Multi-Processor Architectural Support for Protecting Virtual Machine Privacy in Untrusted Cloud Environment

Yuanfeng Wen[1], JongHyuk Lee[1], Ziyi Liu[1], Qingji Zheng[2], Weidong Shi[1], Shouhuai Xu[2], and Taeweon Suh[3]
1. University of Houston, Houston, TX 77004, USA
2. University of Texas at San Antonio, San Antonio, TX 78249, USA
3. Korea University, Seoul, 136-701 South Korea
{wyf,ziyiliu, larryshi}@cs.uh.edu, jonghyuk.lee@daum.net, {qzheng, shxu}@cs.utsa.edu, suhtw@korea.ac.kr

ABSTRACT

Virtualization is fundamental to cloud computing because it allows multiple operating systems to run simultaneously on a physical machine. However, it also brings a range of security/privacy problems. One particularly challenging and important problem is: how can we protect the Virtual Machines (VMs) from being attacked by Virtual Machine Monitors (VMMs) and/or by the cloud vendors when they are not trusted? In this paper, we propose an architectural solution to the above problem in multi-processor cloud environments. Our key idea is to exploit hardware mechanisms to enforce access control over the shared resources (e.g., memory spaces), while protecting VM memory integrity as well as inter-processor communications and data sharing. We evaluate the solution using full-system emulation and cycle-based architecture models. Experiments based on 20 benchmark applications show that the performance overhead is 1.5%–10% when access control is enforced, and 9%–19% when VM memory is encrypted.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General–System architectures

General Terms

Security, Design

Keywords

Multi-Processor Architectural Support, VM Privacy, Cloud

1. INTRODUCTION

Cloud computing is revolutionizing the information technology, ranging from personal to enterprise to government computing. While cloud computing can provide computational and storage resources on demand and at a low cost, it creates new security/privacy problems. This is fundamentally caused by the separation of resource users (i.e., cloud tenants) from resource owners (i.e., cloud providers).

New threats and concerns include: (i) failures in ensuring separation between tenants in terms of storage and memory; (ii) subversion of hypervisor or Virtual Machine Monitor (VMM) [1]; (iii) attacks launched from one Virtual Machine (VM) against the host platform or the other colocated VMs on the same platform [2, 3]; (iv) eavesdropping a tenant's VM contents by a compromised VMM, untrusted resource owners, or malicious insiders. These threats have caused a large degree of reluctance in adopting the cloud paradigm [4, 5]. According to a survey of more than 500 global executives and IT managers in 17 countries [6], 20% executives trust their internal systems over the cloud due to concerns about security threats and loss of control over data and systems. Indeed, many data center customers demand their services to be hosted by dedicated servers that are physically isolated from other customers' servers. This would ruin, to a large extent, the merits of cloud computing that are essentially based on virtualization and sharing of physical resources. For example, according to VMWare [7], the number of VMs executed per core can be up to 16 VMs/core.

There have been studies on solving some of these problems. However, most studies are based on the assumptions that cloud providers are trusted and VMMs are secure. These assumptions are questionable because of insider threats and VMM vulnerabilities [8, 9, 2, 3, 1]. In this paper, we aim to address: how can we use hardware architecture to reduce the amount of trust one has to put in the cloud vendors or VMMs? For this purpose, we propose a security-enhanced multi-processor cloud server design, which uses hardware-enforced access control to manage shared resources and cryptography to protect confidentiality to VM images and states (e.g., register states, physical memory). The main contributions of our work are: (i) design of an architectural solution for enhancing VM privacy protection in "untrusted" cloud environments; (ii) architectural solution for privacy protection on multi-socket, multi-processor servers; (iii) hardware approach for protecting VM integrity in multi-core environments; (iv) evaluation of the proposed solution using system emulation and cycle-based full-system simulators. To our knowledge, our paper is the first that tackles the VM privacy and integrity problem in *multi-processor* untrusted cloud computing environments, where the VM memory space is mapped to distributed physical RAMs of a server.

2. PROBLEM STATEMENT

At a high level, we want to ensure that only the VM owner can have access to the VM contents in the presence of the following threats.

Untrusted cloud vendors. Cloud vendors cannot be fully trusted by the cloud users for many reasons. For example, cloud vendors might have the incentive to backup, replicate, and store cloud users' VM contents for purposes such as optimizing cloud service performance, and therefore may inadvertently breach privacy of the cloud users. Moreover, as clouds become increasingly more decentralized and distributed, there could be *phishing clouds* or malicious cloud infrastructures that are set up by the adversaries.

Insufficient VMM security. VMMs have access to the entire memory space as well as VM states. Even in hardware-assisted virtualization, system states (e.g., page tables) are maintained by VMMs using tracing or shadowing technique, where a shadow copy of the system states (e.g., virtual machine page tables) are kept and maintained by the VMMs. Because VMM sees and manages the VM contents, it can attack the VM memory in any fashion it wants.

Insufficient memory protection. The VM memory contents can be eavesdropped by using simple hardware-based approach that can bypass the software protection mechanisms. For example, hardware-based RAM capture devices can scan and dump physical memory contents [10], while bypassing the guest OS [10].

Insufficient security support for multi-socket platforms. There is no mechanism that protects the communications between the nodes in an Opteron-like multi-processor system. Hyper-transport implements a packet-based communication protocol for data transfer. Because hyper-transport itself does not provide any protection on communications against tampering, in order to enable security protection on multi-node based systems, one needs to add extra protection measures (e.g., message authentication code) to safeguard inter-node packet communications.

3. ARCHITECTURAL DESIGN

Our goal is to prevent most software-based attacks and physical eavesdropping attacks against VM memory space. The design exploits trusted processors and I/O controllers to achieve hardware-assisted virtualization, while using additional safeguards for VM privacy and integrity.

3.1 Assumptions

We assume that the processor hardware is trusted, which is the minimum assumption any solution might have to make. This is consistent with the threat model that attacks may come from a compromised VMM, a cloud insider, or an untrusted cloud provider. We assume that a user VM is subject to eavesdropping attacks by which the adversary can exploit software or hardware (e.g., a hybrid DDR-SSD RAM) to compromise the data privacy in a VM. We assume that the adversary can launch low-cost physical attacks but not sophisticated one that are below the computer chip level (e.g., attacks that require micro-probing or chip de-packaging); the sophisticated attacks can be dealt with using solutions that are orthogonal to our effort. We assume that side-channel attacks against cryptographic keys are handled separately by complementary solutions that can be used in conjunction with our solution. In short, the assumption about trusted hardware means that security is supported from below the VMM level and cannot be tampered by the VMM or cloud vendors.

3.2 Basic Ideas

The basic ideas underlying the solution is highlighted as follows. First, motivated by the observation that current virtualization grants too much power to the VMM than what is actually necessary, we aim to reduce the power and functionality of the VMMs and confine their responsibilities to resource management and scheduling. Second, instead of trusting the large number of cloud vendors, we trust the micro architectural components. This effectively reduce the number of parties that we have trust. Third, instead of adopting a privilege hierarchy, which is the traditional architecture-design principle, we rely on a fine-grained privilege system where architectural components, VMMs and VMs have their unique responsibilities and privileges. Fourth, conceptually, our solution can be applied to any multi-processor systems based on distributed shared memory, such as Xeon servers using QuickPath interconnect and hyper-transport based multi-processor PowerPC systems.

3.3 Overall Architecture

As shown in Figure 1, we extend hardware-assisted virtualization with additional software and architecture features. Specifically, a processor vendor can integrate a set of architectural components on-chip to safeguard VM memory space from eavesdroping. Functions of the new architectural components include: (i) security enhancement to the processor core such as special control registers and support for special instructions; (ii) hardware mechanism to provide privacy-enhanced protection of VM contexts; (iii) hardware memory access control engine (i.e., memory access firewall) that is integrated into each processor core; (iv) a fine-grained privilege system; (v) privacy-enhanced SRI to safeguard per-VM physical memory space via an integrated crypto engines. In our solution, VMM is still responsible for managing resources and supporting multi-tenant environments, where multiple VMs can reside on the same physical platform. However, each VM is responsible for setting its own privacy policy and for communicating its privacy policy to the trusted architecture. The new or enhanced components will be elaborated below.

Cryptographic keys. A hardware vendor (e.g., Intel, AMD) can create a pair of public/private keys $(\mathsf{PK}_v, \mathsf{SK}_v)$, where the private key SK_v is fused into its processors and the public key PK_v is made public. When a VM is bootstrapped, a pair of symmetric keys (k_1, k_2) is shared between the processors $P_1, \ldots, P_m$. This can be achieved by letting one processor (for example) P_1 choose (k_1, k_2) according to some appropriate cryptographic algorithms. Then, P_1 sends (k_1, k_2) to P_j, $2 \leq j \leq m$, under the protection of PK_v over the connections that couple P_1 and P_j. Note that a malicious physical man-in-the-middle may try to cause different P_i's to see a different set of processors. As a consequence, the processors do not actually share the same (k_1, k_2) and cannot communicate effectively. Because the processors are assumed to be

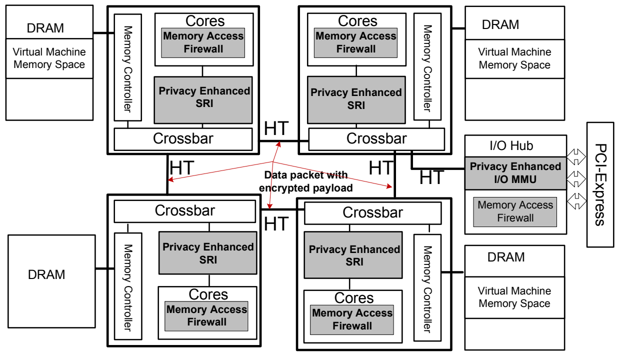


Figure 1: Quad CPU hyper-transport platform with privacy-enhanced nodes.

secure, this only causes a kind of denial-of-service attack, which can always be launched by a powerful adversary who has full control of the hardware. Each processor uses k_1 to maintain a data structure for storing its data in memory in encrypted form and the ciphertext can be decrypted by other processors that are allocated to the same VM. When a processor needs to communicate with another processor, k_2 is used to protect the communication.

Each cloud user can have a pair of public/private keys ($\mathsf{PK}_u, \mathsf{SK}_u$), which can be used to digitally sign the (encrypted version of) VM images. The VM images are encrypted by the cloud users offline before they are executed by a secure processor, while the image decryption key can be encrypted using the public key PK_v. The encrypted VM images can be decrypted only by the processors that hold SK_v.

We stress that letting all the processors manufactured by the same vendor possess the same private key PK_v is for the sake of the system description. The rationale is the following: Since we cannot pre-determine which user's applications will run in which processor, any trusted processor must be able to decrypt the encrypted virtual machine image of any user, which cannot be fulfilled by letting a user and a processor conduct a key-exchange protocol so as to encrypt the virtual machine image that can be decrypted by the processor. This problem can be solved using a new cryptographic technique called Proxy Re-Encryption [11, 12, 13]. Putting into the context of the present paper, we can let each vendor generate a unique "re-key" for each of the processors it manufactured, denoted by $sk_{processor}$. This technique allows the processor to decrypt the ciphertext that is generated using PK_v, where the plaintext content can be a symmetric key that is actually used to encrypt the virtual machine image. Moreover, even if $sk_{processor}$ is compromised, PK_v is still secure.

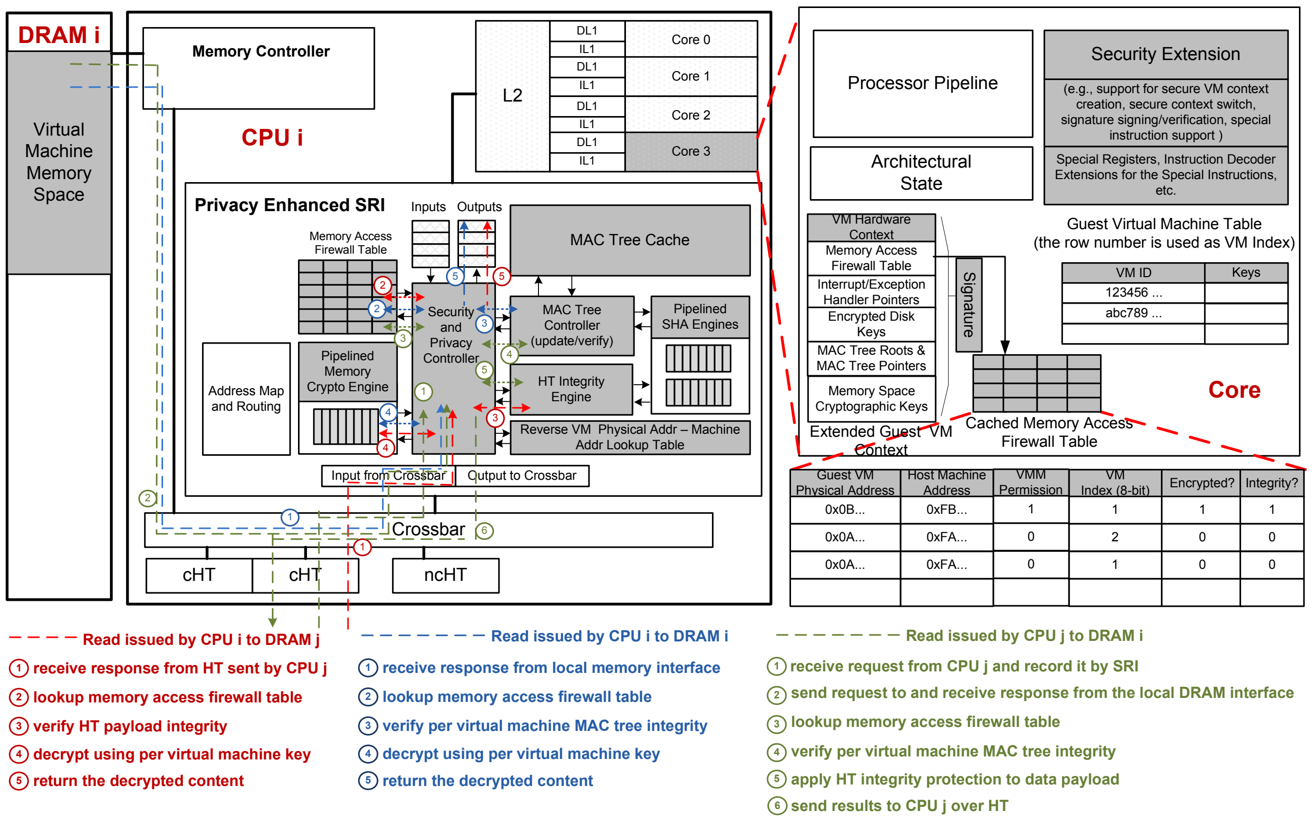
VM memory access firewall. We allow a VM to manage and configure its access control and permission settings for its physical memory space. This is achieved through a memory space property table, or memory access firewall table. In addition to the conventional on-chip processor states, architectural virtual machine context [14] is expanded to include this table as part of the VM context. The memory access firewall table specifies VM's physical address space, memory regions that require the access control, sharing permissions, and cryptographic protections. For example, a VM can indicate that access to a certain memory region is restricted so that even the VMM cannot access that memory region. Additionally, the VM can indicate the memory regions that need to be encrypted (using hardware crypto engines that integrated together with the SRI) and/or protected with integrity mechanisms. Further, a VM can describe which memory regions are shared with the VMM or other VMs.

VM context. A cloud user can can describe VM image properties and settings via a VM context template that comprises the memory space property table. The context template is digitally signed by the cloud user with its private key SK_u, so that the signature can be verified by the processor that is provided with PK_u (via an appropriate public key certifiation mechanism). A VMM can create VM instances from a VM context template, while the trusted processor will confirm that the machine memory space allocation is done properly by the VMM (i.e., there is no violation of memory space access control policies or inconsistency in host machine memory allocation).

A VM can access the context signed by the processor using special instructions. It can verify both the signature and the memory access firewall table during the boot process. A VM can terminate the boot process if any suspicious behavior or mismatch is detected. For each VM, the trusted processor uses its memory space property table at runtime to enforce memory access control and protection policies as specified. When a VM is scheduled for execution, the firewall table is buffered in the cache by the trusted microprocessor core. Access to a VM memory space is verified by consulting the memory access firewall table. To prevent violation from other processors or I/O devices, the memory access firewall table is propagated to other nodes and I/O controllers where every memory access is checked against the cached memory access firewall table. If a memory region requires encryption, the privacy-enhanced SRI will encrypt the memory at unit of cache line size [15, 16]. The SRI is integrated with the pipelined crypto engine (see Figure 2).

Each VM is associated to an ID, which is a 16-byte long UUID, created by the trusted processor, and specified in the signed VM context. The VMM cannot alter or tamper the VM ID without being detected by the trusted processor and the VM. A VMM has its own UUID. Whenever VM_enter (switch from VMM to VM) or VM_exit (switch from VM to VMM) is executed, the trusted processor will automatically reset the ID register. If the ID register stores the VMM ID, the VMM is running; otherwise, the ID register stores UUID of the VM that is currently running. VMM cannot impersonate a VM; whereas a VM can read its ID using a special instruction. L1 cache, L2 cache, and TLB entries are tagged with VM IDs. Since VM IDs are 16-byte long, a lookup table is used by the trusted processor to map each 16-byte VM ID to an 8-bit index and then cache lines are tagged with the 8-bit indices. The index is automatically assigned to a VM by the trusted processor when its context is created. The index value is recycled after the VM is terminated. The maximum number of concurrent guests per server is 256, which is large enough for a single cloud server [17]. The index is not used by the VMM and cannot be tampered by the VMM because the index is part of the VM context, which is protected with a digital signature of the trusted processor. Therefore, it poses no security risk.

VM contexts include both the virtual CPU state and the memory property table. During VM context switch (switch from VM to VMM or one VM to another VM), the cur-


DRAM i
Virtual Machine Memory Space
Memory Controller
CPU i
L2
DL1
IL1
Core 0
Core 1
Core 2
Core 3
Privacy Enhanced SRI
Inputs
Outputs
Memory Access Firewall Table
MAC Tree Cache
Security and Privacy Controller
MAC Tree Controller (update/verify)
Pipelined SHA Engines
Address Map and Routing
Pipelined Memory Crypto Engine
HT Integrity Engine
Reverse VM Physical Addr – Machine Addr Lookup Table
Input from Crossbar
Output to Crossbar
Crossbar
cHT
cHT
ncHT
Processor Pipeline
Security Extension
(e.g., support for secure VM context creation, secure context switch, signature signing/verification, special instruction support)
Architectural State
Special Registers, Instruction Decoder Extensions for the Special Instructions, etc.
VM Hardware Context
Memory Access Firewall Table
Interrupt/Exception Handler Pointers
Encrypted Disk Keys
MAC Tree Roots & MAC Tree Pointers
Memory Space Cryptographic Keys
Extended Guest VM Context
Signature
Guest Virtual Machine Table (the row number is used as VM Index)
VM ID
Keys
123456 ...
abc789 ...
Cached Memory Access Firewall Table
Core
Read issued by CPU i to DRAM j
1 receive response from HT sent by CPU j
2 lookup memory access firewall table
3 verify HT payload integrity
4 decrypt using per virtual machine key
5 return the decrypted content
Read issued by CPU i to DRAM i
1 receive response from local memory interface
2 lookup memory access firewall table
3 verify per virtual machine MAC tree integrity
4 decrypt using per virtual machine key
5 return the decrypted content
Read issued by CPU j to DRAM i
1 receive request from CPU j and record it by SRI
2 send request to and receive response from the local DRAM interface
3 lookup memory access firewall table
4 verify per virtual machine MAC tree integrity
5 apply HT integrity protection to data payload
6 send results to CPU j over HT

Guest VM Physical Address	Host Machine Address	VMM Permission	VM Index (8-bit)	Encrypted?	Integrity?
0x0B...	0xFB...	1	1	1	1
0x0A...	0xFA...	0	2	0	0
0x0A...	0xFA...	0	1	0	0

Figure 2: Architectural support for protecting VM privacy.

rent VM context is preserved by the trusted processor. To prevent tampering of the saved VM context, the trusted processor encrypts sensitive states (e.g., register values) and computes a digital signature for the entire VM context. The cache lines and TLB entries tagged with the swapped-out VM ID (8-bit index) are flushed before the execution returns to the VMM.

A VM with privacy protection can share memory regions with other VMs as long as they share the same cryptographic keys. In addition, a VM can share memory with the VMM by declaring multiple regions with different access policies via the memory access firewall. Similarly, access control policies can be specified for memory regions that are shared between I/O devices and VMs. A VM can open up certain physical address ranges for supporting I/O devices that are allocated to it under hardware assisted I/O virtualization. The settings can be added to the memory access firewall and enforced at runtime. Further, the memory access firewall can be integrated with a virtualization capable I/O controller (e.g., AMD I/O MMU) by extending the address translation mechanism that the I/O MMU already supports. For memory spaces opened for I/O devices by a VM, the contents are not encrypted.

3.4 Privacy-Enhanced SRI

To enable VM privacy protection on multi-socket, multi-processor servers (e.g., dual/quad Opteron processor platforms, Intel dual/quad processor platform based on Quick-path), a set of architectural features are integrated with the system request interface (SRI) as shown in Figure 2. SRI handles and routes requests from processor cores. Pipelined crypto and hash engines are integrated with the SRI to provide encryption protections and tamper-evidence capabilities (see Figure 2). The privacy-enhanced SRI ensures privacy of the VM memory space using hardware-assisted memory encryption. It protects integrity of VM memory space through MAC trees and pipelined hash engines. The MAC trees detect tampering against a VM physical memory space. In order to support distributed DRAM based, multi-processor platforms, global MAC trees are split into sub-trees for the processors (see Figure 3). The sub-tree design avoids racing conditions and communication overhead for MAC tree synchronization, while allowing VMs to access physical resources across multiple processors.

When a VM is scheduled to execute, the trusted processor will verify and load its context. In addition to caching the memory firewall table inside a processor core, the table is also propagated to the SRI and the SRIs of the co-located processors on the same platform. The SRI contains buffers for storing the received memory requests. For each request, the SRI matches the request with the memory access firewall table and finds the corresponding privacy policy. Then, it executes cryptographic operations according to the policy. Each request is associated with an ID (i.e., 8-bit VM or VMM index) that indicates the originator of the request. If an I/O device is allocated to a VM using hardware-assisted I/O virtualization (e.g., I/O MMU or hardware I/O virtualization), requests from the I/O device will be associated with the VM's ID. The association is done by the I/O MMU or a processor core.

The privacy-enhanced SRI responds to both local DRAM access requests (issued from local processor cores) and re-

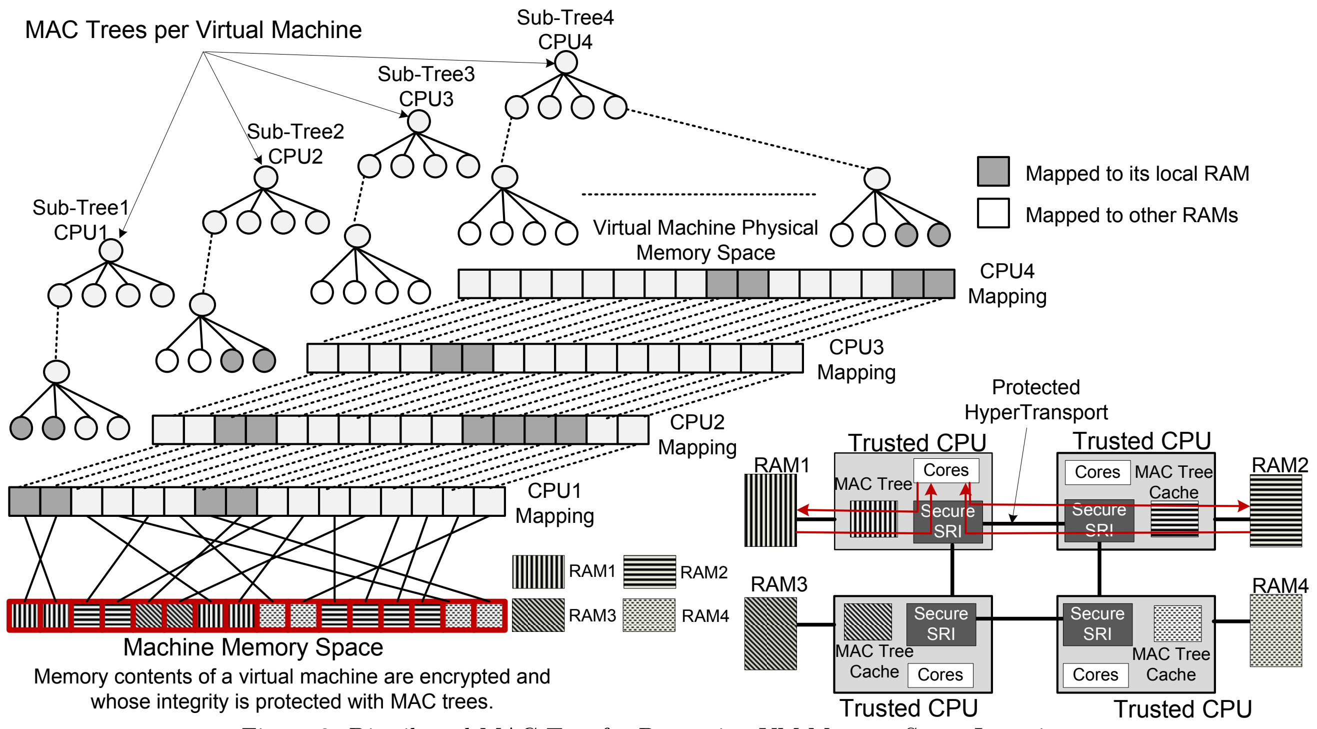


Figure 3: Distributed MAC Tree for Protecting VM Memory Space Integrity.

quests from the co-located processors that are transmitted over the hyper-transport bus. Each SRI is only responsible for integrity of VM memory spaces allocated to the DRAM it connects to (via a crossbar). To prevent tampering of VM memory space through attacks against hyper-transport communications, we enhance hyper-transport communications with tamper-evident integrity protections. The privacy-enhanced SRI uses a HT integrity engine to protect and verify integrity of packets transmitted over the hyper-transport bus. The HT integrity engine computes and appends a message authentication code to each data payload that is sent to a hyper-transport interface via the crossbar. For accelerating MAC tree performance, each SRI further integrates a MAC tree cache (e.g., [18]). Each MAC tree cache line can store 256-bit hash codes.

A VM index is associated to each cache line tag. The privacy-enhanced SRI contains a reverse lookup table that translates machine physical address to the physical address of a matching VM. This lookup table is needed because the memory access requests received over the hyper-transport bus use actual machine physical addresses (instead of VM physical address). Figure 2 illustrates the processing steps that occur inside a privacy-enhanced SRI for both local and remote DRAM read requests.

There is a legitimate concern of VM ID spoofing attacks by co-located untrusted hardware components, where an I/O controller or inter-connected processor sends requests with spoofed VM IDs. To address such threats, each processor will verify, during the hardware boot time, credentials of all the hyper-transport connected I/O controllers and processors to ensure that they are all trusted. For example, if a processor connects to two other processors and one I/O controller, at the boot time, the processor will verify that all hyper-transport neighbors are trusted by sending a challenge to each neighbor via an appropriate authentication method.

3.5 Distributed MAC Trees

In order to protect VM integrity from malicious VMM or untrusted cloud operator, we ensure that VM memory space cannot be tampered by any of them. Merkle tree [19] is a useful data structure for this purpose. However, conventional Merkle trees cannot be applied for our purpose because the VM memory space across multiple RAMs that are controlled by different processors. We design a distributed Merkle tree scheme for our purpose.

The key idea in the design is that each VM has a subtree per CPU and RAM. There is no global root for all the subtrees as shown in Figure 3. When a VM runs on multiple processors, each processor will maintain one sub-Merkle tree for it. The collection of Merkle trees from all processors work together to assure the VM memory integrity. For a specific VM, the processor constructs its Merkle tree as follows (assuming chunk is the minimum data block that is verified by integrity checking for both the VM memory and physical RAM): Let the leaves in the Merkle tree denote the chunks in VM memory (# of leaves equals to # of chunks), and let the leaf be the content in the chunk of VM memory if the corresponding chunk in VM memory is mapped to the physical RAM that is directly controlled by the processor, and the other leaves with a special value that indicates that these chunks are mapped to the RAMs of other processors. A processor can build a Merkle tree based on the leaves, and store the hash values of the tree in its local DRAM. As shown in Figure 3, the illustrated VM runs on four processors where the VM has four Merkle subtrees maintained by the four processors. The MAC tree cache temporarily stores the hash codes of a local Merkle subtree. Note that the four Merkle subtrees have the same structure that is static during the VM's lifetime.

In order to verify the integrity of data in a chunk, the entire chunk is brought into the processor together with the sibling nodes along the path from that leaf to the root, so that the processor can verify it with the stored hash value of the root. When a chunk of data is modified, the processor only needs to update the path from that leaf to the root and stores the hash value of newly updated root in the on-chip secure memory. When a chunk of data in one RAM (say RAM 1) is migrated to another RAM (say RAM 2), processor 1 will set the corresponding leaf with the special value in its Merkle tree to indicate that the data is not stored locally, and update the Merkle tree along the path, while Processor 2 will update its local Merkle tree in a similar fashion.

When a VM is paused, its MAC tree Cache entries need to be flushed and the MAC tree roots need to be swapped out of the secure processor as well. A VM context contains states of virtual CPU. For secure processors, the virtual CPU states are extended to include MAC tree roots and RAM locations of MAC trees. When a VM is paused, its virtual CPU states are saved, certified, and signed by the processor. When VM execution resumes, the secure CPU first verifies integrity of the virtual CPU states (e.g., MAC tree locations and roots) and then restores the states. Consequently, any tampering of the MAC trees will be detected.

However, a malicious VMM may try to violate this designed behavior. For example, it does not copy the contents correctly when a leaf-node is re-allocated. To deal with this, we need to add tamper-evidence to cross-validate the migrated MAC tree branches. In order to detect a malicious VMM that tries to tamper integrity of VM memory space during physical memory migration, we check the hash value of the newly added nodes against the old hash values in the MAC tree it comes from. If they do not match, there is an integrity violation. Leaf-node migration and the associated integrity verification are performed by the trusted CPU using emulated instructions that run under SMM (system management mode), a special mode beyond the reach of operating system and VMM. This procedure is treated as one atomic operation. Whenever an error occurs during the procedure, the whole operation is aborted and the involved MAC trees are restored to the original states.

3.6 Privilege Handling

Our design is a fine-grained privilege system, which is in contrast to the conventional ring-based privilege system in commodity processors. This is necessary because in a ring-based privilege system, VMMs are granted with all of the privileges that a VM can have (inclusive). A VMM can execute any privileged instruction that a VM can execute [14]. As a consequence, the VMM can do everything that a VM can do (including peeking into the VM's contents). In our design, VMMs and VMs have different roles and therefore different privileges. For example, a VM can alter its memory firewall table (i.e., adding or removing property rows, changing policies) using special instructions. A VMM can modify its own memory access firewall table. However, VMM is prohibited from modifying the tables of the hosted VMs; otherwise, the VMM can compromise the VM's privacy by altering the VM's memory access firewall table.

4. PERFORMANCE EVALUATION

We evaluate the proposed design using functional system emulation and cycle-based architecture models. In order to tune up our simulation models, we consider the latency by using the reference RTL implementations.

4.1 Implementations

Crypto unit for memory/HT encryption. We use the Advanced Encryption Standard (AES) for encryption. AES deals 128-bit data blocks with a key of 128/192/256 bits. Specifically, we evaluate the Verilog RTL pipelined implementation [20]. This implementation takes around 30 cycles to encrypt a 128-bit data block, operates at around 330MHz with cost of around 14K LUT, and can achieve over 40Gbps throughput. AES is integrated with the system request interface (SRI). The total area cost is 1,000k gates. Synthesis of the SRI crypto logics using Synopsys design compiler with 45nm technology shows that the total area overhead of the crypto unit is negligible when compared to the size of COTS microprocessors. Details of memory encryption design based on streaming operations of AES can be found in some related work [21, 18]. Reference implementation of the secure hyper-transport is based on an open source hyper-transport core [22].

Integrity verification and MAC tree. In the reference implementation, we use a hierarchical message authentication code tree as described in the earlier sections. A MAC value is generated using SHA-256 hash function [23] for each cache line size memory block of a VM. All the MACs form one layer of nodes and are stored linearly. Similarly, a new MAC value for the next level in the MAC tree is computed by concatenating the new MAC line and the secret key of the application as the inputs to the SHA-256 function until the root MAC is generated. The root MAC is stored inside the processor once the program enters the trusted environment to avoid any potential tampering of the root node. Whenever the external memory of a cache line is modified, the root is updated through a specific path from the leaf node to itself. The MAC tree is 8-way. The leaf level MAC is stored as part of the L2 cache lines. So only the internal MAC tree nodes are cached by the MAC tree cache. Operation and design details of the MAC tree can be found in the related work [24, 25]. Performance simulation of the MAC tree is based on Verilog implementation of SHA-256, synthesized using Synopsys compiler. This design is totally asynchronous and has a gate count of 19,000 gates. The latency for this design is 74ns for 512 bits of padded input (required padding in SHA-256).

Onchip hardware overhead. The onchip hardware resources required include memory access firewall tables, MAC tree cache, virtual machine ID tables, and pipelined crypto engines. Assuming 48-bit physical address space and memory access firewall table size of 96 entries, the hardware cost is about 10.2K bits per core. For 128-bit VM ID and VM ID table of 64 entries (64 VM per server), the hardware cost is about 8K bits per core. The reverse lookup table integrated with the SRI has 64 entries by default. The table is fully associated with 18 bits tag and data. The overall onchip hardware cost remains small when considering the typical transistor count of today's server processors (e.g., commercial Xeon processor has over 2.6 billion transistors).

4.2 Simulation Environment

We use Bochs [26], Simics [27] and FeS2 [28] for functional emulation and performance evaluation. Bochs is used for functional validation, focusing on correctness. Simics+FeS2 are used for performance verification and modeling, focusing on performance evaluation using detailed architectural models.

Bochs models an entire platform including network device, hard drive, VGA, multiple processors, and other devices to support the execution of a complete OS and its applications. It emulates x86 instructions and supports emulation of Intel VMX hardware support for virtualization. For functional verification, Bochs is extended to emulate new instructions and architectural designs including these described in the present paper. Further, VMX support of Bochs is modified to support the proposed features including VM context extension, memory access permission attribute table, etc. We emulate a multi-processor platform, which supports Xen 3.3 and run Ubuntu 8.04 Linux distribution as guest operating systems.

FeS2 provides a detailed architectural model for x86. It supports accurate execution-driven timing-model that includes cache hierarchy, branch predictors, and superscalar x86 out-of-order core. It is implemented as a module for Simics. The memory model is provided by GEMS [29]. FeS2 can decode x86 instructions into uops. Implementation of the uops is based on [30]. Architectural support for hyper-transport platform with privacy-enhanced nodes such as SRI and I/O MMU are implemented in the FeS2. Architectural models of memory encryption and integrity verification are added to the FeS2 architecture model with performance settings derived from the actual Verilog implementations of the memory encryption and MAC tree integrity engines. For hyper-transport, it is based on [22] with encryption and integrity protection support.

4.3 Benchmarks and Machine Parameters

For performance evaluation, we use 20 benchmarks from the Phoronix Benchmark Test Suite [31] and additional benchmark applications, including:

- clamav, diff, gzip, jpython, luindex, snort, sphinx, and xalan from the Phoronix Test Suite, which includes a comprehensive set of applications, covering application domains of scientific computing, compression, cryptography, media encoding, web serving, database query processing, and graphics rendering;
- npbbt, npbft, npbep, npblu, npbmg and npbsp from NASA parallel benchmarks [32];
- php and python from the benchmark suite for PHP and Python; and
- 7zip, gcypt, openssl and vpxenc.

The simulation is performed with a 4-wide out-of-order superscalar processor running at 2GHz and x86 ISA. The simulated platform has four x86 processors that are connected via hyper-transport links. Each x86 processor has its own local physical memory and uses a bimodal branch predictor. Each processor has a 32-entry load/store queue, 128-entry reorder buffer, and non-blocking caches with 16-entry MSHR. The I-TLB and D-TLB have 64 fully associative entries. The memory bus width is 128-bit. The memory firewall has 64 entries, with 32MB minimum setting and 4GB maximum setting. Each region has its own access permission setting. Lookup of the memory firewall table takes 2 cycles. Machine physical memory space is divied into 256MB size regions and allocated to a virtual machine. The simulated SRI with privacy protection uses 64-entry reverse lookup table and 64KB MAC tree cache. The hash cache is configured with 4-way associativity, 32-byte block size, and 6-cycle access latency. Processors are connected via bidirectional hyper-transport links. The maximum bandwidth of the latest 32-bit bidirectional hyper-transport is more than 50 GB/s. The simulation starts when the application passed the initialization stage (using Simics checkpoint support). The cycle-based simulation executes each benchmark application for one billion instructions or until it is completed, depending on which condition is met first.

4.4 Results

First, we consider the impact of hyper-transport for the 20 benchmark applications in Opteron-like multi-processor system in three settings: 0 hop, 0−1 hop (70%:30%), and 0−1−2 hop (70%:20%:10%). In the 0 hop setting, all physical memory pages of a VM are mapped to the DRAM managed by a specific processor. In the 0−1 hop (70%:30%) setting, 70% of the physical memory pages are loaded into the DRAM managed by a specific processor and 30% are loaded into the DRAMs managed by other processors of 1 hop distance. In the 0−1−2 hop (70%:20%:10%) setting, a specific processor manages 70% of the physical memory pages on its DRAM and each DRAM of other processors of 1 and 2 hop distance has 10% of the physical memory pages of a VM. In the experiment, we assume that firewall lookup and TLB translation take 2 cycles. Figure 4 shows the performance overhead of hyper-transport in comparison with 0 hop. On average, the overheads of 0−1 hop and 0−1−2 hop are about 2.7% and 3.7%, respectively. The overhead is caused by accessing physical memory pages in DRAM of other processors.

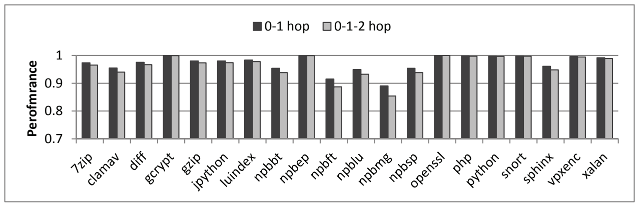


Figure 4: Performance overhead of hyper-transport with different numbers of hops with 0 Hop as the baseline.

Second, we consider the performance of memory access firewall table and the memory space encryption engine via the 20 benchmark applications. Figure 5 shows the performance overhead of hyper-transport and firewall at the same time. We observe that the average performance overheads of 0 hop, 0−1 hop, and 0−1−2 hop are about 0.5%, 3.2%, and 4.1%, respectively. Compared with the overhead of hyper-transport, the overhead of memory firewall is about 0.5%.

Figure 6 shows the performance overhead of hyper-transport, firewall, and privacy protection. We observe that the average performance overheads of 0 hop, 0−1 hop, and 0−1−2 hop are about 1.5%, 4.2%, and 5.2%, respectively. In comparison with the overhead of hyper-transport and firewall, the overhead of encryption is about 1.0%. The privacy-protection overhead depends on the cache miss rates. As shown in Figure 7, the cache miss rates of the benchmark applications are usually low. Figure 8 shows the percentage of each overhead of hyper-transport, firewall, and encryption under 0−1−2 hop. We observe that overheads of the applications vary significantly.

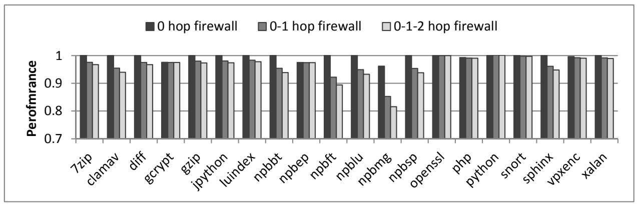


Figure 5: Performance overhead of hyper-transport and memory firewall with different hops.

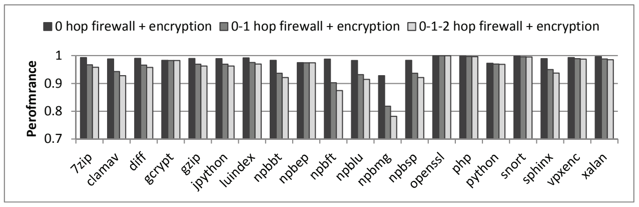


Figure 6: Performance overhead of hyper-transport, memory firewall, and memory encryption with different hops.

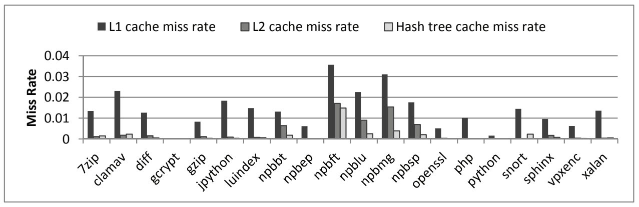


Figure 7: Data cache miss rates.

Third, we consider the performance of the cached memory access firewall tables. Larger firewall cache often has longer lookup latency. In the case that the latency of checking memory access permission is longer than the L1 hit latency, a significant impact on the application performance may occur. Figure 9 shows benchmark performances when the latency of checking memory access permission takes 3 cycles in total (including TLB access time) and the L1 cache latency takes 2 cycles. In our design, data is not returned to the processor core unless the firewall lookup confirms that the memory access is permitted. Under this design, almost every benchmark applications suffer significant performance degradation. This issue can be addressed by two possible solutions. One solution is to design two firewall table caches such that one large but slower firewall table cache is paired with one small but faster firewall table cache. For each memory access, the two firewall table caches are looked up in parallel. The small firewall table cache stores the frequently used lookup entries and can use the LRU policy. For an eight-entry firewall table cache, the access time can be 1 cycle. As shown in Figure 9, this solution can reduce the maximum overhead for all benchmark applications to 1.5%. The other solution is to return data immediately after a cache hit and allow the program to execute speculatively, but only commit the memory access instruction after the permission is verified. We defer a full-fledged investigation of this solution to future work.

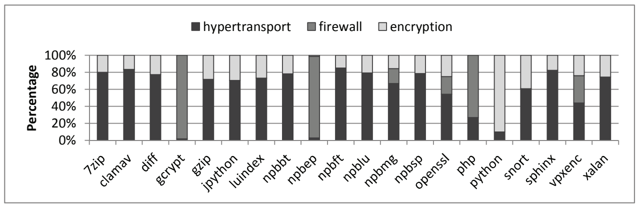


Figure 8: Percentage of each overhead of hyper-transport, memory firewall, and memory encryption with 0−1−2 hop.

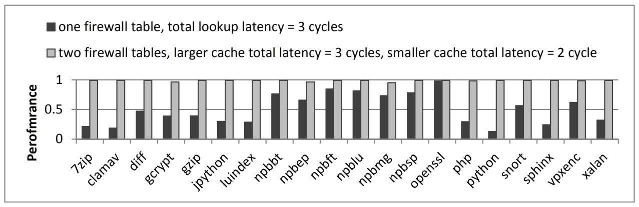


Figure 9: Performance impact of memory firewall table lookup time.

5. RELATED WORK

Trusted computing and related research. There is a body of literature on trusted computing and its applications to secure systems. Trusted computing is centered on the concept of TPM (trusted platform module), which assures secure boot but does not protect confidentiality and integrity of the physical memory space. Therefore, the problem we aim to tackle cannot be solved by using a solution that is centered on TPM. Although systems such as Flicker [33] can be used to protect security-sensitive applications from the underlying malicious software modules (e.g., operating system), it requires to freezing the entire software stack, which makes it inappropriate for cloud computers.

Architectural support for physical RAM privacy. There have been many designs for encrypting physical memory to counter physical attacks (e.g., [34, 35, 36, 21, 37, 18, 38, 39, 40, 41]). Representation examples are: 1) protecting data privacy by performing decryption in parallel to memory access [21]; 2) protecting data privacy and integrity in distributed shared memory multi-processors systems [39] by adapting the Galois/Counter Mode of operation with the counter-mode encryption [38], or by using the address independent counter-mode encryption and Merkle tree built on top of the counters [42]; 3) preventing secret leakage against intrusive memory attack by integrating secret sharing and coding based schemes [40]; 4) a hybrid hardware-software approach to full system security named SecureME [41]. However, it is not clear at all how these solutions can be retrofitted to solve the problem we aim to tackle. Moreover, these solutions do not deal with running VMs in multiprocessor servers of distributed physical memory.

Architectural support for VM security. There have been architectural solutions to protecting applications and data from powerful software attacks [43, 44]. Recent efforts [45, 46] try to address this issue by means of architectural support. However, these architectural solutions primarily deal with software-based exploits from malicious hypervisors, but do not deal with malicious cloud insiders that can launch physical attacks against the DRAM. Moreover, our solution targets multi-processor server platforms (e,g., AMD

Opteron-like systems), which cannot not be handled by these solutions in their current design.

6. CONCLUSION

We have presented the design of architectural support to protect VMs from untrusted cloud vendors and malicious VMMs in multi-processor platforms. The design exploits hardware mechanism (e.g., memory access firewall, privacy enhanced SRI) to enforce VMs' access control and protection policies of their resources. The design uses cryptographic mechanisms to protect the confidentiality of VM memory spaces and system states, and provide secure inter-processor communications and data sharing. Evaluations using cycle-based architecture models show that performance overhead is compatible to the security gains.

7. ACKNOWLEDGMENTS

This work was supported in part by the U.S. National Science Foundation under Grant No. 1111925, and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2012046684). Any opinions, findings, and conclusions or recommendations expressed in this work are those of the author(s) and do not necessarily reflect the views of the funding agencies.

8. REFERENCES

[1] R. Wojtczuk, "Subverting the Xen hypervisor," in *Black Hat USA*, 2008.

[2] K. Kortchinsky, "Cloudburst – hacking 3D and breaking out of VMware," in *Black Hat USA*, 2009.

[3] T. Ormandy, "An empirical study into the security exposure to hosts of hostile virtualized environments," in *CanSecWest*, 2007.

[4] J. Heiser and M. Nicolett, "Assessing the security risks of cloud computing," *Gartner Report*.

[5] P. Mell, "Nist presentation on effectively and securely using the cloud computing paradigm v26," http://csrc.nist.gov/groups/SNS/cloud-computing/index.html, 2009.

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

[7] "Vmware to increase consolidation ratio to 16 vms / core ?" http://virtualization.info/en/news/2010/01/vmware-to-increase-consolidation-ratio.html.

[8] Secunia, "Advisory sa37081 - VMware ESX sever uodate for DHCP, kernel, and JRE," http://secunia.com/advisories/37081/.

[9] P. Ferrie, "Attacks on virtual machine emulators," *Symantec Security Response*, vol. 5, 2006.

[10] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot - a coprocessor-based kernel runtime integrity monitor," in *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, ser. SSYM'04, 2004, pp. 13–13.

[11] M. Green and G. Ateniese, "Identity-based proxy re-encryption," in *ACNS*, 2007, pp. 288–306.

[12] G. Ateniese, K. Benson, and S. Hohenberger, "Key-private proxy re-encryption," in *CT-RSA*, 2009, pp. 279–294.

[13] G. Hanaoka, Y. Kawai, N. Kunihiro, T. Matsuda, J. Weng, R. Zhang, and Y. Zhao, "Generic construction of chosen ciphertext secure proxy re-encryption," in *Topics in Cryptology Ů CT-RSA 2012*, ser. Lecture Notes in Computer Science, vol. 7178. Springer Berlin / Heidelberg, 2012, pp. 349–364.

[14] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig, "Intel Virtualization Technology: Hardware support for efficient processor virtualization," *Intel Technology Journal*, vol. 10, no. 3, pp. 167–177, Aug. 2006.

[15] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas, "Caches and merkle trees for efficient memory authentication," in *Proceedings of the 9th High Performance Computer Architecture Symposium*, 2002.

[16] D. Clarke, S. Devadas, M. van Dijk, B. Gassend, and G. E. Suh, "Incremental multiset hash functions and their application to memory integrity checking," in *Advances in Cryptology*. Springer-Verlag, 2003, pp. 188–207.

[17] "VMware increase consolidation ratio to 16 VMs per core," http://virtualization.info/en/news/2010/01/vmware-to-increase-consolidation-ratio.html.

[18] W. Shi, H.-H. S. Lee, M. Ghosh, C. Lu, and A. Boldyreva, "High efficiency counter mode security architecture via prediction and precomputation," in *Proceedings of the 32nd International Symposium on Computer Architecture*, 2005, pp. 14–24.

[19] R. C. Merkle, "A digital signature based on a conventional encryption function," in *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, ser. CRYPTO '87. London, UK, UK: Springer-Verlag, 1988, pp. 369–378. [Online]. Available: http://dl.acm.org/citation.cfm?id=646752.704751

[20] "Pipelined AES OpenCore," http://opencores.org/project/aes_pipe.

[21] J. Yang, Y. Zhang, and L. Gao, "Fast secure processor for inhibiting software piracy and tampering," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36, 2003, pp. 351–.

[22] D. Slogsnat, A. Giese, M. Nüssle, and U. Brüning, "An open-source hypertransport core," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 1, no. 3, pp. 14:1–14:21, Sep. 2008.

[23] "Secure hash standard," http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf.

[24] C. Lu, T. Zhang, W. Shi, and H.-H. S. Lee, "M-tree: A high efficiency security architecture for protecting integrity and privacy of software," *J. Parallel Distrib. Comput.*, vol. 66, no. 9, pp. 1116–1128, 2006.

[25] W. Shi and H.-H. S. Lee, "Authentication control point and its implications for secure processor design," in *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-39)*, 2006, pp. 103–112.

[26] K. Lawton, "Welcome to the Bochs x86 PC Emulation Software Home Page," http://www.bochs.com.

[27] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson,

A. Moestedt, and B. Werner, “Simics: A full system simulation platform,” *Computer*, vol. 35, 2002.

[28] “Fes2: A full-system execution-driven simulator for x86,” http://fes2.cs.uiuc.edu/index.html, 2007.

[29] M. M. K. Martin and Sorin, “Multifacet's general execution-driven multiprocessor simulator (gems) toolset,” *SIGARCH Computer Architecture News*, vol. 33, pp. 92–99, Nov 2005.

[30] M. T. Yourst, “Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator,” in *ISPASS 2007*.

[31] “Phoronix test suite,” http://www.phoronix-test-suite.com/.

[32] “NAS Parallel Benchmarks,” http://www.nas.nasa.gov/publications/npb.html.

[33] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, “Flicker: an execution infrastructure for tcb minimization,” in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, ser. Eurosys '08, 2008, pp. 315–328.

[34] D. Lie, C. A. Thekkath, and M. Horowitz, “Implementing an untrusted operating system on trusted hardware,” *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 178–192, Oct. 2003.

[35] G. Suh, C. O'Donnell, and S. Devadas, “Aegis: A single-chip secure processor,” *Design Test of Computers, IEEE*, vol. 24, no. 6, pp. 570 –580, nov.-dec. 2007.

[36] 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 17th annual international conference on Supercomputing*, ser. ICS '03, 2003, pp. 160–171.

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

[38] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, “Improving cost, performance, and security of memory encryption and authentication,” in *Proceedings of the 33rd annual international symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 179–190.

[39] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, “Using address independent seed encryption and bonsai merkle trees to make secure processors os- and performance-friendly,” in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40, 2007, pp. 183–196.

[40] J. Valamehr, M. Chase, S. Kamara, A. Putnam, D. Shumow, V. Vaikuntanathan, and T. Sherwood, “Inspection resistant memory: architectural support for security from physical examination,” in *Proceedings of the 39th International Symposium on Computer Architecture*, ser. ISCA '12, 2012, pp. 130–141.

[41] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic, “Secureme: a hardware-software approach to full system security,” in *Proceedings of the international conference on Supercomputing*, ser. ICS '11, 2011, pp. 108–119.

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

[43] J. S. Dwoskin and R. B. Lee, “Hardware-rooted trust for secure key management and transient trust,” in *Proceedings of the 14th ACM conference on Computer and communications security*, ser. CCS '07, 2007, pp. 389–400.

[44] R. Lee, P. Kwan, J. McGregor, J. Dwoskin, and Z. Wang, “Architecture for protecting critical secrets in microprocessors,” in *Proceedings. 32nd International Symposium on*, 2005, pp. 2 – 13.

[45] J. Szefer and R. B. Lee, “Architectural support for hypervisor-secure virtualization,” in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '12, 2012, pp. 437–450.

[46] S. Jin, J. Ahn, S. Cha, and J. Huh, “Architectural support for secure virtualization under a vulnerable hypervisor,” in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44 '11. New York, NY, USA: ACM, 2011, pp. 272–283.