loader, kernel, and sensitive applications using a trusted platform module (TPM) at the start-up time. Proxos is similar to the Terra system in providing isolated and trusted computing environments [41]. While Terra does not allow the use of commodity OS resources for a trusted execution environment, Proxos supports the use of commodity OS resources by partitioning system call interfaces and routing system calls from a trusted VM to a normal VM.

Several recent studies discuss the vulnerability of hypervisors, and propose solutions to improve their security. To prevent control-flow hijacking attacks to hypervisors, Hypersafe assures the control-flow integrity of a running hypervisor [46]. HyperSentry measures the integrity of a running hypervisor in a stealthy way to prevent the compromised hypervisor from disabling the integrity checking component [8]. In the Xen hypervisor, the management domain (domain0) often becomes the target of attacks, as it has a higher privileged functionality than guest VMs, but has a similar vulnerability. Gordon divides the management domain into trusted and untrusted components to minimize TCB [30]. Li et al. proposed a secure VM execution environment to protect guest virtual machines from the untrusted management domain [12]. With this scheme, the management domain cannot access any memory of a virtual machine after the creation of the virtual machine.

CloudVisor adds an extra software layer under a conventional hypervisor, to protect virtual machines from a compromised hypervisor [13]. This approach is similar to our proposed system, as both systems attempt to separate critical functions of updating security-sensitive data structures from the hypervisor. However, CloudVisor uses a software-based approach to add an extra layer using the current hardware support for virtualization. This paper extends our preliminary study of hardware-based secure virtualization with a detailed system architecture, cloud architecture, and evaluation [21].

7. CONCLUSIONS

In this paper, we proposed a hardware-based mechanism called H-SVM to isolate the memory of a VM securely even under a compromised hypervisor. Unlike prior hardware-based mechanisms which support tamper-evidence or tamper-resistance for a hardware attack, H-SVM simplifies the complexity of hardware supports significantly by assuming a software-only threat model. We believe this restricted threat model is appropriate in the current cloud computing environments, where systems can be protected from physical intrusions in an isolated data center. We also discussed that a virtual machine can be deployed in an authenticated manner with the physically isolated cloud management system. The performance evaluation by emulating the overheads in the Xen hypervisor, shows negligible performance degradations.

8. REFERENCES

[1] Advanced Micro Devices. AMD I/O Virtualization Technology (IOMMU) Specification, 2009.

[2] Advanced Micro Dvices. Secure Virtual Machines Architecture Reference Manual, 2005.

[3] Advanced Micro Dvices. AMD64 Architecture Programmer's Mannual: Volume 2: System Programming, 2007.

[4] Advanced Micro Dvices. AMD-V Nested Paging, 2008.

[5] Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2`, 2008.

[6] R. Anderson and M. Kuhn. Low Cost Attacks on Tamper Resistant Devices. In *Security Protocols: 5th International Workshop, LNCS*, pages 125–136, 1997.

[7] A. M. Azab, P. Ning, E. C. Sezer, and X. Zhang. HIMA: A Hypervisor-Based Integrity Measurement Agent. In *Proceedings of the 2009 Annual Computer Security Applications Conference, ACSAC 2009*, pages 461–470.

[8] 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 *Proceedings of th 17th ACM Conference on Computer and Communications Security, CCS 2010*, pages 38–49.

[9] D. J. Bernstein. Cache-timing attacks on AES. Technical report, 2005.

[10] D. Champagne and R. B. Lee. Scalable Architectural Support for Trusted Software. In *Proceedings of the 16th IEEE International*

Symposium on High-Performance Computer Architecture, HPCA 2010, pages 1–12.

[11] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports. Overshadow: a Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008*, pages 2–13.

[12] A. R. Chunxiao Li and N. K. Jha. Secure Virtual Machine Execution under an Untrusted Management OS. In *In Proceedings of 2010 IEEE 3rd International Conference on Cloud Computing (CLOUD), CLOUD 2010*, pages 172–179.

[13] H. C. Fengzhe Zhang, Jin Chen and B. Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *To Appear the 23rd ACM Symposium on Operating Systems Principles, SOSP 2011*.

[14] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles, SOSP 2003*, pages 193–206.

[15] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference Engine: Harnessing Memory Redundancy in Virtual Machines. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation, OSDI'08*, pages 309–322, 2008.

[16] M. A. Halcrow. eCryptfs: An Enterprise-class Cryptographic Filesystem for Linux. In *Proceedings of the Linux Symposium, Linux 05*, pages 201–218.

[17] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest We Remember: Cold-boot Attacks on Encryption Keys. *Commun. ACM*, 52:91–98, May 2009.

[18] Intel. Intel Virtualization Technology for Directed I/O, 2011.

[19] Intel Corporation. Intel Advanced Encryption Standard (AES) Instruction Set, 2011.

[20] T. Jaeger, R. Sailer, and U. Shankar. PRIMA: policy-reduced integrity measurement architecture. In *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies, SACMAT 2006*, pages 19–28.

[21] S. Jin and J. Huh. Secure MMU: Architectural Support for Memory Isolation among Virtual Machines. In *Proceedings of the 7th Workshop on Hot Topics in System Dependability, HotDep 2011*.

[22] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. NoHype: virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th annual international symposium on Computer architecture, ISCA 2010*, pages 350–361.

[23] G. H. Kim and E. H. Spafford. The Design and Implementation of Tripwire: a File System Integrity Checker. In *Proceedings of the 2nd ACM Conference on Computer and communications security, CCS 1994*, pages 18–29.

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

[25] D. Lie, C. A. Thekkath, and M. Horowitz. Implementing an Untrusted Operating System on Trusted Hardware. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 178–192, 2003.

[26] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems, ASPLOS 2000*, pages 168–177.

[27] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. D. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy, S&P 2010*, pages 143–158.

[28] R. C. Merkle. Protocols for Public Key Cryptosystems. In *Proceedings of the 1980 IEEE Symposium on Security and Privacy, S&P 1980*, pages 122–134.

[29] G. Miłós, D. G. Murray, S. Hand, and M. A. Fetterman. Satori: Enlightened Page Sharing. In *Proceedings of the 2009 conference on USENIX Annual technical conference, USENIX'09*, 2009.

[30] D. G. Murray, G. Milos, and S. Hand. Improving Xen Security through Disaggregation. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, VEE 2008*, pages 151–160.

[31] G. Neiger, A. Santoni, F. Leung, D. Rodger, and R. Uhlig. Intel Virtualization Technology: Hardware Support for Effcient Processor Virtualization. *Intel Technology Journal*, 10(03):167–178, 2006.

[32] D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: the Case of AES. In *RSA Conference Cryptographers Track, CT-RSA 2006*, pages 1–20, 2005.

[33] D. Page. Defending against cache-based side-channel attacks. *Information Security Technical Report*, 8:30–44, March 2003.

[34] A. G. Pennington, J. L. Griffin, J. S. Bucy, J. D. Strunk, and G. R. Ganger. Storage-Based Intrusion Detection. *ACM Transactions on Information and System Security*, 36(7):18–29, 2003.

[35] RUBiS Benchmark. `http://rubis.ow2.org`, 2008.

[36] Secunia Vulnerability Report: VMware ESX Server 4.x. `http://secunia.com/advisories/product/25985/`, 2010.

[37] Secunia Vulnerability Report: Xen 3.x. `http://secunia.com/advisories/product/15863/`, 2010.

[38] Security Is Chief Obstacle To Cloud Computing Adoption. `http://www.darkreading.com/securityservices/security/perimeter/showArti%cle.jhtml?articleID=221901195`, 2009.

[39] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing. In *Proceedings of the 2003 International Conference on Supercomputing, ICS 2003*, pages 160–171.

[40] Survey: Cloud Computing "No Hype", But Fear of Security and Cloud Slowing Adoption. `http://www.circleid.com/posts/20090226_cloud_computing_hype_security`, 2009.

[41] R. Ta-Min, L. Litty, and D. Lie. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 279–292.

[42] Trusted Platform Module. `http://www.trustedcomputinggroup.org/developers/trusted_platform_module%`.

[43] VMware ESX and ESXi. `http://www.vmware.com/products/vsphere/esxi-and-esx/index.html`, 2010.

[44] VolanoMark. `http://www.volano.com/benchmark`, 2009.

[45] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th symposium on Operating systems design and implementation, OSDI'02*, pages 181–194, New York, NY, USA, 2002. ACM.

[46] Z. Wang and X. Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *IEEE Symposium on Security and Privacy, S&P 2010*, pages 380–395.

[47] Z. Wang and R. B. Lee. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. *ACM SIGARCH Computer Architecture News*, 35:494–505, May 2007.

[48] Windows Azure Platform. `http://www.microsoft.com/windowsazure/`, 2010.

[49] Xen Hypervisor. `http://www.xen.org/`, 2010.

[50] J. Yang and K. G. Shin. Using Hypervisor to Provide Data Secrecy for User Applications on a Per-Page Basis. In *Proceedings of the 4th International Conference on Virtual Execution Environments, VEE 2008*, pages 71–80.

[51] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. Hardware Enforcement of Application Security Policies Using Tagged Memory. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008*, pages 225–240.